TURBO

THE [illegible]

by

Simon N Goodwin

Jerry J [illegible]

Greg Dillon

DIGITAL PRECISION

[illegible]

[illegible]

QL TURBO MANUAL

7th September 2002

SUMMARY OF CHAPTERS

SECTION 1 - WELCOME

SECTION 2 - COMPILING PROGRAMS

SECTION 3 - FUNDAMENTAL CONCEPTS

SECTION 4 - ADVANCED CONCEPTS

TURBO TOOLKIT: This is documented in separate volumes which discuss new commands, utilities and demonstration routines.

DETAILED CONTENTS

Hierarchical WHEN ERROR and RETRY exception trapping on any QL. How to 'recover' from run-time errors detected by TURBO, or by your own code. Use of DEBUG and TURBO_V.

(T) Customising TURBO.

How to make a version of TURBO optimised for your QL. Changing devices, memory usage and other defaults.

(U) Tuning TURBO for top performance.

Tips and wrinkles for professional users and compiler enthusiasts. Advice on optimisation plus benchmark timings and other vital statistics for train-spotters.

(V) The TURBO story.

Genesis and some measure of Revelation!

SECTION 1 - WELCOME

8th May 2005

INTRODUCTION

This Manual is designed for TURBO version 4.21. There have been many changes since version 3.24.

- TURBO now operates under the Pointer Environment.
- A number of corrections have been made.
- A number of new facilities have been added.

Full comments on these appear in TURBO_UPDATE_TEXT.

This Manual is the original Manual with the minimum of alteration. Thus in several places descriptions of the QL assume that the screen size is the original 512 x 256 and that the information is stored at $20000. Also the descriptions do not always take into account the fact of the Pointer Environment.

I hope that nevertheless you can get some pleasure from this version of the Manual!

NOW READ ON . . . .

CHAPTER A - A CRASH COURSE FOR THE IMPATIENT

SEVERAL SIGNPOSTS

This chapter is a rapid introduction to the TURBO package, intended to get you going as quickly as possible. If you're new to SuperBASIC compilers you should start with Chapter C and then work your way through the text.

If you're familiar with SUPERCHARGE and want a 'TURBO conversion course' you should skip this and start with Chapters B and then E, unless you're really in a hurry.

LOADING AND RUNNING THE COMPILER

You're now ready to compile your first program. A good one to try is:

```
100 DIM a$(40)
110 a$="First Program"
120 PRINT a$
130 SUSPEND_TASK 78
140 STOP
```

It is very wise to LIST and RUN any program that you have not compiled before, to make sure that it works!

Then put the TURBO COMPILER medium in drive 1, or whatever drive you have indicated that TURBO is to run from.

Make sure that there's space on one of your drives for task and report files. Then type:

```
CHARGE
```

The PARSER - the first part of the compiler, which analyses your program to determine its meaning - will load. If anything goes wrong, check that you've got the correct medium in the default drive, and check that the TOOLKIT is loaded. CHARGE is short for SuperCHARGE or TurboCHARGE.

CHARGE works like EXECUTE_A - you can abort the parser at any time by pressing ALT and SPACE (or other keys you set).

THE TURBO COMPILER FRONT PANEL

TURBO may pause while setting up certain controls, if the 'default' values it assumes are not appropriate. TURBO lets you correct the value, then continues (see Chapter E).

The top two lines of the display are used during parsing and code-generation - you cannot change their contents directly. Likewise you can't change the contents of the HELP window which may be at the bottom of the screen.

You can select all the other boxes on the screen individually. Just press the arrow keys to move from one box to another.

It doesn't take long to learn your way around the 'front panel'. You may press F1 to call up 'help'. If you want to start a compilation straight away, just press SPACE.

The controls which you will want to change most often have been arranged around the compile control, where you start. You can change any or all of the default values which TURBO displays after loading, to suit your system or your way of working - see Chapter T.

All of the values in the windows can be changed, either by pressing ENTER and typing a new value, in the case of the central three lines, or by pressing the UP and DOWN arrow keys, to wind numbers back and forth, or by pressing SPACE, to 'toggle' switches. ENTER gets you directly back to COMPILE, from the top or bottom line of options.

The COMPILE and memory allocation controls

If you press the SPACE key at this point, or any other time when the COMPILE control is selected (white on black) TURBO will immediately start to compile the current SuperBASIC program, using the control settings and file-names that you can see on the screen.

Setting the dataspace

The control to the left of COMPILE, labelled Object data, sets you set the amount of space which will be reserved for data in the compiled file. You can alter the dataspace assigned to the task which will be created by pressing ENTER and typing a new value in K, between 1 and 9999. Alternatively you can wind the setting up and down in steps of 1K, by pressing the up or down arrow keys.

Setting the buffer size

The control to the right of the COMPILE button lets you set the size of the 'buffer' TURBO uses to store intermediate information passed between the PARSER - which works out the meaning of a program - and the CODE GENERATOR - which produces an appropriate task.

You can set the buffer size just as you set DATASPACE. A small program requires 1 or 2K of buffer space; requirements for large files vary, but are often about 75 per cent of the SuperBASIC program size.

The OUTPUT file control (Object)

The control above COMPILE, called "Object", lets you set the name of the task file which TURBO will generate, if all goes well. TURBO checks that the default output file-name, or the name supplied as a parameter of CHARGE, does not already exist and can be created. This is the name of the task that, all being well, TURBO will create.

Press ENTER to edit the default name. The left and right arrow keys move and delete (with CTRL), as usual - the up and down arrow keys move the cursor to the end of the line. (Note that on some machines you must use ALT with left and right arrow keys instead of the up and down keys.) Press ENTER again to confirm your entry.

If the file already exists TURBO prints DELETE (Y/N) ? next to the name and waits for you to press a key. Press Y to delete the file, and allow TURBO to continue; press N if you want that file to be preserved.

If you decide to preserve a file, or you specify a file that cannot be created, TURBO must find out what file you really want to create. TURBO can't continue until you type a valid task name.

The top line of controls

The five controls above "Object" affect the code of the task which TURBO will produce. The first four are fairly technical in their function, and will not be explained in this introductory chapter. For details please read Chapter U.

The 'Set ... windows' control

The 'Set windows' control, at the end of the top line of controls, lets you tell TURBO how many of the windows currently open in SuperBASIC are to be 'copied' and opened - with the same dimensions - in the compiled task. The maximum value is the number of channels that the SuperBASIC interpreter has currently allocated, or 32 if that's a silly value.

If your task will open all its own windows you should set this to zero. If the task only uses the default PRINT window, set the control to one. Set 2, if your task uses windows 0 and 1, and 3 if it uses the listing window as well as the other two. Settings beyond 3 will only be needed if your task uses windows with SuperBASIC channel 3 or more, but it doesn't open these itself.

The REPORT control

This control appears beneath 'COMPILE'. The 'report' is the file, printout or display that TURBO produces to indicate how it gets on when it tries to compile your file. You can change this just like the output file name, but in this case it might be sensible to type a device name, such as SER, as the destination for the report.

If the report name is null (no text at all) or DISPLAY, TURBO assumes that you want a report in its own 'rubber window' on the screen.

The bottom line of controls

The controls from the REPORT FILE name downwards - the MANUAL/AUTO/QUIT control, local string control, LIST, SOUND and PAUSE switches - relate to the way that TURBO itself interacts with you. They either control the output of messages or other aspects of TURBO's behaviour during and after a compilation.

The MANUAL/AUTO/QUIT control

This is the leftmost control on the bottom line - it can display various messages. For the time being, leave it set to 'Manual', and read Chapter T later to find out more.

The PHANTOM STRING control

This is another control that can display several messages. All of these relate to the way that un-dimensioned strings in your program are treated. These strings may appear as LOCALs or parameters, but TURBO has no way of knowing whether or not they are used in other contexts.

All strings used by TURBO must have a set maximum length, so that they can be processed efficiently. If a string is used, but never declared in any way, TURBO assumes a maximum length of 100 characters, and issues a warning. If need be you can change this maximum, and get rid of the warning, by adding an explicit DIM.

If in doubt, set the PHANTOM STRING control to 'Create $' if you want LOCAL and parameter strings to be dimensioned globally too, just in case. Set 'Report $' if you just want a list of such names. Set 'Ignore $' and TURBO will assume that you know what you're doing!

The LIST control

If you set this control to YES, TURBO generates an accurate listing of your program as it compiles it, as part of its report, interleaved with any error or auto-corrector messages that the PARSER or CODE GENERATOR may produce.

We recommend that you say YES when experimenting with TURBO, and when compiling a program for the first time, and NO when you expect everything to go fairly smoothly.

The SOUND control

This control enables or disables the clicks produced as you make use of the front panel, and the squeaks when you type something invalid (unless EDIT$ or EDIT% gets there first - see Chapter E) or an error is detected during parsing.

The PAUSE control

Set PAUSE to YES if you want the compiler to wait for a key after reporting errors or auto-corrector messages on the screen - select NO if you want it to carry straight on.

You can't adjust this setting unless you've selected a report to the DISPLAY. At other times the control reads N/A, short for Not Applicable.

HOW TURBO GOES ABOUT A COMPILATION

When you press SPACE with the COMPILE control selected TURBO starts work in earnest. TURBO works in two steps, which are linked automatically if a valid program is being compiled.

First the PARSER converts your BASIC program into 'intermediate code' which is stored in the buffer memory (if space permits) or in a file. It checks the program to make sure that it does not contain errors, issuing reports at appropriate points if need be.

Unlike the SuperBASIC interpreter, TURBO examines every single statement in a program, so it may find problems 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.

Error messages are prefixed by four asterisks, so as to distinguish them from program text. They appear in the DISPLAY window in white letters on a red background, to make them stand out even more.

Sometimes TURBO spots things that it thinks should be brought to your attention, even though they are not errors. These cases are indicated by 'warnings' or 'auto corrector messages' which appear in black on green on the DISPLAY window; they start with four plus signs.

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

At the end of the second pass TURBO reports the total number of errors and warnings. If there were any errors the compiler will stop, without trying to convert the incorrect code into an executable task. Your BASIC will still be in memory. You must correct the errors by examining the report and editing your program in the normal way.

If no errors have occurred when the end of the second pass is reached the code generator is loaded automatically, and the SuperBASIC interpreter comes 'back to life' while it works.

The code generator performs two more passes, reading all the intermediate code twice: once while selecting 'building-blocks' and once while generating machine code for the executable task. A message appears on the main TURBO display when the code generator has finished.

TESTING COMPILED PROGRAMS

You can test a compiled program with the EXECUTE command. If you compiled your program into the file FLP1_TEST_TASK, this command would run it:

```
EXECUTE FLP1_TEST_TASK
```

Type LIST_TASKS, and you will see that your task is running, as well as the SuperBASIC interpreter. TURBO uses the first twelve characters of a compiled task name as the 'name' for LIST_TASKS. Use Control C to switch between cursors, as noted in Chapter D.

Compiled tasks sometimes stop with an error report. As with interpreted programs, there are some errors that can only be detected when code is executed. These, along with all the other TURBO messages, are listed in Chapter F.

The most common message says 'increase dataspace please'. This usually means that you have not given the task enough memory for its variables and other information generated as it runs.

In general, you should not compile programs that you have not tested with the SuperBASIC interpreter - it is much easier to test code interactively with an interpreter, rather than try to resolve problems from a listing and run-time error reports. However TURBO is sometimes a useful tool to 'check' programs, since - unlike the SuperBASIC interpreter - it indicates the position and cause of each error clearly.

Further reading

This chapter is a very rapid tour of the TURBO system. Please read on, when you have time to do so or if you run into problems. Don't let the bulk of this manual put you off. It doesn't mean that TURBO is complicated or hard to use. There's lots of information here, because Sinclair left a lot unsaid. The manual also explains TURBO's many extensions to SuperBASIC, for those who wish to use them.

In practice, TURBO is very simple and 'obvious' when you come to use it - friendlier than the interpreter or SUPERCHARGE, for instance. This manual supplements the software, and helps you to get the best from TURBO.

CHAPTER B - AN INTRODUCTION FOR ENTHUSIASTS

TURBO AND SUPERCHARGE

The aim of this chapter is to give the thousands of people who have used SUPERCHARGE a quick insight into the style and advantages of TURBO. The information should also be useful to anyone who has used commercial SUPERCHARGED tasks and wants to know how they might perform if compiled with TURBO.

The chapter does assume a knowledge of SuperBASIC programming. If you're not yet au fait with the lingo you're best advised to skip this and read the next couple of chapters carefully - they form an introduction for those new to the QL.

GENEALOGY

SUPERCHARGE and TURBO are closely connected, in that SUPERCHARGE was a prototype for many of the routines and data-structures in TURBO. But TURBO is an entirely new product, even though it does many things that SUPERCHARGE could also do.

In this chapter we'll explain the main advances of TURBO compared with SUPERCHARGE, and point out some less-obvious but important improvements in code size, speed and functionality. This is only an introduction -for nitty-gritty details you should consult sections 3 and 4 of the TURBO Manual.

Incidentally, TURBO has virtually NO disadvantages compared with SUPERCHARGE!

HISTORY

It took almost two years - dismal years for Sinclair and the QL - to bring SUPERCHARGE to the market and rave reviews. Since then SUPERCHARGE has proved itself to be a powerful and popular programming tool, in the hands of people like Chas Dillon, Elmar Duensser, Ian Robinson, and organisations such as ABC Electronic, Q-Soft, Pyramide, Sector Software, Shadow Games, Strong, Eidersoft and WD Software. Alas, Medic couldn't stay the course!

BEST LAID PLANS...

We designed SUPERCHARGE as the first, impressive step on a route intended to put virtually all the potential power of the QL at the user's fingertips.

We planned a development programme that would push the speed of compiled tasks a factor of ten beyond SUPERCHARGE, by Christmas 1987.

That ultimate compiler, it was planned, would be able to 'boil down' programs, recognising and changing algorithms and data-types for top speed, making code more concise than even the most diligent human could hope to.

THE TURBO DESIGN

Against this background, we've developed the potential of our design as quickly and completely as possible. The first decision was to hire more people. Dave Newell and Gus Chandler joined the original team of Simon Goodwin, Freddy Vachha and Gerry Jackson.

We checked our market research, and asked a sample group of users to prioritise improvements. An ambitious specification was agreed, close to the planned 'ultimate' design in all but the areas of interactive testing, graphics, mixed-mode arithmetic and 'peephole' optimisation.

We reckon we've got the user-interface, utilities, integers, string-handling, memory usage, extensions, demonstrations and documentation about as good - by any measure - as the QL and five humans can deliver.

Even in the other areas the original SUPERCHARGE performance has been significantly improved; 'hooks' have also been provided, to allow the job to be completed by optional follow-up products: an interpreting debugger, new display driver, and concurrent code-optimisers. A revised version of our best-selling SPRITE GENERATOR is already available; it incorporates changes to take full advantage of existing TURBO optimisations.

Rough prototypes for all the other 'follow-up's exist, but it will cost months and many thousands to bring them all to the market, despite the substantial investment that's already been made. Ultimately the Market will decide (sic).

SO WHAT'S TURBO?

TURBO, together with a QL, is a complete SuperBASIC development kit. All you need to make full use of TURBO is the User Guide that came with your QL, the TURBO TOOLKIT manual, and this volume, the TURBO Manual.

If you're really keen we recommend a couple of technical books - Jan Jones' "QL SuperBASIC - The Definitive Handbook" (McGraw Hill) and Andy Pennell's "The Sinclair QDOS Companion" (Sunshine). You don't need these to get the best from TURBO, but they do provide extra insight into the QL's interpreter and operating system.

Of course 'complete SuperBASIC development kit' doesn't tell you much. TURBO is a state-of-the-art SuperBASIC compiler. Like SUPERCHARGE, TURBO helps you to write fast, powerful software that shows the full power of the QL.

QL WORLD said this: "SUPERCHARGE should help many people to produce quality programs with a fraction of the effort machine-code requires." Your Sinclair added: "SUPERCHARGE really shows that you can do things on the QL that you can't on other machines." With TURBO you can do much more, more quickly, and the results are even better.

TURBO has ALL the features of SUPERCHARGE - speed, multi-tasking, nine digit precision, and others - plus many enhancements and

extensions. TURBO also includes new supporting material - programs, examples and text to make the QL, SuperBASIC and TURBO easier to use.

THE TURBO PACKAGE

The complete set of files comprising TURBO is obtainable on the

Scottish QL Users Group (SQLUG)

www.jms1.supanet.com

The files are all zipped and start with the letters "trbo".

The TURBO package consists of the following seven elements:

1. PARSER_TASK and CODEGEN_TASK

This is the main part of Turbo, a 'compiler' program that converts SuperBASIC into fast 68008 machine-code;

2. TURBO TOOLKIT

This, which is needed to run TURBO, has over 100 new commands, directives and functions for SuperBASIC;

3. UTILITY_TASK

Contains the configurator for TURBO TOOLKIT;

4 .T_CONFIG_DATA and T_CONFIG_LOAD

These together form the utility that allows standard QL Config blocks to be added to Turbo compiled programs;

5. TurboPTR

A set of utilities that allow Turbo programs to use the Pointer Environment. A replacement for QPTR;

6. Associated programs and examples

DATASPACE_TASK - sets dataspace

LIBRARY_MANAGER - extracts routines from DEMOS

MAKE_MODULES - splits a program into modules

TASCOM - turns executable programs into keywords

TURBO_TK_DEMOS - a set of SuperBASIC procs and fns;

ADJ_DS - sets stackspace and dataspace

7. The set of manuals.

TURBO enhances the QL in two ways. The new COMPILER increases the power of existing parts of SuperBASIC, and works more quickly and more flexibly than SUPERCHARGE. The TOOLKIT adds new features to the language, while still allowing testing with the interpreter of all but the most advanced features. You need the compiler to get the best out of the TOOLKIT, and vice versa - but even more so!

The advanced features are discussed in most of the rest of this manual-there's no point in repeating all the detail here. The best way to learn to use TURBO is to experiment with the new features one by one, before testing your knowledge on a combined project. Each feature is a big advance upon SUPERCHARGE facilities.

This is a brief list of new, unique TURBO features:

IMPLICIT DATA TYPES

Any variable name can be designated as a string or integer type, whatever its last character may happen to be. Thus compiled code on ANY QL can use integer FOR loops, and integer and string SELECT. This can make programs several times faster and more concise. They also - particularly in the case of string SELECT - make programs easier to read and write. We haven't bothered with the most mysterious possibility - string FOR statements!

WHEN ERROR

TURBO error-trapping is by far the most comprehensive implementation on the QL - it resembles the exception-handling scheme of the new language ADA. You can trap all errors to a single point, or any number of 'local' error handlers, in a compiled task on ANY QL.

Error handlers can find the line and error number, then RETRY or CONTINUE to an 'outer' error-handler, or jump back into the program, discarding intermediate local and temporary values which might otherwise clog up the system. All other values (global strings, arrays, integers, floats, and loop details) are preserved.

POWERFUL ARRAY HANDLING

Ever needed to make an array bigger or smaller without losing all its contents? That's no problem with TURBO's RUBBER ARRAYS.

TURBO also comes with VIRTUAL ARRAY routines - now the only limit on array size is disk or microdrive space. VIRTUAL ARRAYS are arrays on

disk or tape, but can be accessed much like normal arrays. They use the QL's automatic cache buffering to allow fast access.

LINKED TASKS

'Pipes' allow any number of tasks to PRINT and INPUT to themselves, or one another. Information can be passed to and from any task as parameters of the EXECUTE command.

TURBO's advanced LINK_LOAD feature means that several tasks can share procedures and functions, so you can compile any size of program on a 128K QL. If you change one module you only need re-compile that one before re-running - linking is instantaneous.

Arrays, variables and channels can be shared between separately compiled tasks.

TURBO TOOLKIT

A vast compendium of commands, utilities and functions for SuperBASIC, Supercharge and Turbo BASIC. Useful stuff - not just calls to ROM routines!

Now programs can perform binary random access, even with microdrives; move or search areas of memory (great for graphics programs, editors, utilities etc.); edit variables on screen; control flashing cursors; TYPE-IN things to the keyboard (for function keys, network/terminal systems etc); control and interrogate tasks and devices; define graphics; change fonts; maintain overlapping windows; play continuous 'background' music - plus lots more.

TURBO TOOLKIT consists of a concise machine-code kernel. The toolkit also has its own manual, plus examples (shared with TURBO COMPILER).

IMPROVEMENTS ON SUPERCHARGE

TURBO can handle many SuperBASIC operations better than SUPERCHARGE -faster, more elegantly or more concisely. This list summarises the improvements:

IMPROVED USER-INTERFACE

SUPERCHARGE, like most software, works by asking the user a succession of questions, in an essentially fixed order, until it knows enough to be able to start compiling. This is not too arduous when - assuming you give sensible replies - only three or four answers are required, but it's a pain if you have to answer many more questions.

TURBO gives you much more control than SUPERCHARGE, yet it is easier to use. You can supply information in any sequence. You can see all your 'current' answers at a glance, and change them at will. Yet you only need to press one key to trigger a compilation that uses 'default' options which you can choose yourself.

TURBO has a 'HELP' screen (press F1). If you don't want it you can run another task in that part of the screen. The listing window, if required, can be anywhere on the screen - you can even expand or shrink it interactively as TURBO uses it. There's no 'mess' on the screen as your program is parsed.

You can abort a compilation at any time by pressing a couple of keys of your choice. You can edit or run interpreted SuperBASIC as code-generation takes place. Unless you write very big programs you may not make much use of these features, because TURBO is capable of stunning compilation speeds - often twice as fast as SUPERCHARGE, itself no slouch, e.g. 500 lines parsed in a minute, and code-generation at over 1,000 SuperBASIC lines per minute.

TURBO takes full advantage of extra memory. You can set the 'dataspace' of a task in the program, or when you compile it, or afterwards. You can even monitor dataspace usage as tasks run.

NO LIMIT ON CODE SIZE

TURBO's library is faster yet about 3K smaller than SUPERCHARGE's. As before, routines are only included in a task if they're needed.

Two kinds of code can be produced, and you can swap between them on a statement by statement basis, regardless of the program size.

In-line code is the fastest that the 68008 can run - even faster and much more concise than SUPERCHARGE code. The alternative, 32-bit threaded code gives almost all the speed of pure machine-code, yet it is stunningly concise: a typical 108K interpreted program occupies just 73K when compiled into a stand-alone threaded task by TURBO.

ARRAY AND REFERENCE PARAMETERS

Now you can pass arrays of any number of dimensions (or other variables) 'by reference' - so parameters can be used to pass values back from BASIC procedures and functions. You can even pass arrays to other compiled tasks. 'Local' or 'value' parameters still work fine; in fact string value passing is much faster than SUPERCHARGE could manage.

RELAXED ERROR CHECKING

TURBO can compile most wrongly structured BASIC, as long as it doesn't crash under the interpreter. TURBO is not strict, like SUPERCHARGE; it automatically mimics the behaviour of the interpreter when it finds a structure error, and annotates the listing to indicate what TURBO has done to 'correct' the mistake.

OPTIMISATIONS

String fetch, store, slice and READ operations are up to 3.5 times faster than with SUPERCHARGE - often they are only limited by the speed of

your QL's memory chips! You can now 'slice' literal strings as well as variables.

Integer, Boolean and comparative operations have been re-written; they are significantly faster and more concise than their SUPERCHARGE equivalents. FOR loops are re-entrant, slightly faster and much more concisely coded.

DATA is packed even more densely. TURBO compacts DATA up to four times more than the interpreter, and packs most integer DATA into half the space SUPERCHARGE would use.

Many floating point arithmetic operations have been optimised. The effect of this is most noticeable on systems with extra RAM, as TURBO's new code is 'slugged', compared with the old ROM, by slow access to the standard 128K.

CHAPTER C - BACKGROUND FOR BEGINNERS

WHAT IS TURBO?

TURBO is a tool that helps you to write efficient, powerful programs. It translates programs written in SuperBASIC, the language built-in to the QL, into 68008 machine code - the most powerful code the QL can use.

SuperBASIC is a souped-up version of the most common language supplied with any micro - BASIC, which stands for "Beginners' All-purpose Symbolic Instruction Code".

Standard BASIC was popular but criticised by academics and the self-styled 'professionals' of the computer world. It was considered easy to write, but inefficient and hard to read.

QL SuperBASIC (for Super-Beginners?) is probably the most 'powerful' version of BASIC available for any micro. SuperBASIC is readable and easy to test. Unlike earlier versions of BASIC, it gives natural expression to most popular styles of programming.

It's easy to learn SuperBASIC, and mistakes are usually obvious and easy to fix, especially if you have TURBO to help you.

By contrast, human beings find machine-code programming complicated, repetitive, pedantic and hard to test. It's just the kind of job you should give to a computer - which is why we wrote TURBO.

TURBO fixes all the shortcomings of Sinclair's SuperBASIC, while building upon its many strengths. It gives you easy access to the speed and versatility of machine-code, while still letting you develop your programs interactively, in friendly, familiar SuperBASIC.

TURBO has three main components:

(1) A COMPILER which automatically checks, annotates and translates programs written in QL SuperBASIC into machine code. TURBO tasks may run 100 times faster than 'normal' SuperBASIC. The more care you take when coding or compiling, the larger the speed-up factor you can expect.

TURBO can compile the vast majority of programs that run correctly, under the interpreter, without alteration. TURBO tasks are self-contained and concise. They run on any version of the QL or CST THOR.

(2) You're reading the TURBO Manual - a tutorial and reference guide to the use of TURBO. The Manual is organised into four sections and 22 chapters. The first section contains introductory material. The second holds advice about using TURBO from day to day.

The third and fourth parts are provided for reference. The third explains the workings of standard QL features and their performance when TURBO is used, while section four deals with TURBO's own special features.

(3) TURBO also includes a TOOLKIT - a library of over 100 new commands, directives and functions which greatly increase the power of the QL and SuperBASIC. The TURBO TOOLKIT comes with its own manual, and several utility programs that make TURBO and the QL easier to use.

Recent history

Before the advent of TURBO, Sinclair SuperBASIC had five big flaws. It was poorly documented. This volume - the TURBO Manual - corrects that. It's rather a hefty read, but that's because Sinclair left an awful lot unsaid in the QL User Guide!

Mistakes sometimes confused SuperBASIC, so that programs could fail mysteriously. It was often inefficient, because it did not distinguish to best advantage between different types of data. TURBO banishes both of these faults.

The QL was advertised as being capable of performing several tasks concurrently. If you've ever needed to do two things 'at once', or to switch rapidly from one task to another, you'll know how frustrating it can be to have to wait for one task before you start the next.

Strange though it may seem, very few micros can run several programs at once. As it turned out, even SuperBASIC - contrary to claims - would only run one program at a time. Most of the facilities needed to run multiple 'tasks' are built-in to the QL, but they only work with machine code!

All this changed in November 1985 with Supercharge - the first ever SuperBASIC compiler, and a prototype for many TURBO features. With Supercharge, as with TURBO, you can run as many compiled SuperBASIC programs as you can stuff into memory. TURBO tasks can communicate and co-operate, to an extent that has never before been possible from any BASIC.

The best thing about TURBO is the fact that it produces very fast code. To understand how it does this you must learn a little about the way that computers and programming languages work. That's the subject of the rest of this chapter.

WHAT IS A COMPILER?

TURBO converts SuperBASIC into machine code. The 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.

At the heart of any micro is the processor. This is not so much an 'electronic brain', as an electronic moron.

A processor can only do three things - it can move small values around memory, do simple arithmetic, 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.

Sounds can be generated by moving values to the second processor, printing can be done by moving patterns to the display memory, and so on.

When you turn on your QL, thousands of instructions built-in to the machine read the key switches and perform appropriate actions - step by tiny step.

The QL contains a machine code program called the SuperBASIC INTERPRETER. Just like a human translator, the interpreter converts statements expressed 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 step over and over again as they work.

The QL's interpreter spends more time trying to recognise SuperBASIC statements than it does performing the corresponding action. Interpreters do not 'learn by their mistakes'.

TURBO is a COMPILER. Compared with an interpreter, a compiler is the ultimate in forward-planning. A compiler examines a program to work out its exact meaning, then creates purpose-made machine-code to do the same job - but with no time-wasting checks and searches.

An analogy may help to make the point. Imagine that you hire a worker who is very fast and efficient, but who can't speak your language. You're the boss - the programmer - and the clerk is the computer. Don't worry - it doesn't get any more Orwellian!

Assume that you have details of the clerk's task, written on a set of filing cards. These 'instructions' can be treated like a computer program. The old clerk, being a little dim (like most computers), used to read through them and do the indicated jobs in sequence.

If your new employee was to work like an INTERPRETER he would take a dictionary and start looking up the words on the cards, one by one. When he'd translated and checked a complete card, he'd do the job - and then start again with the next card, and so on.

The worker can start almost at once, using the old cards you can read, but he is slowed by the need to translate every word every time.

It might be better to use the approach of a COMPILER. In this case you give the clerk a biro and a set of blank cards, and let him translate the instructions once and for all. You end up with a copy of the original instructions, in a different language. You can't understand them, but the clerk can work with them more easily.

A bright clerk - or compiler - can incorporate other improvements into the new set of instructions. For instance, an old card might say 'empty the 200th bucket', requiring the worker to count through the buckets from the start to find the correct one.

The new clerk might be able to re-write the instruction, or make a map, so that the correct bucket - or variable, or program line - can be found at once. This saves time, as long as the process does not change. If changes are required a new set of specific instructions must be made.

To summarise: interpreters are quick and simple while you're working out what needs to be done, but compiled instructions are more efficient once you know what to do.

Real interpreters

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

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

When that program is interpreted the value of X is worked out as slowly the thousandth time as it was the first.

Each time, Sinclair 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 for all of line 20.

TURBO looks at the listing of any 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 is much faster.

CHAPTER D - HOW THE QL WORKS

SPACE, TIME AND MULTITASKING

The QL is an unusual home computer, in that it allows several programs, or 'tasks', to run apparently simultaneously. One task is always present - this is the SuperBASIC interpreter, which also serves as the QL's command-line processor, from which other tasks can be invoked or controlled.

Other 'tasks' are programs loaded with the built-in EXEC or EXEC_W commands or the TURBO TOOLKIT commands EXECUTE and LINK_LOAD. These 'extensions' add significantly to the power of the QL's multi-tasking -to the point where the QL programmer can do things that are not possible even in the new computer language ADA which has been developed at colossal expense by the American 'Department of Defense'.

'Extensions' is another important term. SuperBASIC is unusual in that it is a language which can be built up by the user. New commands or functions can be written in machine-code in a standard format, and then used by the SuperBASIC interpreter, or any compiled SuperBASIC task. TURBO is supplied with a TOOLKIT of powerful SuperBASIC extensions, which include commands, functions and compiler directives.

Tasks differ from SuperBASIC program source or machine-code loaded with the LBYTES command, in that more than one task may appear to be running at any one time. This concurrent execution of several programs is called 'multi-tasking'. The TURBO compiler and its associated code-generator are themselves tasks.

All programs compiled by TURBO 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.

Tasks have two parts - code space and data space, a term that will be used so often that we shall abbreviate it to one word!

Every program compiled with TURBO requires at least 6 kilobytes (K) of memory. This space consists of about 5K of code used to set up the task, plus routines needed in all compiled programs and code to handle and report errors. A further 1K of space for data (channel tables and system information) is always allocated. You can alter the size of the dataspace associated with any task, as we shall explain later in this chapter.

The total size of a compiled program can be anything from 6K upwards. In general, compiled code is more concise than the original BASIC program, unless very small programs are considered. The exact ratio of sizes will vary from one program to the next. Very large programs are often MUCH smaller than their interpreted counterparts.

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, DEVICE_STATUS, is provided to give you this information. This function is part of the TURBO TOOLKIT, and is documented in its User Manual.

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 a 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, but several tasks may read a file concurrently. The QL reports 'in use' if a task tries to write to a file which is already being used 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. TURBO TOOLKIT contains new commands which make it easy to save and restore areas of the display from within a task.

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 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 task is loaded into the QL's memory, space is reserved for the data which the task will generate, as well as for the program code.

TURBO uses this dataspace 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 'please increase dataspace'.

By default, 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 needlessly. You can 'dial up' a dataspace setting on the 'front panel' display of TURBO when you compile a program.

It is easy to change the amount of data space which will be allocated to a specific task. You can also change the default value assumed by the compiler's front-panel; Chapter T explains how to do this.

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

```
EXEC FLP1_DATASPACE_TASK
```

Once the task 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.

You can stop the DATASPACE program at any time 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.

5160 code bytes, 2048 data bytes. New data size (in Kilobytes) ?

The QL asks you 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 1K or more than 850K of data space for any task. Compiled programs require a minimum amount of space in which to set up essential tables, and even a fully expanded QL cannot run a 600K task. If you type a value less than 1K or more than 850K you are asked to try again:

Size should be between 1K and 850K!

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.

DATASPACE uses a 'transient window'. It stores the original contents of the part of the display that it intends to use in its own 9K DATASPACE when it loads, and restores that display later. This will not happen if you reduce DATASPACE's own dataspace below 9K.

PLEASE DO NOT INCREASE the dataspace of DATASPACE beyond 9K - this will do no good, and could make it impossible to re-load the utility to reverse what you have done. We congratulate the perverse genius who first alerted us to this possibility.

A high-speed summary of the QL design

The core of the QL design is the SCHEDULER. This is a routine which shares out processing time between loaded tasks. TASKS are independent QL programs. They are created with EXECUTE or LINK LOAD and may stop of their own accord, or because of the action of another task.

Each task is identified by two numbers: a task number, which indicates its position in the scheduler's table of tasks, and a 'tag', which distinguishes between different tasks that re-use a certain position. A task also has a PRIORITY, which determines the proportion of the total processing time which it can use.

When a task is loaded it is allocated an area of memory. The first part of this is used to hold the CODE - the instructions that the task performs - and the second, adjacent part is reserved for DATA generated by the task. The code area contains all the information from the task file. The data area is initially empty. Its size depends upon the DATASPACE associated with the task.

Tasks may also own other areas of memory, explicitly reserved with RESPR or ALLOCATION, or implicitly reserved by the system. These are taken from the HEAP - the other end of the QL's memory from task code and data space.

When you turn on the QL the SuperBASIC interpreter is the only task running. It cannot be removed, so it always has the number and tag 0,0. Unlike all other tasks, the SuperBASIC interpreter's data area can expand or shrink.

SuperBASIC provides the COMMAND LINE INTERFACE from which the system is controlled. Task 0,0 is the one interrogated when TURBO tasks needs details of SuperBASIC EXTENSIONS.

Every task may own CHANNELS - connections that allow information to be read from or written to DEVICES. The QL system assumes that the first two channels are always available - these correspond to the SuperBASIC interpreter's channels 0 and 1, which must never be closed.

Interpreted and compiled SuperBASIC tasks use a CHANNEL NUMBER, usually between 0 and 31, to identify a particular channel among those owned by that task. These numbers may be re-used as the task opens and closes channels. Each task has it's own set of channel numbers.

DEVICES are routines that let you send or receive data in a standard way. Each device can cope with one or more CHANNELS. Devices often allocate memory on the heap as they work. Devices usually correspond to physical things, like serial ports, disk interfaces, or the display.

STACK SPACE

Part of the dataspace of each compiled task is reserved for the system stack. The normal amount set by TURBO is 350. This will not always be enough. For example programs making use of the Qmenu extension will need more, perhaps up to 800 bytes. The stack space can be changed in a compiled program by using the program ADJ_DS.

As with DATASPACE_TASK ADJ_DS is activated by a command such as:

```
EXEC FLP1_ADJ_DS
```

Pressing "p" or clicking on Pick a File will give rise to s request for a file name. This will be by way of the Menu Extension if is present. Otherwise you will have to type in the name. If it is a valid file its name and dataspace will be printed. In addition, if was compiled by TURBO its stack space will also be printed.

You can now proceed to change the dataspace by pressing "d". Once this is done, and the program was compiled by TURBO you can decide to change the stackspace by pressing "y". If you press "n" only the dataspace will be changed.

If you had started by changing the stackspace, assuming that that option was available, you be asked whether or not you wanted to change the dataspace.

The stack space can be set to an even nember of bytes from 350 to 9998. The dataspace can be altered from 2K to 9999K. This is a bigger range than allowed by DATSPACE_TASK described above.

Remember that if you ask for a large amount of stackspace you should check that the stackspace is sufficent to cover that amount.

SECTION 2 - COMPILING PROGRAMS

8th May 2005

CHAPTER E - TURBO DRIVING LESSONS

TOURING WITH TURBO

This chapter is a complete tutorial introduction to the TURBO package, with worked examples. The best and fastest way to learn to use TURBO is to work your way through this chapter - which is intended to be interesting, rather than onerous.

This chapter will explain two things:

1. How to load and use the TURBO TOOLKIT.
2. How to compile programs with TURBO.

HOW TO LOAD AND USE THE TURBO TOOLKIT

TURBO TOOLKIT is loaded by doing an LRESPR FLP1_ ***

If you have added TURBO TOOLKIT to your boot program, it will load automatically when you press F1 or F2. It displays a brief message when it has loaded, telling you its version number. You can also check this at any time with one of the toolkit functions, just as you can use VER$ to find out the version of your QL ROM. Type:

```
PRINT TK_VER$
```

to establish that the toolkit has loaded, and to see the version number again.

We can experiment with some of the powerful commands:

```
PRINT SEARCH_MEMORY(0,49152,"Sinclair")
```

This searches through the QL's ROM, looking for the name of the instigator of the QL. The number printed corresponds to the address, in the ROM, where the name 'Sinclair' appears. Be careful to spell it correctly - with one capital letter at the front - or it will not be found, and the function will print zero.

Among many other tricks and treats, documented in its own User Manual, TURBO TOOLKIT lets you move the contents of memory at great speed. You can duplicate the top half of the QL's screen at the bottom with this command:

```
MOVE_MEMORY 16384, 131072 TO 131072 + 16384
```

This works because the QL keeps its display in 32768 bytes of memory, from address 131072 onwards. The command works because the first number, 16384, corresponds to the number of bytes to be moved, and to half the size of the screen area. The next two values, separated by TO, tell the QL where to move memory FROM and TO.

This should look impressively fast - but it will be even faster if you compile it with TURBO! Compiling is the next step in this tutorial.

NOTE that this trick will not work unless the screen size is 512 pixels by 256 pixel and the scren information is stored at address 131072.

COMPILING PROGRAMS WITH TURBO

The story so far

TURBO converts slow SuperBASIC programs into machine-code - the fastest language the QL can handle. TURBO is itself compiled, so it works very quickly.

A compiler cannot speed up mechanical operations, like reading and writing from disk, but it can accelerate all the operations that are performed in memory, once data has been read or written.

TURBO can do just about anything in memory much faster than the interpreter - but the key to making good use of TURBO is to take advantage of its greatest strengths, so we'll spend a moment reviewing these.

We have not made quantatative statements here, because TURBO performance varies widely between programs and QL systems, and the discussion of these variations is the main purpose of this Manual. However, one point should be clear - if you want to make a program faster, compiling it with TURBO is the best thing you can do!

TURBO compiles floating-point (decimal) arithmetic to run at much the same speed as its predecessor SUPERCHARGE. The limit, in this case is imposed by the processor's set of instructions, which do not allow floating-point numbers to be handled directly. TURBO must therefore use fairly well-established methods to process them. However TURBO does contain some 'optimisations' beyond SUPERCHARGE, which can increase its floating-point speed in many common cases.

TURBO is much more impressive than SUPERCHARGE in its handling of text and string operations, which dominate the processing time of many programs. TURBO can fetch and store strings 3.5 times faster than SUPERCHARGE. Handling of very short strings - less than about 20 characters - is not accelerated so much, because it takes as much time to find the text as it does to process it.

TURBO can also handle integers - whole numbers, explicitly stored as such - with blinding speed, often twice as fast as SUPERCHARGE and much faster than the QL could possibly perform floating-point arithmetic.

Finally, TURBO, and TURBO TOOLKIT, allow you to do many things that are simply impossible in standard interpreted QL SuperBASIC.

Compilation countdown

You must have the version of TURBO TOOLKIT supplied with your compiler loaded before you start to compile a program.

If you try to compile something without loading the TOOLKIT at all, TURBO will print a list of commands that it needs but can't find. Reset the system, load the TOOLKIT and try again.

Now your'e ready to compile your first program. LOAD the program and type

```
CHARGE
```

If you want to set the name of the output file you can do this now by typing it after CHARGE. For example

```
CHARGE ram1_my_prog
```

will set the output file name to "ram1_my_prog".

The PARSER - the first part of the compiler, which analyses your program to determinhe its meaning - will load. If it doesn't, check that the default device, set by DEFAULT_DEVICE or PROG_USE contains PARSER_TASK.

TURBO performs a number of checks as it starts to run.

CHANNEL CHECKING

If channel numbers above 31 have been used in the current SuperBASIC program, TURBO will warn you.

TURBO tasks can use no more than 32 channels, but this should not be a practical limitation - the message usually indicates that the SuperBASIC programmer has been using channel numbers arbitrarily rather than systematically.

Every unused channel-number wastes at least 40 bytes of memory, so you should correct the program to make it more efficient under the interpreter, as well as suitable for compilation.

OUTPUT FILE CHECKING

TURBO checks that the default output file-name, or the name supplied as a parameter of CHARGE, does not already exist and can be created. This is the name of the task that, all being well, TURBO will create.

If the file already exists TURBO prints DELETE (Y/N) ? next to the name and waits for you to press a key. Press Y to delete the file, and allow TURBO to continue; press N if you want that file to be preserved.

If you decide to preserve the file, or you specify a file that cannot be created, TURBO must find out what file you really want to create. It puts the cursor at the end of the existing name and allows you to edit it, just like a SuperBASIC line.

The left and right arrow keys move and delete (with CTRL), as usual -the up and down arrow keys move the cursor to the end of the line.

(On some computers you may have to use the ALT key with left and right arrows instead of the up and down keys.)

TURBO can't continue until you type a valid task name.

REPORT NAME CHECKING

TURBO checks that the default report name is valid. In the case of a file, it must be possible to create that file. If the report name is null (no text at all) or DISPLAY, TURBO assumes that you want a report in it's own 'rubber window' on the screen. Otherwise TURBO needs a valid QL device or file-name, so that it knows where to send the report.

If the file already exists TURBO prints DELETE (Y/N) ? next to the name and waits for you to press a key. Press Y to delete the file, and allow TURBO to continue; press N if you want that file to be preserved.

If you decide to preserve a file, or you specify a name that cannot be used, TURBO must find out where you really want to send the report. It puts the cursor at the end of the existing name and allows you to edit it, just like a SuperBASIC line. Type a valid name, or scrub out everything on the line with Control Left Arrow and press ENTER to select the default, DISPLAY.

TURBO's 'FRONT PANEL' DISPLAY

TURBO has a sophisticated but easy-to-use 'front panel' display. The idea is to allow you to change lots of options if you want, but not to FORCE you to answer a succession of questions needlessly.

A simple HELP window appears at the bottom of the screen when you press F1. If you don't call up this window you can use that part of the screen for other task displays.

Other tasks will run happily with TURBO; you can use the SuperBASIC interpreter while code is generated, but not while the PARSER is analysing your program. TURBO works very fast because it reads your program directly from memory, without intermediate files - this would not work if you were altering the program as TURBO tried to analyse it!

The top two lines of the display are used during parsing and code-generation - you cannot change their contents directly. Likewise you can't change the contents of the HELP window. But you CAN select all the other boxes on the screen, just by pressing the arrow keys to move around the display.

It doesn't take long to learn your way around this 'front panel'. If you want to start a compilation straight away you can do so - just press SPACE while the COMPILE control is selected. The controls which you will want to change most often have been arranged around the compile control,

where you start.

You can change any or all of the default values which TURBO displays after loading, to suit your system or your way of working. You can also change other parameters, such as the character size used in display reports, which is documented in Chapter T of this manual.

If a control is irrelevant - as in the case of the PAUSE control when you don't want a DISPLAY report - it is locked and marked N/A, which stands for 'Not Applicable' and means you're not, at present, allowed to select or alter that control. Similarly you can't set the number of windows to be copied to more than the current number of channels open in SuperBASIC - this stops you asking the system to copy channels that are undefined.

All of the values in the windows can be changed, either by pressing ENTER and typing a new value, in the case of the central three lines, or by pressing the UP and DOWN arrow keys, to wind numbers back and forth, or by pressing SPACE, to 'toggle' switches. You can control most things with a joystick plugged into CTRL1, or our 'mouse', if that's what turns you on.

If you are on the top or bottom line of the panel the ENTER key will get you straight back to the COMPILE option in the middle. There's one exception to this rule, discussed later - it only crops up if you want to confirm your intention to ABORT the compiler.

Hidden depths

There is an underlying structure to the layout of the front panel which may not be immediately apparent. It is worth pointing this out before we discuss the nitty-gritty details of what each control does.

All the controls in the two lines above COMPILE affect the code of the task which TURBO will produce.

All of the controls in the two lines below COMPILE - such as the List, Sound and Pause switches - relate to the way that TURBO itself interacts with you. They either control the output of messages or other aspects of TURBO's behaviour during and after a compilation.

The last two controls are on the same line as COMPILE; the left-hand one, "Object data:", affects the task, whereas the right-hand one "TURBO buffer:", affects TURBO's own behaviour but not the task.

Most TURBO users read from top to bottom and left to right (SuperBASIC is tough going unless you adopt this convention) so the code controls are all 'before' the COMPILE switch and the behaviour controls come 'after' it.

The local string control rather spoils this, as it lets you tell TURBO to create strings on one of its settings. The other two control the listing so - on balance - we put it at the bottom, although we did seriously consider making it run up and down the screen as appropriate!

The front panel is not just pretty, or clever - it is much more flexible and friendly than a list of questions. You can supply information in any sequence, and you can always see the valid answers, or defaults, that are currently set.

The front panel involves deceptively little code - about 150 lines of simple SuperBASIC - so it doesn't use up lots of your memory. This economy is possible because it is implemented as a 'finite-state automaton' - a kind of digital ball-race, where characters drop in at the top and have different effects depending upon previous 'events'.

The OUTPUT FILE, REPORT FILE and COMPILE options have already been discussed, so they won't be illustrated again in this Chapter.

Briefly: you can edit a file or device name by pressing ENTER when the control is selected. You're not allowed to switch to another control till you've specified a sensible file or report device. SPACE triggers a compilation when COMPILE is selected.

THE DATASPACE or STACK SIZE CONTROL (Object data or Stack size)

You can alter the dataspace assigned to a file by pressing ENTER and typing a new value in K, between 1 and 9999. Alternatively you can wind the setting up and down in steps of 1K by pressing the up or down arrow keys.

If there is a sensible DATA_AREA directive in your program it will over-ride the value set with this control.

By pressing SPACE you can toggle the panel between Object data and Stack size.

You can alter the amount of user stack which will be assigned to the compiled program by pressing ENTER and typing a value from 1 to 9998. Alternatively you can increase or decrease the setting in steps of two by pressing the up or down arrow keys. Values 350 to 9998 will result in a stack size of that amount, rounded up to an even value. Values below that will result in Dflt appearing in the panel. This means that the value stored in Codegen_task will be used instead. This default value can be set to a value between 350 and 9998 by configuring Codegen_task.

SETTING TURBO'S BUFFER SIZE (TURBO buffer)

TURBO uses a 'buffer' to store intermediate information passed between the PARSER - which works out the meaning of a program - and the CODE GENERATOR - which produces an appropriate task. SUPERCHARGE used to put information on the display, which was economical but unsightly.

You set the buffer size with the up and down arrows or by pressing ENTER and typing a sensible number, as with the DATASPACE control.

TURBO can use any size of buffer up to the limits of free memory on your computer - the minimum is 1K. If the PARSER fills up the buffer it creates a temporary file by adding TEMP to the task name, and then uses the buffer as a temporary store to avoid constant file-access. The buffer MUST be large enough to hold all of the longest quoted string in your program - 1K is usually ample for this, though if you are using the option to include a file of extensions (see Chapter Q) the buffer must be large enough to hold the largest extension file with a bit over.

The bigger the buffer, the faster compilation will occur - but there's no point allocating space which will not be used. A small program requires 1 or 2K of buffer space; requirements for large files vary, but are often about 75 per cent of the SuperBASIC program size.

THE MANUAL/AUTO/QUIT CONTROL

This control will normally show "Manual", but if you start TURBO with

```
CHARGE \
```

it behaves as if you had immediately pressed SPACE to start compilation. If you are quick you might just see "Auto" in the leftmost panel before the panel disappears to make way for the information shown during Parser's operation. However if there is a problem with the settings compilation does not immediately start and TURBO waits for manual intervention. For example the Object file might already exist. In this case you will have time to see "Auto" displayed.

Whether "Auto" or "Manual" is displayed in the panel, moving to it and pressing SPACE will cause it to toggle between "Quit now?" and "Manual". If you press ENTER when Quit now? is showing the program aborts. This is the only case where pressing ENTER when a panel on the bottom line is selected does not take you straight to COMPILE.

THE PHANTOM STRING CONTROL (Report $/Create $/Ignore $)

This is another control that can display several messages. All of these relate to the way that un-dimensioned strings in your program are treated. These strings may appear as LOCALs or parameters, but TURBO has no way of knowing whether or not they are used in other contexts.

All strings in TURBO must be dimensioned, so that they can be processed efficiently. If a string is used, but never declared in any way, TURBO assumes a maximum length of 100 characters, and issues a warning. You can change this maximum, and get rid of the warning, by adding an explicit DIM.

Set the PHANTOM STRING control to 'Create $' if you want LOCAL and parameter strings to be dimensioned globally too, just in case. Set 'Report $' if you just want a list of such names. Set 'Ignore $' and TURBO will assume that you know what you're doing!

CONTROLLING THE GENERATION OF A LISTING (List:)

TURBO can generate an accurate listing of your program as it compiles it. This listing is sent to the report file, with any error or auto-corrector messages that the PARSER or CODE GENERATOR may produce.

Use the SPACE key to swap this control between YES (produce listing) and NO (don't). TURBO goes fastest when it does not produce a listing, but problems are easier to identify if you have one.

We recommend that you say YES when experimenting with TURBO, and when compiling a program for the first time, and NO when you expect everything to go fairly smoothly.

THE SOUND CONTROL (Sound:)

This control enables or disables the clicks produced as you move around the front panel, and the squeaks when you type something invalid or an error is detected during parsing. It has immediate effect.

The only sounds unaffected by this control are the ones produced when you make a gross error when editing a name, dataspace value or buffer size. These sounds crop up if you type more characters than will fit in the window, or try to 'escape' without entering a sensible value by pressing the up or down arrow keys when editing. They are produced by the TURBO TOOLKIT EDIT$ and EDIT% functions, rather than by TURBO itself.

BEHAVIOUR AFTER AN ERROR OR WARNING IS DISPLAYED (Pause:)

If you request the default report destination, DISPLAY, you have the option of getting TURBO to wait for a key-press after every error or auto-corrector message is displayed. This saves you having to press CTRL F5 (which also works) to pause the listing as it rolls up the screen.

Set PAUSE to YES if you want the compiler to wait for a key after errors or auto-corrector messages - select NO if you want it to carry straight on.

You can't adjust this setting unless you've selected a report to the DISPLAY - the control reads N/A, short for Not Applicable.

THE STRUCTURE CONTROL (Freeform/Structured)

This control allows shorter programs to be produced if the SuperBASIC program is "structured". A structured program is one where no program lines appear after a procedure or function. There are three exceptions to this rule.

1. REMarks are allowed anywhere. 2. DATA lines can appear anywhere 3. DEBUG and DEBUG_LEVEL can appear anywhere.

Unless you are completely sure that your program is correctly structured, it is better to use the setting "Freeform".

CODE SIZE CONTROL (<64K / >64K)

The option <64K gives smaller faster programs than the other setting. However, if your program is too big the code generator will stop with an error message and you will have to recompile eith the ">64K" option, or reduce the size of your program.

THE DIAGNOSTICS CONTROL (.... Nos)

This control determines whether or not TURBO puts line-number information in the generated task. The SPACE key changes the setting.

If you set this control to "Display Nos" the compilation will proceed more quickly and the compiled task will be smaller - often by a significant margin, on large programs - but no line-numbers will be included in the program although they will be displayed during parsing and code generation. Run-time reports will use the line number zero wherever an error occurs in the program.

You should leave this control set to "Include Nos" unless you are very confident that your program will not fail when it is executed.

There is a third setting, "Omit Nos". This is like "Display Nos", but it reduces the time taken for parsing because no line numbers are displayed at the top of TURBO's window during passes 1 & 2.

THE OPTIMISATION CONTROL (BRIEF/REMs/FAST)

This control lets you 'tune' the tasks generated by TURBO to make them faster or shorter.

Three settings, swapped with the SPACE key, are allowed; BRIEF, REMs and FAST. The first is the most general. The second is the most efficient if you are willing to analyse your program in detail.

Short integer programs that must run quickly will work best if compiled with this control set to FAST. It may also help to set "Omit Nos" ,if you must have the maximum possible speed and you know that the program works properly.

There is not space to discuss the effects of this control in this chapter - you should refer to Chapter U for full details. It is usually best to set this control to BRIEF unless you've read Chapter U and have a specific effect in mind.

THE WINDOW COPY CONTROL (Set .... windows)

This control lets you tell TURBO how many of the windows currently open in SuperBASIC are to be 'copied' and opened - with the same dimensions - in the compiled task.

The control is set with the UP and DOWN arrow keys. The maximum value is the number of channels that the SuperBASIC interpreter has currently allocated, or 32 if that's a silly value.

If your task will open all its own windows you should set this to zero. If the task only uses the default PRINT window, set the control to one. Do that if you're working through the tutorial. Notice that TURBO is pedantic about grammar! Set the control to 2 if your task uses windows 0 and 1, and 3 if it uses the listing window as well as the other two. This is bad form - the listing window is meant for listing and editing - but it is often done.

Settings beyond 3 will only be needed if your task uses windows with SuperBASIC channel 3 or more, and it doesn't open these itself.

If you choose to copy any windows you should be aware that in some cases the copied window may be out of range when run. This will happen if a program is compiled with a copied window outside the 512 x 256 range and run on a machine set to 512 x 256.

WHAT HAPPENS DURING A TURBO COMPILATION

When you press SPACE with the COMPILE control selected TURBO starts work in earnest. TURBO works in two steps, which are linked automatically if a valid program is being compiled. The second step is only performed if the program does not contain errors. Both steps are invoked by the CHARGE command - you don't have to trigger them separately, though you can if you wish.

STEP 1 - TURBO parsing

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

The parser scans your program from beginning to end twice. Each scan is called a 'pass'. Unless you have chosen "Omit Nos", TURBO displays the number of each program line as it is analysed.

During the first pass through your program TURBO 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. The first pass does not report errors.

In 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. Error messages are prefixed by four asterisks, so as to distinguish them from program text. They appear in the DISPLAY window in white letters on a red background, to make them stand out even more.

Sometimes TURBO spots things that it thinks should be brought to your attention, even though they are not errors. At other times it may find a minor error or misunderstanding of SuperBASIC's grammar, which is discussed in Chapter I. These cases are indicated by 'warnings' or 'auto corrector messages' which appear in black on green on the DISPLAY window; they start with four plus signs.

You can change the size of the DISPLAY window while it is being used by TURBO - this is useful if you are doing something else with another task while parsing takes place. Press ALT and UP to reduce the size of the window by one line, and ALT and DOWN to make it larger. These controls only work while data is being written to the window, but that shouldn't matter - a window that is not being used is unlikely to cause offence! The keys to operate these controls can be configured using Config or Menuconfig on Parser_task.

If you asked for a listing, error and auto-corrector messages are inserted just after the point at which each error is discovered. In some cases the true cause of the the error may be at some point earlier in the program, but the point at which the error is reported is still a useful indication of the exact nature of the error. There is a full list of messages in Chapter F.

Unlike the SuperBASIC interpreter, TURBO examines every single statement in a program, so it may find problems 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, TURBO 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 TURBO reports the total number of errors and auto-corrector 'warnings' that it found. If there were any errors the compiler will stop, without trying to convert the incorrect code into an executable task.

If your program fails to compile you must correct the errors by examining the report and editing your program in the normal way. When the errors are gone, and you have heeded any warnings, you can re-start TURBO and try again.

STEP 2 - 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. If the compilation was initiated by CHARGE the code generator is loaded automatically, and the SuperBASIC interpreter comes 'back to life' while it works.

The code generator performs three more passes, reading all the intermediate code thrice: once making changes to optimise the code, once again while selecting 'building-blocks' and finally while generating machine code for the executable task. As before, the line number being analysed is shown, unless you have turned off diagnostics.

A message appears on the main TURBO display when the code generator has finished. This will be written on a green background if all is well, and on red if some problem has occurred. The exact problem is reported - the most common fault is running out of space on the destination device.

TESTING COMPILED PROGRAMS

You can now run the compiled program by typing EXECUTE. You don't need to type the name of the program at this stage, because EXECUTE picks the program just compiled if no name is given. EXECUTE_A and EXECUTE_W work in the same way.

The rest of this section deals with compiled program testing in a general sense.

It may be a good idea to use EXECUTE_A when you first run a new program, as this lets you halt it by pressing ALT SPACE - or whatever other keys you may have assigned - if it appears to get stuck in a loop or you want to terminate it prematurely for some other reason.

EXECUTE has the advantage over EXECUTE_A that you can still use the SuperBASIC interpreter while the compiled task runs. However you can get into a muddle about which task you're talking to, and it is a bit harder to kill off a task that you no longer want - you must use LIST_TASKS to find out what the QL is running, and then REMOVE_TASK to get rid of the appropriate one.

To make this identification easy, TURBO uses the first twelve characters of a compiled task name (or fewer, if there aren't that many) as the 'task name' which LIST TASKS will display.

ERRORS IN COMPILED PROGRAMS

Compiled tasks sometimes stop with an error report. As with interpreted programs, there are some errors that can only be detected when code is executed. These, and their causes and cures, are listed at the end of the next chapter.

The most common problem is a task which stops and says 'increase dataspace please'. This means that you have not given the task enough memory for its variables and other information generated as it runs. The cure is either to use DATASPACE_TASK on the compiled program or to recompile the program with a larger dataspace.

In general, you should not compile programs that you have not tested with the SuperBASIC interpreter - it is much easier to test code interactively with an interpreter, rather than try to resolve problems from a listing and run-time error reports.

Used together, the interpreter and compiler should complement one another - use the interpreter to get a program working, and then compile it to bring it up to a fast, professional standard, with support for multi-tasking and other TURBO 'goodies'.

However, you may sometimes decide to compile a program just to find out more about the errors that you know it contains. The SuperBASIC interpreter often stops dead after an error which it has failed to diagnose properly, and TURBO can be a valuable tool in correcting such problems.

TURBO TUTORIAL - THE HOME STRETCH

Use EXECUTE or EXECUTE_A to run the program you have just compiled.

You can correct dataspace problems by following the explanation in Chapter D, but you'll have to re-compile the task if you haven't given it any windows - this is because the code to open windows automatically will not be in the generated task. TURBO is very economical, and never includes code unless it is needed!

Once you've got the task running with EXECUTE, type CTRL-C, or whatever other keys you've set, to make the cursor flash in the task's window instead of the SuperBASIC one.

Any time there's a cursor displayed by a compiled task, and you haven't used EXECUTE_A, EXECUTE_W, or EXEC_W to put SuperBASIC to sleep, you can switch back to the interpreter with CTRL-C - perhaps to use the DIR command. If SuperBASIC is still awake you will see a static cursor in window zero, where you normally enter commands and program lines.

Try swapping back to BASIC and typing LIST_TASKS 0 - a list of the tasks running, including BASIC and your compiled task, should appear at the bottom of the screen.

You can swap back to the task later by pressing CTRL-C again, so long as the task has a cursor waiting for you. The cursor is the QL's way of signalling that a task is waiting for input. You can't type things into a task that is not showing a cursor - Chapter G discusses this problem, which is 'built in' to the QL's design.

The end of the beginning

You've reached the end of the tutorial - you've got some idea of the power and versatility of TURBO. Please read on, experiment, and adapt the routines we've supplied. USE this manual when you get stuck, as -inevitably - you sometimes will.

Persevere with TURBO, and you'll discover why we persevered with its development, through dark days for Sinclair and the home computer industry. It's up to you now. Have fun!

CHAPTER F - COMPILE- AND RUN-TIME MESSAGES EXPLAINED

SUMMARY OF REPORTS AND MESSAGES

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.

The messages are listed in three groups, each in alphabetical order. The first group consists of 'errors', which the compiler prefixes with '****' to make them stand out in the report file. You must correct the program, to remove the cause of the error, before the program can be compiled into a task.

The second group consists of 'warnings' which are prefixed by '++++' in compiler reports. These messages indicate that TURBO has detected a trivial mistake or a potential cause of problems, such as a missing END DEFine or END FOR, and assumed appropriate code to correct the mistake. TURBO does NOT change your program for you, but it is easy to look through the report and make appropriate changes.

TURBO can correct many 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 TURBO's corrective action.

The last group consists of 'run-time' messages - reports that may be returned by compiled programs. Unless these crop up as soon as a task is invoked they are accompanied by the name of the relevant task, and the number of the line most recently started before the error occurred. This line number is always 0 if that task was compiled without "Include Nos" selected.

This is a complete and exhaustive list of TURBO messages; some of these are most unlikely to appear in normal use.

A FUNCTION MUST RETURN A VALUE

Either you have tried to RETurn from a function without supplying a 'result' as a parameter of RETurn, or you have started a statement with the name of a function.

The second possibility indicates one of two logical errors. Either you've tried to call a function without specifying a destination for the value that will be returned, or you have tried to store a value in a variable that has the same name as a function.

AMBIGUOUS DECLARATION OF NAME

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

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.

ARRAY NAME REQUIRED

The first parameter of DIMN 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.

ARRAY OPERATION NOT ALLOWED

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, unless the statement is DIM, LOCAL, EXTERNAL, GLOBAL or REFERENCE, in which case the entire array should be specified.

Alternatively this message may indicate that you have tried to re-dimension an array which has been declared as EXTERNAL to this module. A shared array may only be re- dimensioned by the module in which it is GLOBAL. Count yourself lucky that you're allowed to do that!

BUFFER TOO SMALL

There are two possible reasons for this message.

1. You have requested the inclusion of an extension file whose length is larger than that of the TURBO BUFFER.

2. Your program contains a constant string (a text literal) which is larger than the 'buffer size' you set when you compiled the program. Since the minimum buffer size is 1024 characters, this is an extremely unlikely error! You can prevent the error by increasing the buffer size so that it exceeds the length of the longest sequence of characters enclosed by single or double quotation marks, in your program.

CALCULATION 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).

CALCULATION TOO COMPLEX

A name declared MANIFEST must br set equal to an explicit number or string.

Otherwise 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. You must simplify the expression or split it into several stages.

CAN'T PASS EXPRESSION BY REFERENCE

You have used a compound expression in calling a procedure or function whose DEFine statement is preceded by a REFERENCE statement for the corresponding parameter. Either assign the value of the expression to a variable which you then use in the routine call, or accept passing by value by deleting the offending parameter from the REFERENCE statement (or the whole REFERENCE statement if this was its only parameter).

CHANNEL NUMBER (0-31) NEEDED

You must specify a valid channel number as the first parameter of this command. TURBO supports 32 channels, numbered from 0 to 31. If this error appears by a calculation which will always have a final value between 0 and 31, put brackets around the calculation to stop TURBO complaining.

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.

CONFLICTING DIRECTIVE STATEMENTS

You have used a name in both a GLOBAL and an EXTERNAL statement of the same module.

DEFINE / DIRECTIVE MISMATCH

For a routine specification in a GLOBAL statement, there is no matching specification in a later DEFine statement, with the only difference being the omitted parentheses around any value parameter.

Another possibility is that a DEFine statement includes a name previously used in an EXTERNAL statement in the same module.

DUPLICATE GROUP NAME

A group name occurs twice in non-consecutive GLOBAL or EXTERNAL statements of the same module.

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.

ERROR DIAGNOSIS FAILED

An internal problem has occurred, 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.

FAULTY LINE REFERENCE

The lines specified in RESTORE, GOTO, GOSUB and ON..GO statements must be fixed values - not the results of a calculation. Furthermore, they must be the numbers of lines that exist. Procedure names may be used instead of line- numbers in ON..GO SUB, but the names must refer to BASIC procedures that expect no parameters, so that a 'GOSUB' is appropriate.

INCONSISTENT MODULE NUMBERS

Either two GLOBAL statements have different module numbers, or an EXTERNAL statement has the same module number as the module's GLOBAL statement(s).

INCORRECT CALCULATION SYNTAX

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.

INCORRECT LINK SPECIFICATION

A GLOBAL or EXTERNAL statement has been found in the wrong part of a program, or there is an error in the format of the declaration. GLOBAL and EXTERNAL statements must come after all the DATA in a program, and after DEFinitions and DIMension statements which refer to their parameters. The format of link declarations is discussed in Chapter R.

INCORRECT NUMBER OF PARAMETERS

A call to a SuperBASIC or machine-code routine 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 TURBO's PARSER is running. The SuperBASIC data area should not be modified while TURBO runs, or the compiler may 'lose its place' and issue this message. The problem cannot occur if you invoke TURBO with the CHARGE 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.

ONLY SCALAR NAME FOR MANIFEST

The name of a string declared MANIFEST must not have been dimensioned as a string array. (Dimensioning a name as a simple string before it is declared MANIFEST is allowed but gives rise to a Warining.)

PARSER DATASPACE FULL

TURBO has not got enough space to keep details of all the line-references, declarations and nested control constructs in your program. The simplest corrective actions are to increase the dataspace of PARSER TASK by a few 'K', as explained in Chapter E, or reduce the size of the program.

PROCEDURES DO NOT HAVE VALUES

You're either trying to use a procedure name (which has no associated value) in IMPLICIT or a calculation, or you're trying to RETurn a value when you're not in a function.

You should not use the RETURN command with a parameter (e.g. RETURN var) inside a procedure definition or at the end of a standard BASIC subroutine (the sort you'd call with GO SUB). Neither procedure-calls nor GO SUB expect you to return a value.

If this report appears by a calculation it generally means that the name has been mis-typed, or a procedure has been accidentally defined instead of a function.

ROUTINE NAMES CAN NOT BE PARAMETERS

You have accidentally put the name of a procedure or function in a REFERENCE directive. Variables of all types and shapes can be passed by REFERENCE, but not routines!

STATEMENT IS NOT YET SUPPORTED

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

This may indicate an attempt to use a 'level' of hierarchical WHEN ERROR trapping which is not currently supported. Consult Chapter S for more details of WHEN_ERROR.

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 be trying to use a variable as if it was the name of a procedure.

WARNING MESSAGES

The next group of 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.

TURBO 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 TURBO has done and why it has done it.

This TURBO facility, called 'auto-correction', is discussed and illustrated in Chapter I, which contains lots of extra information about SuperBASIC syntax and the auto-corrector.

WARNING: CLEAR DESTROYS ALL STRINGS

All strings in TURBO are arrays, even if they are 'automatically' dimensioned for you by TURBO. It follows that they are de-allocated by CLEAR.

If this warning appears on a CLEAR which is only executed right at the start of your program, your best bet is remove the CLEAR, as TURBO automatically CLEARs variables at the start of a program.

If the CLEAR is executed while a program runs you should make sure that you re-declare all the strings you will want to use later -particularly any which were automatically created at the start of the program by TURBO.

WARNING: DEBUG LEVEL NOT GIVEN - 0 ASSUMED

A DEBUG command has been issued before DEBUG_LEVEL.

WARNING: DEBUG LEVEL RESTRICTED TO 9

DEBUG_LEVEL has been given with a parameter greater than 9.

WARNING: DEBUG 0 ASSUMED

A DEBUG command has been issued without a following integer.

WARNING: DEBUG RESTRICTED TO 9

A DEBUG command has been issued with a following integer greater than 9. TURBO has assumed that DEBUG 9 was intended.

WARNING: DIM var$(100) ASSUMED

The string shown (var$) is used in your program but not dimensioned. TURBO has automatically dimensioned it to a maximum length of 100 characters.

WARNING: DIRECTIVE IN THE WRONG PLACE - IGNORED

TURBO has found a CONTINUE or RETRY statement outside a WHEN clause in a compiled program. If the interpreter found such an error it would stop abruptly, with no message! TURBO is more polite, and ignores the statement while warning you of its presence.

This warning may indicate that GLOBAL or EXTERNAL directives in your program have been ignored, because they were not at the start of the file. Move them by typing EDIT, changing the line-number, and deleting the old line. The auto-corrector indicates the relevant line-numbers in it's report. Then re-compile the program!

The rule about the placement of GLOBAL or EXTERNAL statements could be relaxed if we made one of two changes to TURBO: either we slow compilations by a factor of 25 per cent, or we double the amount of code and data (in compiled tasks, and elsewhere in the TURBO system) used to implement LINK_LOAD. We feel that either 'solution' would be counter-productive!

Finally, this warning appears if you try to use the SNOOZE or CATNAP directives in a program or module with no GLOBAL routines in it. This is pointless because such a task would do absolutely nothing (but occupy memory) thereafter until it was removed.

WARNING: DUPLICATE DEBUG LEVEL IGNORED

A Debug level may only be specified once.

WARNING: END DEFINE sub ASSUMED

TURBO has encountered a DEFine or REFERENCE 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.

The auto-corrector assumes that you meant to put an END DEFine before the next definition. This corrects the error, so long as you have not tried to nest one definition within another and there was not meant to be any code between the two definitions.

WARNING: END FOR ASSUMED

TURBO has reached the end of a block and realised that it has started a FOR loop in the meantime, but never found the appropriate END. To keep things properly nested, the auto- corrector adds an END FOR here.

WARNING: END FOR var ASSUMED

TURBO has assumed an END FOR where it is required by the SuperBASIC syntax. In the example, 'var' corresponds to the name of the variable controlling the loop.

WARNING: END IF ASSUMED

TURBO has reached the end of a block and realised that it has started an IF since then, but never found the appropriate END. To keep things properly nested, the auto- corrector has added an END IF at this point.

WARNING: END REPEAT ASSUMED

TURBO has reached the end of a block and realised that it has started a REPeat loop since then, but never found the appropriate END. To keep things properly nested, the auto- corrector adds an END REPeat here.

WARNING: END SELECT ASSUMED

TURBO has reached the end of a block and realised that it has started a SELect since then, but never found the appropriate END. To keep things properly nested, the auto- corrector adds an END SELect here. It is rather unlikely that this program would work, even under the interpreter.

WARNING: END TREATED AS NEXT

The TURBO auto-corrector has found a conditional END FOR. In correct SuperBASIC, END FOR is a 'static' marker, indicating the end of the scope of a loop. Every loop must have an END, and it is not sensible to have one that may be here one minute and gone the next!

The auto-corrector will issue a sequence of messages, documented in Chapter I, to correct the error while preserving the behaviour of the program under the SuperBASIC interpreter. TURBO changes the compiled program, but leaves your SuperBASIC alone.

WARNING: END WHEN ASSUMED

TURBO has reached the end of a block and realised that it has started a WHEN_ERROR since then, but never found the appropriate END. To keep things properly nested, the auto-corrector adds an END_WHEN here.

WARNING: EXCESS PARAMETERS IGNORED

There are more parameters in the call than in the definition. The excess parameters (from the right) are ignored.

WARNING: EXIT var ASSUMED

TURBO is in the throes of correcting a conditional END FOR or END REPeat statement in your program. 'var' is the name of the loop identifier. Please read Chapter I to find out what the auto-corrector is up to.

WARNING: EXTENSIONS ARE TOO LONG

The total length of the string which includes all the extension files exceeds 32766.

WARNING: EXTRA ELSE CLAUSE IGNORED

TURBO has found a second (or subsequent!) ELSE matching a single IF statement. The extra ELSE, and all the code after it up to the END IF, is ignored, just as it would be by the SuperBASIC interpreter. You might as well remove the code!

WARNING: EXTRA END IGNORED

TURBO has found an END when it had not previously found a matching start of block. The extra END is ignored, just as it would be by the SuperBASIC interpreter.

WARNING: EXTRA END TREATED AS RETURN

TURBO has found an END when it is not scanning a definition. There are two possible reasons why this warning might appear. Either one definition has been put inside another, which is not allowed and will cause other warnings, or one definition has more than one END. The auto-corrector has assumed the second case. Check the program to make sure that this correction is appropriate.

WARNING: FAULTY EXTENSION STRING

A REMark %%.. string designed to cause inclusion of a file of SuperBASIC extensions is faulty.

WARNING: FILE <file> CAN'T BE OPENED

The filename <file> requested as an extension can't be opened.

WARNING: FILE <file> IS TOO LONG

The filename <file> requested as an extension is longer than 32752.

WARNING: LINE NOT FOUND num ASSUMED

The line referred to in the program does not exist. Like the interpreter, TURBO has assumed you meant to refer to the next existing line number - the one whose number is shown in the warning message.

WARNING: LOCAL var$(100) ASSUMED

You have not specified an explicit maximum size for this local string, so TURBO has assumed a length of up to 100 characters.

WARNING: MEANINGLESS COMMAND IGNORED

The command indicated is incompatible with the form of a compiled program. When a TURBO task is executed the original program text is not present. Thus commands such as LIST, MERGE, LRUN and so on are meaningless - they work with SuperBASIC text, not compiled machine-code.

WARNING: MISSING PARAMETERS ASSUMED

There are more parameters in the definition than in the call. Zeroes and/or null strings have been added, as appropriate.

WARNING: NEXT TREATED AS END FOR

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.

TURBO has detected an attempt to end a loop with NEXT, rather than END FOR, as you would in 'traditional' BASIC. It assumes that you really meant END FOR, and corrects the statement for you. You should really change the source, to avoid problems when testing the program with the interpreter. For further information, refer to Chapter I.

WARNING: NUMERIC PARAMETER ASSUMED

TURBO 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 variable will have its value passed to machine-code, but subsequent changes will not affect its value.

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.

WARNING: REFERENCE TO name UNUSED

The variable name shown was amongst the latest group of REFERENCE names, yet it did not appear as a parameter in the subsequent DEFinition. Have you made a typibg mistake?

WARNING: TOO MANY EXTENSIONS

A call to include an extension file has been issued after eight have been accepted. Only eight are permitted.

WARNING: var$ ONLY EXISTS AS A LOCAL

TURBO is warning you that the variable shown is a local or parameter string, but it is never dimensioned globally. Attempts to use it outside the procedures or functions where it is local will give a NOT FOUND error.

You are advised to check the listing to make sure that you are not using this name for a global, as well as a local, string. If so, add a DIM statement so that the string exists even if the relevant procedure or function has not been called.

WARNING: VALUE PARAMETER ASSUMED

The variable shown will have its value passed to machine-code, but subsequent changes will not affect its value.

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.

Under rare circumstances the line number shown may not be the exact location of the error; sometimes TURBO merges or interleaves lines in order to obtain extra efficiency. In any case the error will be very close.

WARNING: WRONG EXTENSION FILE

A file requested as an extension file is too short to contain one or both of the initialisation routine or the definition block.

RUN-TIME ERRORS

Errors which occur while a compiled program is running are reported using the standard QDOS messages, documented in the 'Concepts' section of the QL User Guide. There are also several messages which are unique to TURBO.

Unlike some other compilers, TURBO 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.

If you have a 'JS' or later version of the QL you can change the text that TURBO, SuperBASIC and most other tasks use to report 'standard' errors.

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.

You can 'trap' any errors which may occur within a compiled program, and check the line-number and error code (shown in brackets after messages in this section). TURBO TOOLKIT also includes several functions which can be used to detect potential errors before they occur. Error trapping is discussed in Chapter S of this manual.

TURBO 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 EXECUTE_W, EXECUTE_A, or another instruction that does not allow you to type further commands until the task has finished, commands, reports and messages appear in window 0. You can conceal these, if you find them annoying, by setting the INK and PAPER in that window to the same colour. No message is printed if a task stops normally, without an error.

Run-time message format

Most TURBO reports indicate the number of the last line which was started before an error occurred. If the line number shown is zero this usually means that the compiled program stopped before the first line was reached - perhaps while setting up channels or variables - or you have compiled your program with "Display Nos" or "Omit Nos" selected.

You are advised not to select "Include Nos" until a program is fully tested.

For example, you might see:

**** TASK test HALTED, AFTER LINE 200. REASON: END OF FILE

Under rare circumstances the line number shown may not be the exact location of the error; sometimes TURBO 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 indicated.

TURBO's own messages and their meanings:

(1) An explicit instruction is printed if a task fails because it has not been allocated enough space for data:

**** TASK x HALTED, AFTER LINE 0, INCREASE DATASPACE PLEASE

The procedure to change the amount of dataspace used by a compiled task is discussed in Chapter E. You may prefer to alter the program to make it less greedy. This condition can be trapped with WHEN ERROR. Its code number is -3.

(2) If you LINK_LOAD modules which contain EXTERNAL and GLOBAL declarations that do not match you will see a report similar to this:

**** MISMATCH BETWEEN MODULES 1 & 2 AT ITEM 7

This indicates that the declarations in modules 1 and 2 do not match - there is some important difference in the way that the seventh item is declared. Perhaps the types of parameters differ, or the number of dimensions of an array is not consistent. You must correct and re-compile the offending module.

(3) You may use most SuperBASIC 'extensions' in compiled programs (see Chapter Q) but these must either be loaded when you EXEC the task or you must have requested the extensions file to be included when the program was compiled. (In any event the extensions must be loaded when the task is compiled.) If any such procedures or functions are not found when a task is started TURBO 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.

(4) If a processor-detected hardware error occurs while a TURBO task is in control of the system (perhaps because of a misdirected POKE$ or overheating problem) the task may detect this condition automatically using a facility that the 68008 can provide, but few programmers use. Most tasks 'freeze' or crash the system after such errors - but TURBO tasks can cope with them, as long as they are detected by the processor or QL support chips.

If such a hardware error is detected, TURBO stops the task, attempts to empty buffers, close all channels, and de-allocate memory, and prints this message:

TASK x HALTED, AFTER LINE 210, HARDWARE EXCEPTION 4

Collectors of random car-numbers will be thrilled to learn that the exact numbers at the end of the line correspond to the following system 'exceptions':

1. Address error - an attempt to access memory in a way that is not allowed.

2. Memory error - the task or TURBO CODEGEN file may be corrupt, or the program in memory may have been POKEd damagingly.

3. Unused software exception. See 2.

4. Level 7 interrupt - an attempt to access Sinclair's in-house hardware de-bugger! This unfortunately closes down the second processor. TURBO should be able to close and preserve file data, but you'll have to press Reset afterwards.

Standard QL messages and their meanings

If TURBO gives a 'reason' for stopping this will be a standard QDOS error message. In general this corresponds exactly to the message which the SuperBASIC interpreter would give in similar circumstances. The following list is not exhaustive, as extensions can return any error code for any reason, but it should be useful nonetheless:

ALREADY EXISTS (-8)

In general, this means that a file which was not supposed to exist has been found.

If this message appears when you try to invoke several modules with LINK_LOAD it means that two or more of the files have the same module number. You must ensure that every module has an unique number, or other modules will not be able to tell the duplicates apart.

BAD OR CHANGED MEDIUM (-16)

There's something wrong with the disk or cartridge the compiled program is using, or there has been an attempt to record on a write-protected microdrive cartridge, or the medium in the drive has been changed while the system was updating or writing to a file.

BAD PARAMETER (-15)

This has its usual meaning - TURBO performs simple parameter checking when a program is compiled, but some mistakes can only be detected when a program is executed. Bad Parameter can be returned by any resident procedure or function, if you supply too few, too many or inappropriate parameters.

TURBO itself generates this error if you try to PEEK or POKE a word or long word at an odd address, or if you supply a negative length to FILL$.

This message also crops up if you try to load an inappropriate task with LINK_LOAD, or if you try to EXEC a TURBO task that has been corrupted.

BUFFER FULL (-5)

A line of input is too long for the area of memory that has been set aside to contain it. This may be because the end of line marker -CHR$(10) - is missing!

TURBO uses ALL the free memory within a task, up to a maximum of 32,767 bytes, as a buffer for INPUT, so you can often avoid buffer full errors by increasing the dataspace of a task.

CHANNEL NOT OPEN (-6)

This message appears if you try to use a channel that is not open (surprise, surprise), or if you specify a channel number less than zero or greater than 31, or if you try to connect more than 32 pipes to a task. The last problem should never crop up if you load tasks with EXEC, LINK_LOAD or EXECUTE.

DRIVE FULL (-11)

There's no room for any more data on a disk or microdrive cartridge. The file is automatically closed by TURBO when your task stops.

END OF FILE (-10)

The program has tried to read from a pipe that has been closed and drained of data, or there's no more data in a file or available from a device, or you've tried to READ when there are are no more DATA items in the compiled task. You can detect this situation before an error occurs with the EOF function.

ERROR IN EXPRESSION (-17)

TURBO has been unable to convert a non-numeric string into a number. To see this error, write: X% = "NINE" ! To avoid this error in INPUT statements, use EDIT%/EDITF instead.

FORMAT FAILED (-14)

There's no disk or tape in a drive, or the disk, tape, drive or interface is faulty or was removed during formatting.

IN USE (-9)

The file or device is already being used, and no other task may use it until the one that got there first has finished.

INVALID JOB (-2)

The program has referred to a task that doesn't exist.

NOT FOUND (-7)

This generally indicates that a compiled program has tried to access an array - or a string - which does not exist, either because it is not LOCAL to the line shown, or has not been dimensioned, or has been destroyed by CLEAR. If the line shown uses strings, and your program uses CLEAR, you may find it helpful to re-compile the program with the 'REPORT $' option selected.

In file-handling commands, this error indicates that the QL cannot find a file or device which your program expects to exist. Use DEVICE_STATUS to trap this error.

This message also appears if your program tries to RETurn when it is not performing a subroutine, function or procedure-call, or if you try to RETRY without having specified a return-point with RETRY_HERE.

If this message appears as soon as you try to LINK_LOAD a group of files it indicates that a module declared as EXTERNAL in one or more of the files is missing. Check that you have supplied all the required module-names. If this doesn't cure the problem, check the EXTERNAL directives in the programs, and make sure that corresponding GLOBAL declarations exist.

OUT OF RANGE (-4)

This indicates that your program has tried to refer to a non-existent array element or slice, or used an invalid colour-number or a graphics co-ordinate outside the relevant window. It also appears if the value used in ON..GO TO or ON..GO SUB is less than one, or more than the number of possible destinations.

OVERFLOW (-18)

This message is generated if values go beyond the allowed range; for instance, if floating-point values exceed about 1.6 E 616 (which, to be fair, is an extremely big number!). There is no check for floating -point 'underflow' - ever so extremely tiny numbers are treated as zero. Division by zero is a favourite way of generating this error.

Overflow is reported if integers - including temporary results obtained in the course of a calculation - become less than -32768 or more than 32767. For more information about floating-point and integer values please consult Chapters L and M.

An overflow is also indicated if you try to concatenate two strings with a total length of more than 32767 characters.

READ ONLY (-20)

A file exists and may be read but not written, because the disk is write-protected or another program is already reading the file. Unfortunately the microdrive software in the QL does NOT issue this report if a cartridge is 'write protected' - it just gives a BAD OR CHANGED MEDIUM later when it gets bored of trying to write to the cartridge.

XMIT ERROR (-13)

You've set the wrong 'parity' on something connected to one of the QL's 'SER' ports.

CHAPTER G - REAL-WORLD PROBLEMS AND SOLUTIONS

THE TURBO ADVICE CENTRE

We have tried to eliminate all of the problems that we know about in SuperBASIC by implementing TURBO, so information for this section is a bit short at the time of writing. Doubtless this situation will change!

OUT OF MEMORY - IMMEDIATELY AFTER CHARGE

If your QL reports OUT OF MEMORY when you type CHARGE, this means that there is not enough memory for TURBO and your program source to co-exist. It is worth typing CHARGE once again, as some QL facilities don't actually release memory that they've locked away until an 'out of memory' error occurs.

If this doesn't help, here are four possible solutions - in ascending order of generality - all fairly quick and easy.

The first is also the simplest, but it costs money. Get some more memory for your QL. This will speed your machine up and - in all probability - cure 'out of memory' problems for good!

The second solution is to try to 'compress' the program source. This is a bit of a 'fudge', but it can sometimes be worthwhile. You can compress a program in several ways... In order of increasing hassle, these are:

1. Typing CLEAR and trying again;

2. Re-booting the computer, loading ONLY the extensions that your program and TURBO need (e.g. TURBO TOOLKIT), re-loading the BASIC program, and trying again;

3. Editing the file (e.g. with THE EDITOR) to remove statements that are not needed (e.g. REMarks), then following the advice of (2).

It DOES NOT help to renumber the file with shorter line numbers, and you save very few bytes by compressing the names of variables and routines, because these are only stored once, however many times they occur in the program.

The third approach is to split the program into distinct tasks which communicate via EXECUTE 'pipes' and - perhaps - 'option strings'. This is an efficient solution if parts of your program just 'process' a stream of data in a standard way.

Such tasks are often called 'filters', and can be quite complex, while still preserving a very simple interface with other programs. This approach, documented in Chapter R, is sometimes very elegant, but it is not as general as the last solution...

The fourth, and generally the best, approach is to split the file into modules. Chapter R also contains details of this technique, which makes program editing, saving and re-compilation faster and easier than before. It is especially powerful when working with large or complicated programs.

You can divide a program 'by hand' with LIST or SAVE, adding GLOBAL and EXTERNAL statements as required, or use the program 'MAKE_MODULESS', on your TURBO cartridge, to split the file into modules and generate the GLOBAL and EXTERNAL declarations for procedures and functions, without your having to lift a finger!

MAKE_MODULES splits a composite file as evenly as possible into several parts that can be edited and compiled separately. You must tell it the name of the input file and the number of parts you want - the program will generate the modules automatically, making up their names from a 'stem' you provide.

Line numbers up to 99 must be free for the GLOBAL and EXTERNAL declarations. You've still got to add declarations for any 'shared GLOBAL' variables - ones that are used in more than one part of the file, but not passed as parameters.

List all the names of shared variables in GLOBAL statements at the start of the module containing the DIMs and initial procedure calls.

The same lists, but starting with EXTERNAL, should appear at the start of the other modules, to show that they know who's boss. These subservient modules should contain a SNOOZE command (generally at the start) so that they don't 'finish' and unload as soon as you've LINK LOADed them!

The whole 'translation' process - needed if you choose to run large programs on a small QL - should take no more than a few minutes, especially if you use a listing, or the TURBO TOOLKIT BASIC_NAME$ function, to remind yourself of the names of routines and variables. As a 'side-effect' your program will be better organised and easier to test.

Pidgin BASIC and Pseudo-Fortran users...

There's one small snag for people who ignore most of the facilities of SuperBASIC and fill their programs with line-numbers rather than sensible names. You can only use modules if your program is made up of procedures and/or functions - not if it is just one monolithic wodge of lines and GO TOs. Luckily, few programs which are too big to compile in one go are written with no 'structure' at all!

If you have only used GO SUBs, and no procedure-calls, you can allow modularisation by changing the GO SUBs into parameterless procedure-calls. This is a trivial job which makes the program much easier to read and does not affect its efficiency when compiled.

Even if the program you've been given is totally formless you can always split it arbitrarily into named procedures which are called in sequence - the trick, in this case, is to minimise the need for variables to be 'passed' from one module to the next. All such variables must be declared as GLOBALs (in the first, or dimensioning one) and EXTERNALs in the others.

THE "MG" ROM

Some problems crop up when graphics commands are used in SuperBASIC programs run on the "MG" version of the QL. These problems stem

from a mistake in that version of the QL's ROM graphics routines, so they also affect programs compiled with TURBO. Tony Tebby's 'patch' to correct these problems fixes both compiled and interpreted programs.

PAUSE AND INKEY$ IN COMPILED CODE

These commands necessarily assume that the task they appear in has total control over the QL keyboard. This can cause problems if the commands appear in multi-tasking code, and you don't want to use EXECUTE_A or EXECUTE_W.

If you start a task with EXECUTE or EXEC and do not 'select its window', INKEY$ and PAUSE will not work until the appropriate window is selected. You select a window as explained in Chapter D, by the typing of CTRL C or some equivalent that you may have set yourself. The QL does not allow this until a cursor in that window is enabled.

You can get around this by starting the task with CURSOR_ON or an INPUT statement. The window concerned will be selected, and will remain selected until you deliberately switch to another. Alternatively you could replace INKEY$ statements with KEYROW calls, although the second processor is rather feeble and quickly gets swamped if several tasks try to call it many times a second.

You can read the CODE of the last key MUCH more quickly from a multi-tasking program with PEEK(SYS_VARS+139). If you get 255 this means ALT with another key - you can find out the other one, in such a case, with PEEK(SYS_VARS+138).

(Note that on the Q40 and Q60 the 255 is found at SYS_VARS+138 and the other key from SYS_VARS+139.)

You'll find a fascinating example of a multi-tasking utility that reads - and writes! - characters in this way in the TURBO TK DEMOS file, under the name UNSHIFT.

Back with PAUSE and INKEY$, you may find that you need to compile the tasks with at least two default windows (0 and 1) if you're using these statements in the program. The commands use channel 0 when working out delays!

A more efficient way to generate a fixed time delay in a compiled program, if you're not bothered about key-presses, is to use the TURBO TOOLKIT command SUSPEND_TASK.

Finally, you may be able to get EXECUTE to behave enough like EXECUTE_A to 'fool' the operating system into thinking that a given task is the only one that could want keyboard characters.

On most QLs you can get the effect of EXECUTE_W without losing the option of using the interpreter later. Instead of EXECUTE_W, type a command of this form:

```
EXECUTE  FILENAME: SNOOZE
```

This sends SuperBASIC to sleep, allowing keys to get through to the loaded task. You can re-start the interpreter at any time by pressing CONTROL SPACE, to break the effect of SNOOZE.

This only works from SuperBASIC - not from within a task - and may not work at all on early ROMs that leave SuperBASIC selected even when it is asleep!

PARSER TASK - PLEASE INCREASE DATASPACE

If the TURBO PARSER halts with an error message, saying 'please increase dataspace', you can increase the allocation as if TURBO were any other task. This problem should only crop up rarely unless you are compiling large programs or modules, on a system with extra RAM.

Use fairly small increments - between 2K and 10K extra each time, depending upon the amount of memory you have 'free'. TURBO is not very greedy.

FURTHER HELP

If a problem leaves you completely stumped, read about any error, warning, auto-corrector message or 'reason' which TURBO may have issued. Then consult the appropriate parts of Section 3, which deal with arithmetic, variables, parameters and extensions, and list interpreter bugs that TURBO exterminates.

Chapter H-for-help should be passed by, for now. It is Your Last Resort. It explains what to do if your program runs (fully and properly) under the interpreter, matches the specifications in this TURBO Encyclopedia, but STILL won't compile, or 'falls over' when it runs as a task. Help is probably literally at hand, in Sections 3 or 4 of this Manual. Use the Contents list to track down your problem's cause and cure.

CHAPTER H - HELP! WHAT TO DO WHEN THINGS GO WRONG

WHEN ALL ELSE FAILS - READ THE MANUAL

we have had a lot of experience helping people to use compilers -especially SUPERCHARGE, which was the prototype for TURBO. We have tried very hard to build that expertise into this manual. We've tried to tell you all you might want to know about TURBO.

UNDERSTAND THE PROBLEM

Try to narrow down problems. We're not asking you to do work on our behalf - we're trying to help you to resolve difficulties without having to wait for us. Surprisingly often, problems go away when you try to pin them down.

Cut problem parts out of a program and see if they still fail. If the problem vanishes your diagnosis was wrong, and you may not need our help at all.

Unplug all possible peripherals on a machine (leaving expansion memory, perhaps, if you need it to compile or run a program) and see if that helps. We can't fix your disk drives for you, for example, however much we fiddle with TURBO!

Use the TURBO options - you may have accidentally mis-directed the compiler. If a problem occurs when you optimise a program for SPEED, does it also occur when you optimise for SIZE, or turn optimisation OFF? This can help you to see where you've gone wrong.

SECTION 3 - FUNDAMENTAL CONCEPTS

8th June 2006

CHAPTER I - SYNTAX AND TURBO'S AUTO-CORRECTOR

WHY AN AUTO-CORRECTOR?

TURBO is developed from SUPERCHARGE. When SUPERCHARGE was launched we discovered that relatively few people programming the QL were familiar with the precise syntax of the language.

This was not helped by the behaviour of the SuperBASIC interpreter, which has a habit of ignoring incorrect syntax elements, and stopping suddenly, with no error message, when it found something else it didn't like or didn't find something it wanted. The lack of a message means it's still a bit hard to tell what's wrong.

The interpreter seemed to have started out as the new, wonderful design that Sinclair always claimed it to be; it then underwent a rapid succession of changes in an attempt to make it more 'familiar' - in the case of SuperBASIC, more like Microsoft BASIC and Sinclair's earlier ZX BASIC. These changes didn't 'fit' the original design very well, and some nice features had to be left out to make room for them.

Even as this work was being done Uncle Clive was launching the machine to the Press, in a bid to pre-empt Apple's official launch of their Macintosh, which would otherwise become the first mass-market micro based on the Motorola 68K family. In the short term this worked much too well - the Press were willing to believe anything that the father of the Spectrum and ZX-81 said. Sinclair Research became committed to shipping the machine long before it was finished.

Under the circumstances it's no surprise that the language has a few rough edges. It was shoe-horned into a 32K ROM, with QDOS, but Sinclair kept adding features and it soon outgrew the available space. The development continued as machines were shipped, and two inevitable problems resulted:

1. We ended up with a BASIC that worked perfectly on 'correct' (test) programs, but behaved rather strangely in other cases.

2. The supplied documentation did not keep up with the software - it was chaotic and often inaccurate or incomplete.

By the time Jan Jones, original author of the interpreter, had escaped from Sinclair and written 'QL SuperBASIC, the Definitive handbook' (published by McGraw Hill UK) most people had made up their own 'rules' for QL programming.

With the exception of some ambiguity in the area of nested short-forms (which sometimes worked under the interpreter, sometimes not) SUPERCHARGE was a full implementation of the syntax of SuperBASIC, although it had some semantic differences. But a lot of people were programming the QL in ways that the interpreter was not originally intended to work, and there were lots of 'quirks' where incorrect syntax, according to the documentation, could give useful results.

Some SUPERCHARGE users found that they had to make many changes to their programs to get them to compile. Others experienced very few problems. TURBO is aimed at the whole community of QL users, and it draws from the experience of SUPERCHARGE. The aim is to compile programs as they are, rather than as they 'should be'.

TURBO tackles this problem in two ways. It contains a lot of sophisticated code, collectively called the 'auto- corrector', which can work out how the SuperBASIC interpreter would treat common errors -in particular those errors which the interpreter ignores or mis-diagnoses - and correct them.

The next part of this chapter contains a list of 'auto-corrector' messages, organised by their meaning, with full examples rather than as summarised for reference in Chapter F.

The second specification problem is addressed differently. TURBO is supplied with a precise statement of the syntax of the language, in the form of diagrams which are easy to follow and will already be familiar in format to many programmers. These 'syntax diagrams' are introduced at the end of this chapter.

THE AUTO-CORRECTOR

TURBO's auto-corrector detects common, non-fatal errors in a SuperBASIC program, and corrects them automatically. The auto-corrector also detects some small differences between the interpreted language and TURBO's fast compiled dialect, and brings these to the notice of the user.

Messages keep you informed of what's going on. These are not produced to show how clever TURBO is, or how hard it is working. It is important that you check the program to make sure that you understand what TURBO has done and why it has done it.

The rest of this section discusses warning messages in a succession of logical groups.

TURBO QUIRK WARNINGS

(1) CLEAR DESTROYS ALL STRINGS. This error message is derived from SUPERCHARGE. It was added because some users became confused by the fact that CLEAR, in a compiled program, deleted the memory reserved for all strings, leading to NOT FOUND errors if the program then tried to use the variables without re-declaring them first.

All strings in TURBO are arrays, even if they are 'automatically' dimensioned for you by TURBO. It follows that they are de-allocated by CLEAR.

If this warning appears on a CLEAR which is only executed right at the start of your program, your best bet is to remove the offending statement - TURBO does an automatic CLEAR at the start of the program.

If the CLEAR is executed while a program runs you should make sure that you re-declare all the strings you will want to use later -particularly any which were automatically created at the start of the program by TURBO. These 'automatic' declarations are the subject of the next warning in this list.

(2) DIM var$(100) ASSUMED. The string shown (var$) is used in your program but not dimensioned. TURBO has automatically dimensioned it to

a maximum length of 100 characters, whereas the SuperBASIC interpreter would find space for it on an 'ad hoc' basis - a slow and error-prone process.

TURBO creates 100-character strings by default, rather than the 256-character strings generated by SUPERCHARGE. If this length doesn't suit you, you can add an explicit DIM at the start of your program, e.g:

```
10 DIM var$(40)
```

This statement would prevent generation of the warning, and indicates that the system need only reserve 40 characters for possible contents of VAR$.

A similar message can appear if you do not specify the length of a string declared in a LOCAL statement:

(3) LOCAL var$(100) ASSUMED. TURBO has assumed a maximum length of 100 characters on your behalf. If this doesn't suit you, you can prevent the warning and indicate any other maximum length by adding the number in brackets after the LOCAL declaration, e.g:

LOCAL bita$ might become: LOCAL bita$(200)

(4) var$ ONLY EXISTS AS A LOCAL. You have wisely compiled your program with the 'Report $' control set, and TURBO is warning you that the variable shown is a local or parameter string, but it is never dimensioned globally. Attempts to use it outside the procedures or functions where it is local will give a NOT FOUND error.

You should check the listing to make sure that you are not using this name for a global, as well as a local, string. If so, add a DIM statement so that the string exists even if the relevant procedure or function has not been called.

TURBO can't check this for you, because there's no rule stopping you having GO TOs in and out of definitions, and therefore the compiler can never be sure what variables exist in any context, unless it actually runs the program through every possible path and circumstance. No compiler in the world can check such general cases. Sorry - but at least we 'warned' you!

(5) MEANINGLESS COMMAND IGNORED. The command indicated is incompatible with the form of a compiled program. When a TURBO task is executed the original program text is not present. Thus commands such as LIST, MERGE, LRUN and so on are meaningless - they work with SuperBASIC text, not compiled machine-code.

One task can invoke another with EXEC or EXECUTE; use COMMAND_LINE and TYPE_IN, as documented in the TURBO TOOLKIT manual, if you want the first task to make way for the second.

If you really want to manipulate the stored SuperBASIC program with LIST, RENUM or some similar command - perhaps in a compiled 'helping hand' utility for BASIC programmers - you can use TYPE_IN to enter commands as if they had been typed at the keyboard.

MISSING 'END' WARNINGS

These four warnings all indicate that TURBO has reached the end of a properly-declared 'block' - a DEFine, IF, FOR, REPeat or SELect clause - and realised that it has started another block in the meantime, but never found the appropriate END. To keep things properly nested, the auto-corrector adds the missing END before the 'outer' one.

(1) END FOR ASSUMED. Sometimes you get the variable name with this message.

(2) END IF ASSUMED.

(3) END REPEAT ASSUMED.

(4) END SELECT ASSUMED. TURBO has reached the end of a block and realised that it has started a SELect since then, but never found the appropriate END. To keep things properly nested, the auto-corrector adds an END SELect here. In fact, in the particular case of SELect, it is rather unlikely that the program would work, even under the interpreter.

CONDITIONAL STRUCTURE WARNINGS

(1) END TREATED AS NEXT. The TURBO auto-corrector has found a conditional END FOR (or REPeat). In correct SuperBASIC, END FOR is a 'static' marker, indicating the end of the scope of a loop. Every loop must have an END, and it is not sensible to have one that may be here one minute and gone the next!

The auto-corrector realises that what the programmer intended was a conditional termination or iteration of the loop - depending upon the current value of the count and the remaining values, if any. The resultant correction is rather complicated, but it has the virtue of leaving a program which is 'properly' nested AND works just like the interpreter.

First the auto-corrector re-codes the END FOR as a NEXT. NEXT can be conditional whereas END FOR cannot. Appropriate other ENDs are generated to 'close' the condition:

(2) END IF ASSUMED. TURBO may generate several of these, perhaps interspersed with END SELect ASSUMED warnings, if your END FOR is VERY conditional!

Then the auto-corrector generates an EXIT, to cope with the case when the conditional test fails and the program 'falls out' of the loop, without ever reaching the END, under the interpreter:

(3) EXIT var ASSUMED. 'var' is the name of the FOR variable.

Finally the END itself is generated, no longer conditional.

(4) END FOR ASSUMED.

That's a fairly tricky 'auto-correction' (it took us a while to work out, anyway!). Here's an example, showing first some code which the program might try to correct, and then the 'auto-corrected' equivalent.

```
10 FOR ACTOR = 1 TO 11
20   TAKE_BOW (ACTOR)
30   IF APPLAUSE THEN
40     END FOR ACTOR
50   END IF
```

```
10 FOR ACTOR = 1 TO 11
20   TAKE_BOW (ACTOR)
30   IF APPLAUSE THEN
40     NEXT ACTOR
50   END IF
60   EXIT ACTOR
70 END FOR ACTOR
```

Notice how the second, correct, example is tidily indented, while the first ends 'in the air'. This is a classic sign of incorrect nesting. Of course, TURBO just prints warnings showing you how it has interpreted the program. It leaves changing the program to you - after all, it's your program!

The same technique is used to correct conditional END REPeats. We leave the nitty-gritty to your imagination.

UNEXPECTED CODE WARNINGS

Sometimes TURBO comes across things that don't make sense, but which the interpreter would ignore. The auto-corrector issues a plaintive warning, and does the same.

(1) EXTRA ELSE CLAUSE IGNORED. TURBO has found a second (or subsequent!) ELSE matching a single IF statement. The extra ELSE, and all the code after it up to the END IF, is ignored. You might as well remove the code, as it will never be executed.

(2) EXTRA END IGNORED. TURBO has found an END when it had not previously found a matching start of block. The extra END is ignored, just as it would be by the SuperBASIC interpreter.

MUTANT DEFINITION WARNINGS

One procedure or function definition may call another, or vice versa, without restriction when a program runs, but it is not correct to write one definition in the middle of another, in a program. 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".

This is the sort of program that causes problems:

```
10 DEFine PROCedure THING
20 PRINT "Start of thing."
30 DEFine PROCedure WIDGET
40 END DEFine
50 PRINT "Rest of thing."
60 END DEFine
```

The interpreter won't stop you writing things like this, and they may run as you expect, but problems are likely to crop up. If you put these definitions at the start of a program and type RUN, the second half of 'THING' will be executed and the program will stop with a NOT FOUND message. It will not skip the definitions, as it should, and continue thereafter. The DEFinitions are said to be 'nested', and this upsets both the interpreter and TURBO.

(1) END DEFINE sub ASSUMED. If TURBO comes across a DEFine while it is still processing a previous definition, the exact error is not clear. The previous definition might terminate happily, when it is called, with one or more RETurn statements. Alternatively - and more catastophically - the second definition may be nested inside the first.

When the auto-corrector finds such a case it prints the name of the definition being read in the message ('sub' in the above example). TURBO assumes that you've forgotten the END DEFine, and puts it in for you. This might not be the correct action, but it is often good enough to leave you with a program that works properly when compiled. You should check such warnings when they appear, and make appropriate changes to your program.

The warning message shows where the END DEFINE should have appeared, so it is easy to correct the error if TURBO's diagnosis is correct.

You may have missed out the END DEFine because your routine always ends with a RETurn. The auto-corrector inserts an END DEFine before the next definition, and this corrects the error, so long as there was not meant to be any code between the two definitions.

If you tried to nest one definition within another TURBO issues a further warning message when the end of the 'outer' definition is found. It is unlikely that the program will work properly. You must move the second definition so that it is not inside the first.

(2) EXTRA END TREATED AS RETURN. If TURBO finds an END when it is not scanning a procedure or function, there are two possible causes. Either one definition has been put inside another (as above) which is not allowed and will cause other warnings, or one definition has more than one END. This warning assumes the second case, as in this example which the interpreter runs without complaint:

```
10 DEFine PROCedure TEST
20 IF ready THEN GO TO 50
30 PRINT "Wait a bit, please"
40 END DEFine
50 PRINT "I'm ready now."
```

```
60 END DEFine
```

The best way to correct this is to change the first END DEFine into a RETurn, but it is rather difficult for the auto-corrector to do this: it has already checked that line, which seemed - in the light of the previous code - to be perfectly correct. TURBO's corrective action is to replace the second END DEFine with a RETurn, which does make the program work the same way whether it is compiled or interpreted.

UNCONDITIONAL 'NEXT' WARNING

NEXT TREATED AS END FOR. 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!

If TURBO finds an unconditional NEXT statement it assumes that you really meant END FOR, and corrects the statement for you. You should really change the source, to avoid problems when testing the program with the interpreter.

If you want proof that NEXT does not always terminate a QL FOR loop (and some people still don't believe this!), run this program under the interpreter:

```
10 FOR I=2 TO 1
20   PRINT "This never gets printed."
30 NEXT I
40 PRINT "Nor does this."
```

The interpreter discovers that the loop is to be skipped, since 2 is 'beyond' 1, and rushes off through the program looking for an END FOR. It doesn't find one, so it just stops - with no error message, and no indication of the fault!

This is not the only case in which the interpreter 'drops out' if it cannot find an END. These errors can be very hard to detect, especially in large programs, as they give no message and no indication of the location of the problem. If you test a program and find this fault, compile it with TURBO and take note of the auto-corrector messages -they should make the cause of the problem clear.

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

CHAPTER J - INTERPRETED AND COMPILED SUPERBASIC

COMPATIBILITY

Insofar as is sensible, TURBO is exactly compatible with the interpreter - programs written to run under the interpreter perform in just the same way when compiled.

We have taken careful note of comments from hundreds of SUPERCHARGE users when designing TURBO, and chosen a 'dialect' of SuperBASIC that is fast yet highly compatible with the interpreter. If you were one of those respondents, thank you for your suggestions.

In every case the reasons for the difference are given, along with suggested actions to be taken if the difference affects your program.

This chapter discusses the following areas of difference, and ends with a list of SuperBASIC interpreter 'bugs' that TURBO has squashed.

(a) Nesting of structures.

(b) Computed line references.

(c) Program editing.

(d) Calculations in DATA.

(e) Channel numbers.

Syntax, parameter passing, arithmetic, string and array handling are discussed elsewhere, as shown in the contents list.

(a) Nesting of structures.

This section is a concise statement of the rules.

(1) Procedure and Function definitions must not be 'nested' in compiled programs. In other words, the end of one definition must preceed the start of any other, as you read through the program from low to higher line-numbers.

This rule helps to simplify the compiler, without imposing any restrictions on the power of compiled programs.

(2) 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. The first is properly nested - the second is not:

```
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. But even if the GO TOs were present, the second example would not be allowed by TURBO.

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 TURBO much more complex and extremely slow, if not completely unworkable; consequently program lines are only considered in the order in which they occur in the listing.

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; these must be enclosed by the structure to which they refer.

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 term-inators. The bugs are not present in compiled programs.

(b) 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 generally 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. TURBO offers one

enhancement over SuperBASIC and SUPERCHARGE in this respect - you may specify the name of a procedure that expects no parameters instead of a line number in ON..GOSUB. This lets you write neatly structured, readable programs while still using the extremely efficient ON..GOSUB command.

Another way of re-coding a jump to a calculated line-number is to 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.

(c) 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. You can use them from a task to operate on a SuperBASIC program which is loaded but not running by entering them as if from the keyboard with the TURBO TOOLKIT facilities COMMAND_LINE and TYPE_IN.

The commands CONTINUE and RETRY are recognised by TURBO, but they do not have the meaning you would ascribe to them while interactively debugging a program. You can't change a task while it is being executed! CONTINUE and RETRY are used in error-trapping and recovery, as documented in Chapter S of this Manual.

(d) Calculations in DATA

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

TURBO does not allow expressions in DATA statements, as their processing would complicate the generation and execution of compiled code, and greatly reduce the compactness of DATA. TURBO can pack numeric DATA up to four times more densely than the interpreter, and sometimes twice as densely as SUPERCHARGE could.

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 complicated calculations in DATA.

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

(e) 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 TURBO allocates a fixed space for 32 channels, numbered from 0 to 31. This should be ample for almost all purposes - remember that channel numbers may be re-used once previously associated files have been closed.

'INTERPRETER BUGS' FIXED BY TURBO

Although SuperBASIC is, by and large, 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 TURBO, 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 TURBO:

(1) Interpreted BASIC can crash if more than nine parameters or local variables are used in a single procedure or function. TURBO 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 in Chapter L indicates the steps you should take to avoid rounding errors, which affect TURBO 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 within the task.

(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 TURBO.

(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 the 'heap' outside the task, rather than the resident procedure area, which cannot expand while the 'adjacent' transient program area is in use. Allocations are only rounded up to the nearest 8 bytes by TURBO, whereas the interpreter always allocates space in wasteful 512 byte chunks.

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) It is impossible to release memory allocated by the interpreter with RESPR. This is just as well, if add-on commands have been loaded into the space, but it is annoying if memory is temporarily needed for some other purpose. The cure is to use the toolkit's ALLOCATION function which lets you de-allocate space - logically - with DEALLOCATE.

Memory allocated with RESPR by a TURBO task is de-allocated automatically when the task stops, by default, but you can make the allocation persist by using ALLOCATION(0,0,BYTES), where BYTES is the amount of memory you want to reserve. Such space can still be DEALLOCATEd explicitly if you wish.

You should take note of the advice in the TURBO TOOLKIT manual if your programs make heavy use of RESPR or ALLOCATION.

(9) 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.

(10) 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. TURBO TOOLKIT corrects this bug so that CALL works correctly on any version of the QL, in interpreted or compiled programs.

(11) If you pass a slice of a string array to any procedure or function (including PRINT) under the interpreter, some memory will be 'lost' to the system until the next CLEAR or NEW. TURBO doesn't do this.

(12) If you try to READ or INPUT information into a slice of a string under the interpreter, your program will stop, usually without issuing any error message. TURBO doesn't do this.

(13) If you accidentally use the READ command without any parameters the interpreter may crash. TURBO detects such a mistake and refuses to compile the program, issuing an appropriate report.

(14) SuperBASIC thinks that -32768 DIV -1 is -32768. TURBO recognises that the correct result is 32768, which is not an allowable integer value, and reports accordingly. The interpreter has similar problems with the MOD operator; again, TURBO works properly.

(15) The interpreter may crash if you try to use END REPeat to jump out of a procedure or function. TURBO recognises that such a program is not properly nested, as discussed earlier in this chapter, and reports the error.

(16) Integer FOR loops (e.g. FOR I%=1 TO 2) are not allowed by the interpreter. TURBO can generate code for integer FOR loops; unfortunately current versions of the SuperBASIC editor do not allow such lines to be entered, so the IMPLICIT% directive must be used to take advantage of this feature and such programs cannot be tested under the interpreter.

If interactive testing is vital, REPeat loops, with explicit counting, should be used when fast integer loops are needed. Alternatively you can test your program using floating-point loop variables, and add IMPLICIT% declarations once everything appears to work.

(17) 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 not present in TURBO, 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.

(18) You can't break into a single-line recursive definition once it starts to run under the interpreter. It is easy to stop the equivalent loop in a compiled task, as long as you didn't use EXECUTE W, in which case you'll have to wait for the dataspace to fill up, causing the task to stop. Press the 'abort' keys if you invoked the task with EXECUTE A, or use REMOVE TASK after EXECUTE. To be fair to Sinclair, this is hardly a major bug!

CHAPTER K - NAMES, VARIABLES AND ARRAYS

SUPERBASIC IDENTIFIERS

Identifiers - often known as 'names' - 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.

In interpreted SuperBASIC, you may use the same identifer for several different purposes at various points in your program. For instance, you might 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% and VECTOR$

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

TURBO datatypes

Variable values may have three types in SuperBASIC: Integer, Floating-point or String. By default, TURBO 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).

You can over-ride the explicit type of a variable, in a compiled task, by including the new directives:

IMPLICIT% and IMPLICIT$

in your program. These commands should be followed by a list of names, which will be treated as integer or string types, respectively. The last character of the name is irrelevant if you use IMPLICIT to force a certain type.

This can make programs more readable, especially if you're used to 'traditional' compiling languages like C and Pascal, where the type of variables must be explicitly declared. It also helps to get around some restrictions imposed by the SuperBASIC line-entry routine, as we shall explain shortly.

The following lines would indicate that NAME and ADDRESS are string types, whereas SPEED and COLOUR are integers:

```
IMPLICIT$  NAME, ADDRESS
IMPLICIT%  SPEED, COLOUR
```

There's no limit on the number of IMPLICIT statements in a program; if you include more than one (which is a silly thing to do!) the second one is considered to over-ride the first. IMPLICIT statements should come before the declaration of arrays, functions or parameters which use the corresponding names, or an AMBIGUOUS NAME error will occur.

The SuperBASIC interpreter ignores IMPLICIT$ and IMPLICIT% statements.

TURBO's rules about the types of names differ in one small respect from those imposed by interpreted SuperBASIC, where the type of a parameter passed to a procedure or function is determined by the type of the corresponding value.

If TURBO did that it would never be able to decide upon the code needed to handle a parameter until the routine concerned was called, and it would have to make a decision during every call. In every case this would lead to slower, larger programs, with lots of redundant code and checking. This is precisely the sort of redundancy that TURBO sets out to abolish!

It follows that the type of each parameter must be known at the time of compilation, so that efficient code to manipulate the parameter can be generated. Thus, in a compiled program, the last character of a parameter name, or any previous IMPLICIT declaration, 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; these will not work under the interpreter, because Sinclair never implemented the considerable amount of extra code needed to handle them properly.

All current versions of the QL ROM, apart from "AH", contain a rather nasty bodge which stops you typing in integer or string names after SELECT or FOR. You can get around this by using IMPLICIT$ and IMPLICIT%, as described earlier in this chapter.

(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, so that TURBO can generate efficient 'customised' code for every case.

(iii) Definition names: the names used for procedures and functions. Every procedure or function must have an unique name - otherwise function-calls could be confused with array references, for instance.

TURBO treats EXTERNAL array and definition names just as it treats names declared in the program. IMPLICIT types may be used in one or more module that refers to a shared variable or definition, but the effective type of each name must be consistent in all the modules.

If an identifier is used for any one of the three purposes listed 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.

You can re-use names as LOCALs without restriction, as long as you keep them in one of the three groups listed above.

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.

TURBO ARRAYS

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 slightly different under the compiler than they are under the interpreter.

TURBO versions 4.20 and earlier only support array 'slicing' - the specification of more than one element of an array with a single reference - for the last dimension of strings. Later versions allow array slices of any sort to be passed as parameters to machine code extensions.

Unlike SUPERCHARGE, TURBO allows whole arrays to be passed as parameters of SuperBASIC procedures or functions. This mechanism is explored in Chapter P (P for Parameters!).

Array subscripts

A maximum array subscript may be any positive integer value, from 0 to 32767. Thus fast integer arithmetic can be used to extract elements. This can make array access more than twice as fast as would otherwise be the case, so we strongly advise you NOT to use floating-point values or expressions as subscripts in time-critical programs.

A single string array element may be declared to fit up to 32,764 characters. Odd lengths are rounded up to make them even, to suit the QL's 16 bit way of doing things. The interpreter allows up to 32,766 characters in a string, and gets very confused - unlike TURBO - if a temporary string ends up with a length of 32,767!

The difference of two between interpreter and compiler allows TURBO to treat the length (two bytes) as part of the string, which often saves time when accessing arrays. The maximum string length, while a parameter of DIM, is not treated as a subscript.

The total number of elements (not characters) in an array must not exceed 65,535. This means that a full-size floating point array is limited to a mere 393K. In theory TURBO could cope happily with a 2 million K string array.

You can't dimension an array unless it will fit in the dataspace of your task - but TURBO virtual arrays, documented at the end of this chapter, provide a way around this 'limitation'. The limit makes programs much faster than would otherwise be the case, although it doesn't seem to help the interpreter much! Remember that the first value of any subscript is 0, not 1.

TURBO indicates an overly-large array with an 'overflow' or 'out of range' message, rather than the misleading 'out of memory' which the interpreter can issue if subscripts in an array declaration are not to its liking, even if that array would not exceed the free memory in size.

TURBO always checks that array subscripts are sensible when a program runs. This helps to avoid mysterious errors, and compiled programs can perform this check very quickly - in about 3 millionths of a second, per subscript.

DATASPACE and array dimensions

This simple procedure lets you work out the amount of dataspace you should allow for any array:

(1) Add one to each of the 'maximum' subscripts used when the array is dimensioned. If this is a 'simple string' - a one-dimensional array of characters - go directly to step (c), below: the dataspace required is just the number indicated there.

(2) Multiply the results together; if this is a one dimensional array, just stick with the result you got from step 1.

(3) Multiply the result so far by the following factors:

(a) 2, for an integer array

(b) 6, for a floating-point array

(c) 2 PLUS the maximum length of each string, rounded up to an even number, for a string array.

The result is the amount of memory, in bytes, that TURBO will use to hold the contents of the array.

Using these rules, DIM A$(20,15,39) will use up:

20+1 * 15+1 * 40+2 = 14,112 bytes.

Remember that DATASPACE is not JUST used to store array contents, although these are often the biggest data items in a task.

Array re-dimensioning

When you re-dimension an array in SuperBASIC, the old values are discarded and new empty space is found. TURBO works slightly differently.

If you re-dimension an array to the same size as it was before, TURBO deduces that you want to clear the contents. This is much the fastest way to clear an array. In the case of a string array, only the lengths are cleared; the data values remain, but these are not accessible unless you deliberately slice them out.

Sloth and SuperBASIC

Any program that uses more than two dynamic areas will tend to be slow and inefficient, because areas keep bumping into one another as they change in size, and then others must be moved out of the way. You can minimise this problem by allocating more space, after each collision, than is immediately needed, but this is obviously wasteful.

The SuperBASIC interpreter uses no less than 13 dynamic areas - an inauspicious number! This means that it wastes large amounts of space keeping them safe distances apart, and even so it spends much of its time shuffling them around when collisions occur.

Sometimes - such as when you use more than nine locals or parameters in one routine - the collision checking, which also wastes time, is inadequate. Values spill over from one area to another, with unpredictable but generally unpleasant consequences.

SuperBASIC is made even less efficient by the bizzare way it expands areas towards lower addresses. According to QL developer Tony Tebby, the interpreter starts by moving ALL 13 areas down memory, using a routine about 1/5 the speed of MOVE_MEMORY, perhaps over-writing temporary 'slave block' file information which must first be written to disk or tape. Then it moves all the areas 'beyond' that where free space is needed, back up where they came from, using the same slothful algorithm.

There are reasons for this, but the result - particularly when loading SuperBASIC text - suggests that those reasons should have been re-examined as the QL was developed. Sadly the premature launch of the machine meant that this bodge was 'frozen' deep into the design of the interpreter.

Dynamic TURBO and mixed-type arrays

It should now be obvious why TURBO requires that you declare the maximum length of strings. This makes programs very much faster than they would otherwise be, and is rarely a great inconvenience on a computer with between 128K and 640K of memory. QL interpreters (including the one that calls itself a compiler!) let you get away without dimensioning 'simple', unsubscripted strings, but they handle them slowly and usually waste space between them, as noted above.

If you need to store lots of items, of varying lengths and maybe several types, an array is not really the best data-structure to use. TURBO TOOLKIT gives you the facilities needed to maintain a 'heap', where text or any other information can be stored, packed back to back more densely than would be possible in an array.

You should use an area of memory allocated with RESPR or ALLOCATION, and an index of data addresses or offsets. SEARCH_MEMORY, PEEK$, POKE$ and MOVE_MEMORY let you scan and move things around in many ways that you can't manipulate array elements. SEARCH_MEMORY can look through about 220K in a second, although this speed can fall somewhat if you're using slow RAM or there are lots of 'near matches' in the area being searched.

String concatenation, together with the FLOAT$ and INTEGER$ functions, lets you build 'records' or 'structures' which will be familiar to Pascal and C users. These are composite data-items containing several elements but accessed as one. Thus you can pack several different types of data into a single entity.

Chas Dillon has proved the effectiveness of precisely this approach in his package THE EDITOR, published by D.P, which uses TURBO TOOLKIT commands to beat the pants off other QL editors, handling text fast and economically.

This technique is worth considering if you're writing data-handling programs with TURBO.

Rubber arrays

If you re-dimension an array so that only the 'last' subscript changes. TURBO will expand or contract the area used to hold the array, PRESERVING the old values, or those that will fit, counted from element zero upwards, if you make the array smaller. In the case of a string array the 'last' subscript is the subscript immediately BEFORE the maximum length.

This feature, known as 'rubber arrays', is very powerful. It is not supported by the interpreter, but this rarely causes problems when a program is tested.

Rubber arrays allow a compiled program to re-assign data space between arrays as they run, without losing required information. If you want to re-dimension AND clear an array you can do this easily enough - just re-dimension an array to the new size, to allocate or de-allocate space, and re-dimension it again with the same subscripts, to clear the contents.

Shrinking and stretching can take a while, at least in TURBO terms - a small fraction of a second, if you've got several hundred K of other arrays - but it is unlikely that you'll want to do it many times every second! The small delay is usually well worth while.

Arrays, dataspace and 'virtual' memory

LOCAL or DIMensioned arrays are created in the dataspace of a compiled task, so you must set this appropriately (using the DATASPACE utility, or the DATA_AREA directive, or the dataspace control on the TURBO front panel. All of these controls are explained in Chapter E.

A compiled task will print an appropriate message if it runs out of dataspace. Excessive array dimensions for that dataspace allocation are the usual cause of the problem - the message will probably indicate the offending DIM line - but dataspace can also run out if routines keep calling one another without RETurning.

The main differences between virtual and conventional arrays are that there are no subscript limits and data is stored on the specified medium,

rather than in memory. Any unused memory outside your task is used as 'slave blocks' - buffers to speed up access to the virtual array, so that the computer doesn't always have to read the disk or cartridge to find data, especially if you work sequentially through an array.

As the routines are written, arrays of 1, 2 or 3 dimensions are supported, but you must pass one or two maximum subscripts to the virtual array routines when you access 2 or 3D ones. You can change this, by using global variables, if you find it annoying, but the present implementation is easier to bolt onto existing programs.

One other difference is that you have to use a procedure call, rather than an assignment, to store values in a virtual array. This should not cause any problems.

TURBO virtual arrays work fine - albeit more slowly - in interpreted SuperBASIC.

CHAPTER L - FLOATING POINT AND LOGICAL ARITHMETIC

FASTER AND BETTER FLOATING-POINT

TURBO's floating-point mathematics routines can accurately compute and display nine decimal digits, rather than the seven or eight shown by the interpreter.

Whether you're colossally wealthy, or past Governments of your country have taken an enthusiastic approach to 'inflation', or you're just plain pedantic, you may find this extra precision very appealing, especially as TURBO floating-point operations are much faster than those of the SuperBASIC interpreter.

The perils of points

It is important to remember that TURBO calculations offer the same potential pitfalls as any other digital computer. These are rarely explained properly, and we would advise anyone programming the QL to handle money to read, or at least scan through, the remainder of this chapter, whether or not they intend to use TURBO for these programs (and if not, why not?!).

Errors may occur if monetary quantities are entered as decimal fractions. 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 (relatively slothful) IBM PC.

To understand the problem, and the best solution, you need to consider the nature of fractions, numbers, and the way a computer works. In the interests of clarity, albeit at the risk of boring you, we'll explain things from first principles.

Counting Arabs

In the 'decimal' arithmetic system which we have inherited from the Arabs - but not, for instance, in Roman numerals - the value of a given digit depends upon its position.

We all know the rules so well that we no longer think about them. A digit in a certain position represents ten times the value it would in the position to the right, and a tenth of the the value of the same digit written to the left.

This is actually quite an ingenious system, as you learnt, hopefully, at primary school and can quickly re-discover. Try to do some simple arithmetic in Roman numerals - say XVII times XXIV - without cheating and converting the numbers to decimal!

If you want to write a whole number - a number with no fractional part - you can always represent it exactly using decimal notation; you just write down the number of units, tens, hundreds and so on up until the answer is complete.

Fractions are not so easy. We put a decimal point after the digit representing a certain number of units. Thereafter, each digit represents a certain number of tenths, hundredths and so on.

But some fractions cannot be written precisely in decimal, however many digits you use. One obvious example is a third, where you need an infinite number of 'three' digits after the point in order to represent the fraction precisely.

Another interesting case is the fraction one seventh, where you must repeat the digits 142857 over and over again after the decimal point. The more digits you use, the more accurate the result; but you can never represent either of these fractions exactly using the decimal scheme.

So far this may seem obvious and irrelevant - but we have established that decimal arithmetic cannot represent some fractions precisely.

Alternatives to decimal

Now imagine a numbering scheme where the value of each successive digit was three times that of the same digit written to its left. The columns then go (for example) nines, threes, ones, thirds, ninths, and so on. Thirty would be written 1010.

If we use this numbering system, the fraction a third CAN be written exactly - as 0.1 (the one after the point represents one third). Similarly one ninth can be written exactly as 0.01 - and one ninth is another number that could not be written precisely with any number of decimal digits - it's a third of 0.333333333 etc, etc.

The system based on multiples of three is, logically enough, called 'base 3'. The ratio of each digit position to its successor is three. Only three posible values are needed for each digit position. Counting goes 0, 1, 2, 10 (one three and no units), 11, 12, 20, 21 and so on.

We can also resolve the problem of writing one seventh accurately, if we make the 'base', or ratio of position values, seven. We must allow each position in a number to represent up to seven values (digits from 0 to 6). Then we can write a seventh exactly as 0.1, in base seven.

We still need an infinite number of digits to write a third in base seven - but the sequence of values is different - a third, in base 7, is 0.222222 recurring! You can check this on the QL, if you're so inclined, by adding two sevenths, two 49ths, two 443rds (443=7*7*7) and so on.

Having brought the QL into the picture we'd better consider how it copes with fractions. As you probably know, all modern digital computers work with 'binary' or base two arithmetic. This is because it's easy to make electronic circuits work with two states - on and off, or 0 and 1.

Binary coded decimal

It's not practical to make computers with ten levels between 'off' to 'on', in order to process decimal digits directly - there's too much risk of

confusion between similar levels.

Of course, you could use groups of binary digits to represent each decimal one. The QL's 68008 processor does support such a scheme, called Binary Coded Decimal - BCD for short. Four binary digits are used to store every decimal one, so BCD values need more space than binary for a given degree of precision.

BCD allows exact decimal fractions to be stored, but it is horribly slow. In the days long before micros, IBM made a computer designed to work entirely with BCD values, but the idea didn't catch on. Pocket calculators often use BCD, but that's because they only have to do a few calculations in a second, at most.

The QL can add 32 binary digits in the same time that it can add two BCD ones. More significantly (no pun intended) the 68008 can multiply and divide binary values with a single instruction, whereas in BCD this must be done by repeated addition or subtraction. It's no surprise that TURBO, like the interpreter and all the other 68000-based computers we know of, uses binary rather than BCD.

It follows that halves, quarters, eighths and so on can be stored exactly in a few bits (0.1, 0.01 and 0.001 respectively), as can fractions that can be made by adding these values. For instance, three quarters is .11 in binary (a half plus a quarter).

All other fractions can only be represented approximately. A third in binary is written 0.01010101 etc. Again, you can easily check this if you don't believe us.

Humans tend to use fractions like tenths and hundredths a lot, because they're used to decimal maths. But a tenth, when written in binary, is an infinite sequence:

0.00011001100110011...

Unsurprisingly, a hundredth also goes on for ever, if written as a binary fraction.

The correct answer

You can avoid all problems with binary arithmetic if you follow one simple rule! Always work in WHOLE UNITS.

As we saw at the beginning of this discussion, it is only fractions that cause problems. If you make sure that you work in pence, or cents, or whatever your basic unit is, you'll never get the wrong answer, so long as you stay within the number of digits the machine can hold accurately.

Try to avoid division except by exact factors, or multiplication by fractions - these operations will always give approximate results. Be sure to round numbers consistently thereafter.

You must use EDIT$ or string INPUT statements to read numbers, so that floating-point approximations never get a chance to creep in. Check that the decimal point is in the right place in the entry, then remove it, to get a whole number which you can process without problems. Insert the decimal point - using string operations, once again - before you print results.

Living dangerously

Some books and articles may suggest different solutions, which essentially revolve around scaling the number up, chopping off a few digits and scaling it back down. This will generally work by obscuring errors that creep in when numbers are entered as decimals.

This alternative is slow and potentially unreliable. Errors can accumulate when approximations are added together, until they become obvious - hopefully obvious to you, rather than your auditors or the Tax Inspector! This technique also restricts the range of numbers you can use: there's a risk that a large number may be scaled beyond the number of digits that can be stored precisely.

Scientific detachment

Most 'scientific' users, and others who need fractions to model the real world, should not be upset by this discussion. When you're working with arbitrary values, binary arithmetic is no less accurate than decimal. Indeed, it packs more accuracy into a given amount of memory than BCD can.

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.

Illogical, Captain!

The QL is rather unusual in one of its uses for floating- point values. It treats the result of a 'logical operator' (things you often use in an IF statement, such as NOT, AND, ==, >= and so on) as a floating-point value, even though logical values are usually only 1, meaning TRUE, or "affirmative" if you're a Vulcan; and 0, meaning FALSE or "negative, Jim".

In fact the QL treats any value other than zero as logically TRUE, and because this includes all the possible floating-point numbers TURBO must allow any of them in an IF test. If you type:

```
IF 123456.7: PRINT "YES"
```

the conditional part of the command WILL be executed, and there won't be an error - most computers use 16 bit integer values to represent TRUE and FALSE - they would reject this command because 123456.7 is not such an integer.

TURBO supports this SuperBASIC quirk, so that programs work consistently in this regard whether interpreted or compiled. For consistency's sake, ALL logical values must be in floating-point format before they're tested by IF, AND and the rest. This happens efficiently and automatically under most circumstances, but you can mess things up and create extra work if you use integer flags.

If you set and test a flag in your program like this:

```
flag% = pointer<131072  OR  pointer> 786431
IF flag% : PRINT "Invalid QL RAM address"
```

TURBO will have to convert the floating-point logical value returned by 'OR' into an integer in order to put it into FLAG%. TURBO must also convert it back to floating-point whenever you test that variable. These needless conversions waste time, so you should use floating-point variables as flags, if you want your programs to be executed efficiently.

Furthermore, it is faster - although slightly more code is needed - to check that integers are non-zero by comparing them with zero (IF int%<>0 THEN) rather than by testing them directly (IF int% THEN).

Esoterica

There are some very minor errors in early ("AH" and "JM") versions of the QL ROM floating-point routines. These can cause slight inaccuracies - typically 1 part in 1E10 - when some decimals are added or subtracted. These errors are fixed by TURBO, and were also fixed by SUPERCHARGE.

Back to Business

In British financial programs you should work in units of 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.

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.

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
```

The next routine, INPUT_MONEY, will accurately read monetary amounts from a specified channel, positioning the cursor at the character co-ordinates you supply. The code is rather long when compared with a simple INPUT, but it is a lot more comprehensive.

The function checks for five possible errors, rejecting very brief entries and 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.

INPUT_MONEY calls a second procedure, COMPLAIN, if an error is detected. COMPLAIN just prints a relevant message on the top line of the window, in this example, but it's easy to change it to do something else.

```
DEFine FuNction INPUT_MONEY(channel,x,y)
LOCAL pence, pounds, digit, minus, pointpos, money$(10)
  REPeat try
    AT #channel,x,y
    CLS #channel,4
    money$=EDIT$(#channel,0,10)
    IF LEN(money$)=0
      COMPLAIN "Entry too brief.":NEXT try
    END IF
    minus=money$(1)="-"
    IF minus
      IF LEN(money$)=1
        COMPLAIN "Entry too brief."
      END IF
      money$=money$(2 TO)
    END IF
    FOR digit=1 TO LEN(money$)
      IF NOT (money$(digit) INSTR ".0123456789")
        COMPLAIN "Invalid character in entry.": NEXT try
      END IF
    END FOR digit
    pointpos="." INSTR money$
    IF pointpos=0 OR pointpos=LEN(money$)
      pounds=money$
      pence=0
    ELSE
      IF "." INSTR money$(pointpos+1 TO) THEN
        COMPLAIN "More than one point.": NEXT try
      END IF
      pounds=money$(1 TO pointpos)
      pence=money$(pointpos+1 TO)
      IF pence>99 THEN
        COMPLAIN "Invalid number after point.": NEXT try
```

```
      END IF
    END IF
    IF pounds>=1E8
      COMPLAIN "Number too large.": NEXT try
    END IF
    IF minus
      RETurn -(pounds*100+pence)
    ELSE
      RETurn pounds*100+pence
    END IF
  END REPeat try
END DEFine INPUT_MONEY

DEFine PROCedure COMPLAIN(message$)
  AT #channel,0,0
  CLS #channel,3
  PRINT #channel;"**** ";message$;
END DEFine COMPLAIN
```

CHAPTER M - FAST ARITHMETIC WITH INTEGERS

INTEGERS FOR SCIENTISTS

This section is aimed at TURBO users who have a mathematical inclination and who need to be able to compute the values of advanced maths functions such as SIN, LOG and EXP.

In SuperBASIC, the advanced maths functions are computed with a high degree of accuracy - more than is necessary for virtually any application. By default, TURBO computes such functions to nine decimal digits of precision - and NASA only needed to use PI to an accuracy of seven digits to land Apollo XI in the right hole on the moon!

So what's the problem? Just that you don't get owt for nowt. TURBO makes heavy use of the QL ROM routines, to ensure compatibility with the interpreter, and it takes a long time for the 68008 to work out such accurate results, even though the TURBO calls is efficient. TURBO is still faster than the interpreter, because it avoids needless syntax checks and other run-time indulgences.

You may feel that Sinclair got the trade-off between accuracy and speed completely wrong, in which case you can correct the mistake! TURBO in-line code is so fast that you can re-code maths functions in compiled SuperBASIC and get a useful speed improvement, with no need to resort to machine-code.

Look-up tables

The simplest way to speed up the handling of advanced maths functions is to 'look-up' values as required in pre-calculated tables - rather like the printed tables that were, until recently, often used in schools.

For instance, a floating point array of 91 elements could be used to hold the values of the sines of integer angles in the interval 0..90 degrees. This interval is chosen because once you know the values of the sines in such a table it is trivial to compute the values of sines of other angles. This computation involves only addition and subtraction, which TURBO can perform very quickly. If you doubt this, draw or plot a sine graph and see how each 90 degree section resembles the shape of the first.

You only need to fill the array once, at the beginning of the program. The array contents may be read from a file, DATA statements, or computed once-and-for-all.

In-between values

'Linear interpolation' - which means working out intermediate values by conceptually 'drawing a straight line' between known, adjacent ones -gives reasonably accurate results for non-integer angles in between those held in the table. If X is the angle, and N% is an integer such that 0 <= (X-N%) < 1 (i.e. N% is the largest integer not greater than X), we can apply linear interpolation to get this approximate formula:

f(x) == f (n%) + (x-n%) * (f(n%+1) - f(n%))

Hence to calculate SIN(22.71 deg) we look up the values of SIN(22 deg) and SIN(23 deg) and use the formula above:

SIN(22.71 deg) == SIN(22 deg) + 0.71 * (SIN(23 deg) + SIN(22 deg))

How precise is this method? This depends upon the function you're approximating. As long as the function is continuous, and not too rapidly-changing, in the area you are interpolating over, the technique is very accurate: for example, the last equation yields SIN(22.71 deg) with 99.997 per cent accuracy - about four and a half decimal places, which is usually plenty good enough for engineering purposes.

In some cases the domain of the function - the range of parameter values - makes integer increments inappropriate - there would either be too few or too many array elements. One such function is ARCSIN, where the domain is -1 .. +1.

In such a case you must choose a small increment which will yield a reasonable number of array elements - say 100. We can get 100 steps between -1 and +1 if we use a step-size of 0.02. Thus the array should contain the values of ARCSIN(X) for values of X as follows: -1, -0.98, -0.96 ... -0.02, 0, 0.02, 0.04 ... 0.96, 0.98, 1.

Allowing for the end values, we need an array of 101 elements. If the name of the array is LOOKUP, it's values might be assigned as follows:

LOOKUP(0) = ARCSIN(-1)

LOOKUP(1) = ARCSIN(-0.98)

...

LOOKUP(49) = ARCSIN(-0.02)

LOOKUP(50) = ARCSIN(0)

...

LOOKUP(99) = ARCSIN(0.98)

LOOKUP(100) = ARCSIN(1)

In short, to find out the value of ARCSIN(X) you look up the value of the (X / 0.02 + 50)'th element of LOOKUP. It is generally faster to write this with a multiplication, rather than a floating-point division:

ARCSIN(X) = LOOKUP(X * 50 + 50)

You can improve this further by adjusting your program to shift ARCSIN values into the range 0..2, thus avoiding the floating-point addition step, but that may not suit you.

In order to calculate the ARCSIN of a number which is not an exact multiple of 0.02 you can use linear interpolation again. If X is that number, and Y is the largest multiple of 0.02 which is not greater than X:

ARCSIN(X) == ARCSIN(Y) + (X-Y) * 50 * (ARCSIN(Y+0.02) - ARCSIN(Y))

Hence, if we wished to compute ARCSIN(-0.367) we would use:

ARCSIN(0.367) == ARCSIN(-0.38) + 0.013 * 50 * (ARCSIN(-0.36) - ARCSIN(-0.38))

In general, if we have chosen an interval of K (K>0), and if X is the number whose function value is to be computed, choose Y so that it is the largest element in the set of precalculated function values that is not larger than X, i.e. 0 <= (X-Y) < K. Then use:

f(x) == f(y) + (x-y)/k * (f(y+k) - f(y))

An alternative implementation of linear interpolation, which sometimes gives more accurate results, involves two arrays. One holds the precalculated function values, the other holds precalculated values of the derivative (gradient or slope) of the function. If f'(x) = d(f(x))/dx (the slope) then:

f(x) == f(y) + (x-y)*f'(y)

Some functions change values too rapidly for linear interpolation to yield reasonable results, and drastically decreasing K has a disastrous effect on array size. The TAN function, for instance, changes fast in the vicinity of PI/2. In such cases we recommend that you use the built-in advanced maths functions, but only for the 'problem' values. Alternatively you can use a convergent series, which is the next technique discussed here.

Convergent series

A convergent series is a class of formula which expresses the value of a function. The word 'series' indicates that the formula is made up of a series of terms which are mathematically related. 'Convergent' indicates that the more terms you use, the more accurately the result resembles the function.

Summing the first few terms of a suitable convergent series is a good general alternative to using look-up tables in all cases where storage is short or the domain of values to be covered is large. A look-up table for EXP(X) is not very practicable!

We list some of the most important series here. Summing them to just three terms will give between two and three decimal digits of accuracy - good enough for most graphical purposes.

Note that all angles above, and from now on until the end of this chapter, are expressed in radians. One radian is 180/PI degrees.

If an angle is not in a suitable range, convert it using relationships like these:

SIN(X) = SIN(PI - X)

COS(X) = - COS(PI + X)

SIN(X) = SIN(2*PI + X)

In some cases a restriction is specified at the end of the line, by an entry in the column of single letters. These letters have the following significance:

A: The series only works with the restriction shown after this letter (e.g. TAN).

B: The function itself has a restricted domain (e.g. ARCSIN).

C: The series doesn't converge fast enough without the restriction (e.g. SIN).

To derive convergent series for trancendental functions of your choice, apply Taylor's expansion:

f(x) = f(y) + f'(y)/1!*(x-y) + f''(y)/2!*(x-y) 2 + ...

When defining such functions ensure that the function name you use is not a SuperBASIC reserved word (LN, SIN, TAN, etc). Make the maximum use of integers and in-line code ('REMark +' - see Chapter U) or the speeded-up function will be embarrassingly slower than the standard one - the difference in speed between integer and floating-point operations cannot be over-emphasised.

This is an example of a very fast function to calculate SIN(X). It assumes values of X are in radians, between 0 and 2*PI. Please study it carefully, and read the notes later:

```
DEFine FuNction FAST_SINE(X)
LOCAL Y%,Z%
  IF X>1.5707963
    IF X<3.1415927
      X=3.1415927-X
    END IF
  END IF
  IF X>3.1415927
    IF X<4.7123897
      X=9.424778-X
    END IF
```

```
    END IF
    IF X>4.712389
      X=X-6.2831853
    END IF
    Y% = X * 100
    Z% =Y%-Y%*Y%/125*Y%/480+Y%*Y%/125*Y%/160*Y%/160*Y%/3750
    RETurn 0.01*Z%
  END DEFine FAST_SINE
```

The computation of Z% is complicated by the 68008's restriction that integers, including interim results, must lie in the range -32768..32767.

As X has been forced into the interval -PI/2..PI/2, Y%=INT(100*X) must lie between -158 and 157. The choice of 100 as the scaling multiple is arbitrary but may be convenient - a more precise result would be obtained if we used INT(SQRT(32767) /1.5707963) = 115.

In order to ensure that no intermediate answer lies outside -32768 to 32767, expressions like Y%*Y%/125*Y%/480 are used. In the worst case, when Y%=158, the intermediate results work out as 24964, 199 and 31442 (integer division, remember!).

Divisions are performed as late as possible, and by as small a number as possible, to keep results accurate.

The third term should have a denominator of 5!*100 4 = 1.2E10. We have factorised it, in this case, to keep temporary results as large (hence accurate) integers.

We could have saved space and a little time by re-using the variable Y% instead of using Z% - but the listing would have been less clear as a result.

In the penultimate line, we multiply by 0.01 rather than divide by 100, even though 0.01 cannot be represented exactly in binary (see Chapter L). Multiplication is faster than floating-point division, and if we divided Z% by 100 without further thought TURBO would use integer division, always giving a result of zero!

CHAPTER N - STRING AND TEXT HANDLING

FAST STRING HANDLING

TURBO can do more or less anything that the SuperBASIC interpreter can do with strings - but much faster. TURBO and the 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. 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 result is that TURBO is extremely fast at slicing and appending strings, but its advantage is reduced if lots of machine-code function-calls are performed. 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.

TURBO also borrows sophistication from the MOVE_MEMORY function when it needs to move strings around memory - long strings are moved 3.5 times faster than SUPERCHARGE could - although SUPERCHARGE was pretty speedy.

The passing of string value parameters has been accelerated by a factor of about seven compared with SUPERCHARGE, and REFERENCE parameter-passing, documented in chapter P, can save even more time. All in all, TURBO string handling is extremely powerful. The main limit on string-handling speed is the memory in your QL, and Chapter U explores this fact, among others.

TURBO also supports 'rubber' string arrays - arrays that you can re-dimension without losing the contents - and 'virtual arrays', which are simulated arrays stored on disk or microdrive, buffered automatically in spare memory. Chapter K contains the full low-down on TURBO arrays.

String dimensions

All strings in TURBO must be dimensioned. Any strings which are not explicitly dimensioned will be set to have the default length. This is usually 100 but can be altered to be between 40 and 257 by Configuring Parser_task. TURBO warns you when it dimensions strings for you - this provides a handy guide to the amount of dataspace that the compiled task will require.

The SuperBASIC interpreter finds space for un-dimensioned strings on an 'ad hoc' basis - if TURBO did this it would tend to run much slower and be wasteful in its use of memory.

In many cases the longer an interpreted string-handling program runs, the more memory it ties up, until eventually it stops with an OUT OF MEMORY error. This happens when the interpreter's storage becomes 'fragmented' by continual allocation and de-allocation. TURBO tasks NEVER suffer from this problem.

Any string length between 1 and 32,764 characters is allowed, but odd lengths are always rounded up to the nearest even number; the SuperBASIC interpreter does much the same thing.

String parameters of SuperBASIC procedures and functions are automatically created as LOCALs when a routine is called. Their maximum length will match the parameter value.

Since all TURBO strings are dimensioned, they cease to exist after a compiled CLEAR statement. TURBO issues a special warning message to alert you to this; it is particularly important to take heed of it if you do not dimension all of the strings in your program yourself, but leave this to TURBO.

If your program tries to access a string which no longer exists it will stop with a NOT FOUND report. If your program uses CLEAR while it runs you should make sure that you re-declare all the strings you will want to use later.

String slicing

You can slice 'simple' string variables, or the last dimension of string arrays, with TURBO just as you can with SuperBASIC.

TURBO takes the view that the default 'end' slice is taken to be the latest LEN of the string, unless this is zero or an insertion is being made:

```
A$(3 TO)="LIZARD"
```

...in which cases the physical maximum length is assumed. This will almost always match the way that the interpreter works, although there are some cases when dimensioned and undimensioned strings are treated inconsistently by the interpreter.

If these cases affect your program it will be obvious when you run it in SuperBASIC. If you run into trouble when testing programs intended for TURBO under the interpreter you can put things right by removing, or starting the program after, the DIM statements that TURBO expects. Restore the code, if need be, once you have finished interpretative testing.

Unlike SUPERCHARGE, TURBO lets you slice string literals in your programs, so lines like this work fine:

```
PRINT "BAD GOODBEST"(rating%*4+1 TO rating%*4+4)
```

SuperBASIC string parameters

Confusion can arise if a string is declared in a program as a parameter or a LOCAL, but never DIM'ed to exist outside. TURBO can't tell whether or not references to the string elsewhere in the program refer to the local variable, used elsewhere after a GO TO or subroutine call.

You have a choice about how TURBO resolves this kind of ambiguity. You can have the names of such strings reported to you, or have them

automatically created (just in case). These options are controlled from TURBO's main menu, using the switch second from the left on the bottom line. The first option corresponds to 'REPORT $' (report strings), and the second to 'CREATE $'.

If you really know what you're doing, and you're sure you've put in all the DIMs you need - or you want a message at run-time if you've overlooked anything - you can tell TURBO to 'IGNORE $', in which case you won't get any irritating warnings when you compile your program, but you still get a 'NOT FOUND' report if TURBO tries to use a string when it doesn't exist.

String parameters and 'non-standard' procedures

SuperBASIC is ambiguous in that strings can be expressed in two forms when they're passed as parameters to a machine-code procedure. You can type:

```
DIR FLP1_  or  DIR "FLP1_"
```

The first parameter is termed a 'name', whereas the second is a normal string value.

If you use disk system, ROM cartridge, or other non-standard 'extension' commands in your compiled programs TURBO has trouble telling names that are meant to be strings from names that refer to variables.

Remember that TURBO has to decide the type of parameters by examining your program - it's too late to change things when the compiled task is running, as fast, special-purpose code will have been produced to handle the assumed type.

If TURBO comes across a name used as a parameter of a non-standard procedure, it will assume that it refers to a floating-point variable, unless it ends with a "$" or "%". If the name is meant to be taken literally, as in the first DIR command instanced earlier, this will be the wrong assumption!

In fact DIR will not cause any trouble, because TURBO 'knows' about all the standard QL commands - the ones documented in the QL User Guide - and all the TURBO TOOLKIT commands.

But other, 'non-standard' commands could require numeric or string parameters. In these cases, if a name is intended to taken literally it must be set explicitly as a string.

INPUT buffers

The "AH" and "JM" versions of the QL had a restriction, in that you could not INPUT strings of more than 128 characters, from SuperBASIC. The interpreter would complain BUFFER FULL and stop, leaving your file open.

TURBO is more sensible - it makes a buffer amongst the free memory inside your task during INPUT operations. This buffer is always at least 128 characters long - if not, the compiled task stops and asks you to increase its dataspace - and may be up to 32,767 bytes long (the maximum that can be read serially in one go) if there's enough free memory.

If you get a BUFFER FULL error when reading a string you can generally cure it by increasing the dataspace of the task. It's generally a good idea to check that the file contains end of line markers (CHR$(10)) before you bother to do this!

CHAPTER O - TURBO DISPLAY HANDLING

THE QL DISPLAY - STRENGTHS & WEAKNESSES

TURBO's display handling speed is limited by the design of SuperBASIC and the QL. TURBO is compatible with every SuperBASIC text and graphics operation. This short chapter summarises the implications of this, and suggests some ways you can circumvent them.

Ideally TURBO would have been supplied with a completely new text and graphics 'device driver', but this would involve several months of extra work re-writing large chunks of the QL's ROM. Reluctantly, we've had to put this off for the time being. Nonetheless, the pictures in this chapter show the impressive results that can be achieved with existing facilities and a little care.

Display fundamentals

The QL uses a block of 32K of memory to hold the display, whether you use MODE 8 or MODE 4. This is a lot of RAM for the processor to keep track of, or to move en masse during operations like CLS and SCROLL. The arrangement of points within this area is also rather odd, apparently to keep the hardware simple. The result is a very detailed but sluggish colour display.

Hardware problems associated with the QL's display are discussed in the section of Chapter U headed 'Speed and Memory'. This chapter deals mainly with the software which controls the QL display.

Character displays

The speed of printing to the display is hamstrung by all the options that may be used. Every time a character is printed the system must take account of arbitrary windows and positions, plus two MODEs, 6 CSIZEs, and sundry settings of INK, PAPER, FLASH, OVER, UNDER, STRIP, FONT, and so on.

TURBO cannot make assumptions about these controls without restricting the compatibility of compiled and interpreted programs. Even CONTROL F5 - used to pause the display - must work perfectly in compiled code.

TURBO uses a variety of tricks to minimise the overhead of writing to the display, but the basic time taken to write all the pixels of each character cannot be avoided. SCROLL and PAN in large windows are also slow, because of the amount of memory that must be moved - you will notice that the TURBO listing is produced much more quickly when you limit the report window to one or two lines.

NOTE: Chas Dillon's EDITOR, published by D.P, was written in SuperBASIC and compiled with SUPERCHARGE and TURBO. THE EDITOR was carefully designed to avoid needless re-writing of the screen. It's many times faster than QUILL, which was developed using terminals, rather than the real QL display!

If the speed at which text is displayed is a problem your best bet is to write strings in one go, rather than as a sequence of single characters, whenever possible. If you can avoid writing anything at all, so much the better. PAN, SCROLL and the TURBO TOOLKIT commands PEEK$, POKE$ and MOVE_MEMORY can be very useful in such cases; in particular, they allow you to save and restore parts of the display almost instantaneously.

Graphics displays

SuperBASIC graphics commands are very sophisticated, with their support for floating-point scaling, shifting and clipping, but this also makes them slow, especially when short lines or points are being plotted. The range of commands is well-suited to technical work but pretty useless for games - there's no support for automatic movement, masking or animation; no multi-coloured symbols and no proper way to detect 'collisions' between graphics.

Such applications demand a completely new set of commands. Digital Precision's 'Super Sprite Generator' is a library of machine-code procedures, functions and utilities for games programming or animation; they are very fast and work well when called from TURBO compiled SuperBASIC.

Area graphics operations such as BLOCK, CLS and FILL use integer co-ordinates and are therefore much more efficient - unfortunately the sheer amount of data that must be handled restricts their speed when working with large areas.

TURBO TOOLKIT commands can be very useful in graphic applications -several demonstrations of this appear in the file DEMOS_BAS. Remember that you need to compile these programs to see them at their best.

CHAPTER P - PARAMETER PASSING

TURBO PARAMETERS

'Parameters' are the numeric or textual values which are sent to a procedure or function by listing them after the name of the routine. Parameters used when calling a routine are termed 'actual parameters'. The names which appear in round brackets after a definition are called 'formal parameters'. In other words, 'formal' parameters are used inside a definition, as formal names for the 'actual' parameters supplied when the definition was actually invoked.

SuperBASIC routine parameters

The SuperBASIC interpreter is very lazy about parameters - it uses a scheme called 'late binding', which tends to characterise slow interpreters. 'Late binding' means that an interpreter does not worry about the types of parameters until the moment they are used. This can be convenient if you're bodging a program together, because you don't have to worry much about types as you go along, but it has two big disadvantages:

(a) It's very hard to tell whether or not a routine will ALWAYS work, as the performance of a definition depends largely upon the actual parameters; you can look these up by examining the whole program, but this can take ages - with a consequent risk of error - if the program is long or parameters are passed around between several routines.

(b) The interpreter can give you very little help when a problem occurs - a string cannot be converted to a number, or a parameter (which might belong to any routine) is not found when it is needed. In such cases the interpreter can tell you where it stopped, but not where it really got into trouble. These two places are often hundreds of lines apart.

(c) Any interpreter must check the existence and type of of every parameter whenever it is used. This is a slow process - it often takes longer than working out the result, which must be checked again, and maybe converted, before it can be stored.

TURBO gets around all of these problems - and works more quickly, reliably and helpfully into the bargain - by fixing the types of SuperBASIC parameters when the program is compiled. This means that it can tell you if parameters are missing in some calls (you can always add zeroes, if the value is not important), or if extra parameters are present - interpreters just ignore these! In both cases TURBO indicates the lines containing each incorrect call. These are the lines you'll need to fix - not the error positions indicated by the interpreter.

You can clear up such problems by compiling a program with TURBO and reading the resultant report.

Unlike an interpreter, there's no extra cost when TURBO uses a parameter, rather than another variable. TURBO works out the type from the last letter of each formal parameter name. TURBO does the same with SuperBASIC functions. Dollar means string, per cent means an integer, and anything else is a floating-point value. TURBO also lets you over-ride these types with the IMPLICIT directives, documented in Chapter M and N, but the type of each name still stays the same throughout the program.

TURBO can generate exactly the right sort of code to handle any data-type. All the decisions are made when the program is compiled -not while it is running - so TURBO parameters are handled at top speed.

Reference parameters

The interpreter works out whether a parameter is passed by 'reference' or 'value' using an ad-hoc rule. The difference is small but important. Changes to a 'value' parameter have no effect outside the routine -they only affect a local copy of the value - whereas changes to a REFERENCE parameter alter the actual parameter as well as the formal one.

In other words, if a parameter is passed by REFERENCE the formal parameter name is really just a temporary substitute for the name of the actual parameter. Parameters can only usefully be passed by reference if the actual parameter is a variable-name, rather than a calculated value. Changes to a calculated value (as in this example) would not be very useful:

```
REFERENCE X
DEFine PROCedure DOUBLE(X)
  LET X = X + X
END DEFine
```

The REFERENCE statement is a TURBO directive, which we explain more about later.

You can call this procedure with the statements:

```
X=2
DOUBLE X
PRINT X
```

which will print 4. But if you wrote:

```
DOUBLE 2
```

TURBO would do the work but no variable would be changed, because you've not supplied a name. TURBO works out 4 and then 'discards' it. It doesn't change the value 2 in the program, into 4 - this would endless confusion and would be most unlikely to help you! You should not expect a value to be returned 'by reference' unless you tell the computer somewhere to put it.

The interpreter decides whether or not a parameter is passed by reference by looking at the actual parameters - names are passed by reference, and anything more complicated is passed by value. Again, this is sometimes convenient but, more often than not it leads to confusion. Once again, you can't always tell what a routine will do unless you examine all the calls to it, as well as the code in the routine and the code of everything directly or indirectly called.

TURBO wants to generate consistent, purpose-made code. Most compiling languages do this by adding extra words (VAR, in Pascal, or REF in Algol) in the declaration of a routine, to indicate that certain formal parameters always pick up values by REFERENCE. The SuperBASIC interpreter won't let us do quite this - it complains or runs incorrectly if you insert anything except spaces in a list of formal parameters - so TURBO has to use a slightly different trick.

Inside a single program, TURBO passes parameters by value unless you tell it otherwise. This is generally fine, but in cases where you may want to change an actual parameter you can indicate this by listing appropriate formal names in a REFERENCE directive, immediately before the DEFinition of the routine.

REFERENCE parameters are useful if you want to pass a value 'back' from a routine, although you could always use a function, or less-local variables instead, to save a little time that would otherwise be spent linking and unlinking names.

You can pass arrays by reference, but not by value. The interpreter imposes this rule too, as it saves the need to allocate and fill big 'temporary areas'. It doesn't matter, and saves lots of time, as long as you only change array parameters when you really WANT to do so.

This routine searches an array for the values 0 and 100, returning the number of instances of each value in the last two parameters:

```
REFERENCE table(0),nulls%,tops%
DEFine PROCedure search(table,size%,nulls%,tops%)
LOCAL i%
  i%=0: nulls%=0: tops%=0
  REPeat scan
    IF I%>size% THEN EXIT scan
    IF table(i%)=0 THEN nulls%=nulls%+1
    IF table(i%)=100 THEN tops%=tops%+1
  END REPeat scan
END DEFine search
```

Notice that the 'dummy subscript' 0 is used in the reference statement, to show that TABLE is a one-dimensional array. This is not needed (or allowed!) in the SuperBASIC DEFinition - another reason why we need a new REFERENCE directive. If TABLE was a three dimensional integer array the line would contain TABLE%(0,0,0).

SIZE% - the number of elements to be scanned - has been passed by value, as we don't expect it to change. You could use DIMN(table,1) here, in interpreted or compiled code, if you always want to search the whole array or a fixed proportion of it. It wouldn't MATTER if the code did change SIZE%, even if it was a number (not a variable) in the call - TURBO would just throw away the changed result, after using it like any other LOCAL, when the routine finished. The same goes for other types of value.

TURBO lets you pass one-dimensional character arrays, or 'simple strings', by value OR reference - reference is quickest if the string is a long one. If a string is passed by value, but the procedure or function, for some strange reason, tries to change it, you should note that you cannot store more characters in such a parameter than were passed originally - TURBO just chops off redundant characters at the end, to make the string fit. You may get an extra character, since string limits are always rounded up to an even length so that 16-bit code can work efficiently.

It doesn't matter what the names of actual reference parameters are, as long as the variables correspond in type and number of dimensions to the formal parameters. If actual and formal parameter types don't match, coercion is automatically done to suit the code in the definition.

REFERENCE directives are ignored by the interpreter, which should still be able to handle the calls happily, albeit at its usual slow pace. BEWARE: the interpreter may fail, in effect at random, if you use more than 9 locals or parameters in one definition. TURBO does not impose this restriction - you can have ANY number of locals or parameters in a compiled task.

You can have as many REFERENCE lines as you like before a DEFinition, but you shouldn't put them inside a definition or TURBO will assume that you've finished that routine and forgotten the END DEFine; the REFERENCE lines will refer to the next definition.

It doesn't matter what order you put names in a REFERENCE line. If any are left 'unused' after a definition has been read, TURBO will warn you about them before forgetting about them.

If you put the same name more than once in a REFERENCE line this has no effect. Repetitions are simply ignored.

GLOBAL and EXTERNAL parameters

Parameters of GLOBAL and EXTERNAL routines are usually passed by reference.

You can tell the compiler to pass certain scalar parameters by value. If so, you must put the formal parameter names in round brackets (so they do not appear to be 'names') in the EXTERNAL and GLOBAL statements. You don't need a REFERENCE command before the DEFinition of a global routine, as REFERENCE is assumed.

For example:

```
100 EXTERNAL 1,PROCEDURE,ADD(x,(y))
110 k = EDITF(100)
120 ADD k,1
130 PRINT "Incremented value = ";k
```

would print one more than the number entered at line 110, if the following GLOBAL is declared in module 1:

```
100 GLOBAL 1,PROCEDURE,BUMP(number,(increment))
110 DEFine PROCedure BUMP
120 number = number + increment
130 END DEFine BUMP
```

You get the usual error message when you LINK LOAD routines with incompatible value and reference parameter declarations:

**** MISMATCH BETWEEN MODULES x & y AT ITEM n

The item and module numbers tell you exactly where to look when you want to sort out the error.

You'll find more details of GLOBAL and EXTERNAL directives in Chapter R.

Parameters and 'extensions'

TURBO versions up to 4.20

The paragraphs below apply to versions of TURBO earlier than 4.21. A description of the changed situation for versions 4.21 and later comes after that.

TURBO passes parameter information to resident procedures by value - not by reference. The interpreter uses the same ad-hoc rule as it does for SuperBASIC routines, so that changes to parameter values sometimes persist after the machine-code has finished, depending upon the exact format of the call.

TURBO gets lots of extra speed, compared with the interpreter, because it keeps values and the details of variables in it's own special form, rather than in the jumbled 'heap' (a technical term!) of eight-byte entries which the interpreter uses. It can't let resident procedures or functions change values inside a task, as they don't know the correct format - they expect the interpreter's scheme.

All machine-code 'extensions' check their parameters internally, by looking at a table which both the interpreter and TURBO can produce to order. The REFERENCE trick that TURBO uses for SuperBASIC would not be appropriate in this case, as extensions don't have to be DEFined in a BASIC program.

In any case the new code required to convert and re-convert formats would usually be wasted, as very few extensions alter their parameters. None of the standard QL routines used by TURBO, or the TURBO TOOLKIT routines, alter their parameters.

The only common extension that DOES alter its parameters is the GET command, included in Toolkit II. This can be replaced in a program to be compiled by TURBO by use of the TURBO TOOLKIT three FUNCTIONS, GET%, GET$ and GETF to read binary values from a channel. This shows a GET command and the equivalent TOOLKIT code:

```
GET #3,A$,B%
A$=GET$(#3): B%=GET%(#3)
```

The first might fail unpredictably if used in a TURBO task, because the code of GET would probably scribble all over TURBO's carefully set-up data-areas in a vain attempt to find and alter A$ and B% !

The POSITION and SET_POSITION instructions can be used instead of GET if you need to perform binary random access.

However, you could still use GET to position a file. Thus

```
GET #3\42
```

will work just as well as

```
SET_POSITION #3,42
```

Array-handling extensions can be written in compiled machine-code, instead of as calls to external code. This means that TURBO's efficient storage-scheme is used, and often leads to extra speed, especially if you use REMark + around code that copies values from place to place, so that TURBO recognises that the code must be extra speedy.

TURBO versions 4.21 and later

Various beta test versions of TURBO were issued with version numbers such as 4k21. All were designed to pass parameters by reference rather than by value.

The first definitive version with the advanced facility was 5.01. This, with one exception, passes all parameters by reference, whether simple variables or arrays. The exception is a one dimensional string. This is passed by value.

For programs running under SMSQ/E the above exception is not needed. Programs including the command "TURBO_ref" will, when compiled, pass simple string parameters by reference.

Passing simple integer or floating point variables to machine code extensions by value requires finding that value, putting it to some location and then setting a pointer to that location. TURBO now sets a pointer to the variable itself. This seems less cumbersome as well as allowing machine code extensions, such as GET, to change TURBO's variables.

The items in arrays are stored in exactly the same way by both the interpreter and TURBO. Also both the interpreter and TURBO use a similar block of information, called a descriptor, to hold information indicating just how the items are set up as arrays. For example, two of the ways that a group of twenty integers could be set up are as a one dimensional array of twenty items, or a two dimensional array of four blocks of five items. It is the descriptor which tells you how the array is made up. However, the form of the interpreter's descriptors is very different from the form of TURBO's.

In order to pass an array parameter TURBO now sets up a descriptor in the interpreter's format in some location and sets the pointer pointing to that location. By this simple means arrays can now be passed to machine code extensions.

A consequence of this is that it is now also possible to pass slices of arrays to machine code extensions. This is because the interpreter's descriptor was devised so that it could indicate just what the slices were without in any way disturbing the set of items themselves. All that TURBO

needed to do was to use the slicing information to set the appropriate values in the interpreter's descriptor set up by TURBO.

As indicated earlier, an exception to this is a simple string. In S*BASIC such a string can be either dimensioned or not. When an undimensioned string parameter is returned by a machine code function or procedure, space has to found if its size is bigger than the original. This is not acceptable to TURBO. However, if the string to be returned is dimensioned, no extra space is needed. If the returned parameter is bigger than the dimensioned space the extra bytes are simply ignored. This is acceptable to TURBO. Unfortunately, however, when a string parameter is accessed by the QDOS vector CA.GTSTR, space is allocated in the variables area which is a procedure that cannot be accommodated by TURBO. Luckily, the SMSQ/E vector, with different code, does not allocate such space and TURBO can cope.

The standard TURBO, from v5.01 onwards, passes all simple string parameters by value to machine code routines. Such compiled programs can be run on all QL type machines, but of course they do not allow the return of string parameters.

But the presence of the command "TURBO_ref" in a program to be compiled will cause TURBO to arrange to send all simple string parameters by reference. Since this will work only if the operating system is SMSQ/E such compiled programs will halt immediately with the message "not implemented" unless the first long word of system variables is "SMSQ".

CHAPTER Q - EXTENSION PROCEDURES AND FUNCTIONS

COMPILING SUPERBASIC EXTENSIONS

SuperBASIC is designed to be an extensible language - in other words 'extensions', or 'resident' procedures and functions can be added to the language easily, using a standard machine-code format. Most such add-on procedures and functions may be used in compiled programs.

All the commands and functions documented in the QL User Guide, and all the TURBO TOOLKIT commands, are considered as 'standard' by TURBO, even though most of them are implemented in this 'ad hoc' way. They are compiled - or rejected, in the case of editing commands like MERGE -quite happily.

Everything else - plug-in ROMs, disk utilities, and other 'toolkits' -is considered 'non standard' and treated consistently. The vast majority of these commands work perfectly with TURBO.

Problems only crop up with routines that assume that they are being called from task 0,0 (SuperBASIC) and those which rely on the presence of interpreter data-structures, which TURBO does not maintain for obvious reasons, though this latter problem disappears with v4.21.

Extension linking

Before program execution starts a compiled program checks that all of the commands it needs are loaded; if this is not the case an error message like this is produced:

PEEK_F IS NOT LOADED.

for each missing command. This message would be produced by TURBO if you tried to use the compiler when a very early version of TURBO TOOLKIT was loaded. A long list of missing names will appear if you try to use the compiler without loading the toolkit at all!

You must load the commands into SuperBASIC before the compiled program will run. In general you must type NEW and then re-load any SuperBASIC that was present before the commands were loaded - command references in the program will have been incorrectly tokenised if the relevant commands were not installed when you loaded the SuperBASIC.

You should always load extensions into space reserved by the SuperBASIC interpreter - NOT by a compiled task - as compiled code may terminate, leaving the extensions in limbo, and to ensure that commands are properly linked into the interpreter. All attempts to look up extensions are made by interrogating task 0,0, not compiled tasks.

TURBO does not insist that add-on code is loaded into any particular area of the QL's memory, but only that the start of the code is CALLed from SuperBASIC - task 0,0 - after loading. As a compiled program begins to execute it searches out appropriate routines by name. They don't need to be in the same place from one run to the next - they don't even have to have been loaded in a consistent order.

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 TURBO 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. This can be prevented by 'including' the command you want in the compiled program in the way described below.

Extension inclusion

There is a second way of allowing TURBO to use extensions even though these have not been loaded at the time the compiled program is being run. This is achieved by including the extensions file, the one which would otherwise be LRESPRd to add the names to those already in SuperBASIC, in the compiled program. The rules governing this are given now.

To include, for example, the extension file "flp1_my_ext" in a program type

```
REMark %%flp1_my_file,0,10
```

anywhere in the program.

The two percentage signs signal to Parser that an extension file is to be included. The two numbers following the name (0 and 10) are

the offset to initialising code, or 0 if none

the offset to the set of definitions of procs/fns.

You will see from this that you need a knowledge of the internal make up of the extension file you want loaded. Unless the two numbers following the name are exactly right the compiled program will almost certainly crash the machine when it is run.

Up to eight extension files can be included in this way. One other restriction is that the total length of all the extension files must be less than 32764 - 6 * the number of files. ** NOTE that this restriction on length was removed from Turbo v4.17 onwards.

If you do include files using this facility you must make sure that the TURBO buffer, which you can set from TURBO's front panel, is at least big enough to contain the largest single extension file to be included. ** NOTE that you do not need a TURBO buffer bigger than 33K from v4.20 onwards.

It is also necesssary that these extensions be loaded when the program is being compiled.

TURBO emulation in brief

When TURBO needs to call a non-standard routine it finds the parameter values as it would to call one of its own routines, 'pretends' that the interpreter is running, and then calls the external code of the non-standard routine. The accuracy of this pretence is obviously limited - this is necessarily the case, or TURBO would have to work just like the interpreter and no advantage would be gained from compilation.

TURBO cannot speed up the actual execution of the add-on routine - the same code is used whether the command is interpreted or compiled - but TURBO does speed up the processing of parameters and expressions, which regularly takes the interpreter longer than command execution in any case.

The biggest benefit is that you can use virtually all the commands that you're familiar with, in a framework many times faster than the SuperBASIC interpreter. The rest of the language - SuperBASIC procedures, functions, arithmetic, loops, string handling, and parameter passing are handled very fast indeed by TURBO.

Unless some terrible mistake has been made in the coding of the routine, any number of TURBO tasks can share a single set of extensions. After a year selling SUPERCHARGE, which worked in a similar way, we know of no commands that fail if several tasks use them at the same time.

Special cases in optimised extensions

TURBO can speed up many commands and functions which it knows about. There are a few idiosyncracies in these routines which must be faithfully reproduced, in order to maintain exact compatibility:

The CHR$ function reduces its integer parameter MOD 256 if it is not a valid ASCII code, rather than give an error message. That has to be done to give the same results as the interpreter.

POKE and POKE_W work similarly, but reduce values MOD 256 or 65536 even if they are up to 32 bits long (about +/- 2E9). This costs a little time but it is essential to keep compatibility with the interpreter -where the distinctions between integers, long words and floating-point values are often rather hazy!

RESPR in a compiled program allocates memory from the system 'heap', outside the task at the bottom of memory, rather than from the area used by SuperBASIC, which is frozen while any task is running! RESPR therefore works like ALLOCATION, in a compiled task. SuperBASIC RESPR rounds allocations up to the next multiple of 512 bytes, whereas TURBO RESPR works with finer 8-byte chunks of memory.

Possible pitfalls and solutions

TURBO cannot check the number or type of parameters passed to non-standard routines when a program is compiled. All it can do is extract the specified values from the compiled task's variable area, indicate their types, and hope for the best. If you type:

```
RENAME 12,PI
```

rather than:

```
RENAME "FILE A","FILE B"
```

You will only discover the error when you come to run the compiled code, since TURBO doesn't know that the command should have two string parameters.

There again, if you'd typed:

```
REMANE "TOTALLY","SILENT"
```

the compiler would spot the mistake at once, since REMANE is not a valid SuperBASIC extension - at least, not as far as we know.

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 TURBO cannot tell whether you are referring 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 or apostrophes. For example:

```
WCOPY turbo,ser
```

would have to be changed to read:

```
WCOPY "turbo","ser"
```

Another potential problem in TURBO v4.20 and earlier was caused by commands that try to modify the variables passed to them as parameters. Luckily such commands are rare - the main one in this area is the GET command of Toolkit II.

There's no need to use GET in TURBO programs, as TURBO works happily if values are returned by functions, and three binary fetch functions equivalent to GET - GET%, GETF and GET$ - are included in the TURBO TOOLKIT. You can use SET_POSITION or POSITION to find your way around a binary file.

The problem is this: TURBO uses a much more efficient variable-handling scheme than the interpreter, but that means that parameters passed to extensions must be kept 'at arms length' from TURBO's own data-store. If a procedure changes the value of a TURBO parameter TURBO will - at best - ignore the change; at worst the procedure could end up zapping an important part of the compiled task, or trying to 'make room' for data as if the interpreter were being used. Under such nasty circumstances just about anything could happen!

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

TURBO can't be exactly compatible with both of these rules, especially as there are about a dozen different QL ROMs in circulation. But TURBO is a professional tool for software developers, and a program compiled on one version must run, without problems or changes, on other QL models.

So long as you resist the temptation to load more than one copy of a given command, all will be well.

Summary

For versions of TURBO earlier than 4.21 machine 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.

For TURBO v4.21 and later none of these restrictions apply except for the reliance on the name list.

TECHNICAL DETAILS OF TURBO EXTENSION CALLS

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 TURBO. In general, procedures and functions which do not rely on the presence of SuperBASIC interpreter data-structures will work perfectly.

Parameter passing

TURBO 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 unless the parameter was passed by reference - the usual case with TURBO v4.21 and later - 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 TURBO 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, (length+11) bytes for each string parameter passed by value and (14 + 4 times the number of dimensions) bytes for each array.

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) - by the code shown below. The function FREE_MEMORY uses this code to determine whether it should return the size of the task stack or that of the QDOS 'free' area.

This 68008 assembly language subroutine, CHECK.D1, checks for D1.L bytes on the maths stack; it works perfectly in interpreted or compiled code, whether SUPERCHARGE or TURBO is used, jumping to NO.ROOM. If there is not enough room when a compiled task is running you should return at once, with the error code -3 (out of memory) in register D0:

```
CHECK.D1 TST.L   $54(A6)
         BEQ.S   COMPILED          For Qliberator
         TST.B   $54(A6)
         BPL.S   OK                Not compiled
         CMPI.L  #'SBAS',-4(A6)    Daughter basic? . .
         BNE.S   COMPILED          . . no
         MOVEQ   #0,D0
         TRAP    #1
         CMPI.L  #'SMSQ',(A0)
         BNE.S   COMPILED
         CMPI.L  #'2.00'
         BLT.S   COMPILED
OK       MOVE.L  A1,$58(A6)        Set BV.RIP
         MOVE.W  $11A,A2           Call BV.CHRIX
         JSR     (A2)
         MOVE.L  $58(A6),A1        Retrieve BV.RIP
         RTS

COMPILED ADD.L   (A6),D1           Work backwards!
         CMPA.L  D1,A1
         BCS     NO.ROOM
         RTS
```

The way that BUFFER is set up can cause problems for the SuperBASIC COPY routines in the QL ROM. COPY, and similar commands, uses the entire BUFFER area as a temporary store as it works. Unfortunately it makes no check on the size of BUFFER and passes that size directly to the 'read and write string' routines in the ROM, which do not expect a buffer size in excess of 32K.

COPY consequently gets steadily faster as the free space in a compiled task is increased up to 32K, but behaves very strangely thereafter! If you need to use COPY in programs where more than 32K may be free you should dimension (and then de-allocate) a large array to 'soak up' the spare memory before calling COPY. This routine should do the trick for any dataspace up to 224K:

```
DEFine PROCedure SAFE_COPY(in$,out$)
  IF FREE_MEMORY>32767
    DIM dummy((FREE_MEMORY-32700)/6)
    COPY in$ TO out$
    DIM dummy(0)
  ELSE
    COPY in$ TO out$
  END IF
END DEFine
```

Believe it or not, the size of buffer used in an interpreted SuperBASIC program depends upon the length of the longest line entered since the machine was turned on!

The User (A7) stack

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 upon extensions by the SuperBASIC interpreter.

Basic variables

A set of pseudo-basic-variables is provided for the use of procedures and functions; these are addressed by A6, as usual. Some of these basic 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 32 standard 40-byte entries) is maintained within compiled tasks. The pseudo-basic-variable BV.CHBAS points to the base of this table. The byte at offset 17 within each channel entry is reserved by TURBO - it is used to handle the '!' separator.

If TURBO detects any attempt to change the size of the channel table within an extension it stops with a 'Channel not open' error.

Why TURBO does not create resident procedures

TURBO is designed to compile stand-alone tasks, but not individual procedures, LINK_LOAD notwithstanding.

TURBO does not work like the interpreter or like a SuperBASIC extension, and with good reason. Single extensions which were originally written in SuperBASIC and compiled would be large and slow. Programs compiled in one step, or composites created with LINK_LOAD, can be much more efficient because they do not need to work in a manner compatible with the interpreter, except when they call standard resident procedures. Remember that these calls are not the area in which TURBO offers a major speed-up.

If you want machine-code speed from your SuperBASIC you should compile your whole program. It is sure to load and run much faster than any hybrid routine could.

SECTION 4 - ADVANCED CONCEPTS

29th May 2006

CHAPTER R - LINKING AND COMMUNICATION BETWEEN TASKS

COMMUNICATION BETWEEN TASKS

This chapter explains how TURBO can support the concepts known as 'task communication', separate compilation of 'modules' and 'shared variables'. These terms are defined as they are explained. The first part of the chapter deals with the communications facilities provided by EXECUTE; the second part deals with LINK_LOAD.

'EXECUTE' PARAMETERS AND PIPES

The TURBO TOOLKIT command EXECUTE allows a parameter, or 'option string', to be passed to any task as it is loaded. When the task is running it can read the string any number of times with the new OPTION_CMD$ function.

This string may contain any information that you care to pack into it, including an arbitrary number of parameters which you can separate in any manner you see fit (e.g. with commas, or CHR$(10), end of line markers).

Several compiled tasks and option strings may be specified in one EXECUTE command; EXECUTE also works from within a compiled program.

If you specify several tasks, 'pipes' - temporary files in memory, which can be read and written to simultaneously - are set up to allow the tasks to communicate via PRINT and INPUT:

```
EXECUTE_A   BAD TO WORSE
```

In this case a pipe will be set up, linking the two tasks. Whatever the first task PRINTs on channel 31 the second will be able to read from channel 30. You can make either of these the default channel, in a compiled task, using CHANNEL_ID and SET_CHANNEL. This line routes PRINT statements with no explicit channel number, to the output pipe:

```
CLOSE  #1: SET_CHANNEL  #1,CHANNEL_ID(#31)
```

DON'T try this in SuperBASIC - Sinclair's interpreter can't cope if its default channels (0, 1 and 2) are re-assigned. You can't EXECUTE the interpreter anyway, as it is loaded from the moment the QL starts up!

Each pipe is normally 200 characters long. Long lines can be read piecemeal with INKEY$ or - more efficiently - with the TURBO TOOLKIT function INPUT$.

If a pipe becomes 'full' a task that tries to write characters into it will pause until there is room for the new text. Room is created when the task at the other 'end' of the pipe removes some characters.

If a task tries to read data from a pipe when there is nothing to be read it will wait until appropriate characters arrive. An end of file condition, which you can detect with SuperBASIC's EOF function, exists when all the data has been read and the task putting data into the pipe has finished, or closed the output channel to the pipe.

Pipe linkages can exist between any number of tasks, each of which can send messages to the one after it in the chain, and receive messages from the previous one. Pipes work very quickly.

Data can go to or from other devices and channels, as well as tasks. An EXECUTE parameter list can start with a SuperBASIC or compiled program channel number (prefixed with a hash character) or a data file or device name, from which data will be read. Likewise, the list may end with an output channel number or the name of a file or device to which information will be sent.

This lets you link a compiled task to the interpreter, or to a file, with no need to 'tell' the task what device to expect.

EXECUTE, LINK_LOAD, and pipes are also discussed in the TURBO TOOLKIT manual.

LINK_LOAD

THE PURPOSE OF LINK LOADING

It would be useful if programmers could split program files into sections, or 'modules'. This would allow them to keep logically distinct modules in separate files. This makes programs more readable and easier to edit.

Programming ergonomics are especially improved with QL SuperBASIC, as the interpreter is very slow when loading and tokenising large programs, yetÿprograms must be tokenised before TURBO can analyse them. Users can save compilation and loading time by re-compiling only the part or parts of a program where changes have actually been made.

It would also be useful - and a rare treat - if several tasks, running concurrently, could access 'shared variables' - variables, of any type, in other tasks. This gives extra power to the programmer: logically distinct, logically parallel processes can have entirely separate code but share access to 'environmental' variables, allowing them to interact.

For example, a racing simulation program might consist of several tasks, each simulating the performance of a vehicle and accessing a shared data-structure representing a 'map of the track' - another entirely separate but concurrent task could read the map and produce graphics showing the position of each vehicle in real-time.

You can then try different vehicle-models in isolation or in arbitrary combination, simply by linking different modules. There's no need to re-compile anything - indeed there's virtually no overhead associated with TURBO linking - shared variables are accessed as fast, or almost as fast, as if they were in the same task.

This scheme provides a natural way of implementing any system which incorporates parallelism; booking systems, resource allocation simulations, games simulating multi-player activity, and even program development systems.

A program editor and compiler could be linked to allow fairly interactive development of software; the compiler would be invoked by the editor after each change to a module and the results tested by linking that module to previously-compiled parts. This is one direction which the TURBO development might take.

The sky's the limit!

Conventional linking strategies

TURBO linking is revolutionary, for all that it has been implemented first on a cheap, mass-market computer. It has many differences from the crude linking which some other compilers offer. It is worth reviewing those systems here, briefly, to explain the weaknesses which TURBO linking can side-step.

Conventional programming languages produce output in an 'intermediate' form when linking is required. You can't execute this output directly -in fact just about the only thing you CAN do with it is link it!

A separate program, often larger, slower and more complex than the original compiler, is used to combine several intermediate files into one executable task. This approach can produce very efficient code, as the linker has a complete understanding of the interaction of all the static datastructures in the task, but few 'real' linkers can produce better code than a stand-alone compiler.

Conventional linkers allow variables and routines to be declared outside the files in which they are used, but they rarely let you change the size of data-items as a program runs. Nor do they allow several concurrent tasks to share code.

TURBO linking

TURBO implements an efficient form of linking that has all the features conventional linkers lack, yet is easy to use.

Linking is all about making names - procedure, function and variable names - correspond between two or more separately compiled files containing complete procedures or functions. These files are called modules.

TURBO can link an arbitrary number of modules as they are loaded -there is no need for a separate 'linking' program or process, and loading is performed at the usual speed.

In each module you must indicate the correspondence between 'dummy' or 'formal' declarations within the module, and 'actual' declarations outside. This turns out to be quite easy. All external variables, procedures or functions are listed in special directive statements in the modules that contain or intend to use them.

For instance, if a module wanted to access an external two-dimensional array and an integer function expecting a string argument, this directive would give the compiler enough information to be able to generate code to handle the items:

```
EXTERNAL table(0,0),size%(text$)
```

In fact the TURBO directive format is slightly more complicated than this, for reasons which we will explore in the following tutorial.

How TURBO works out the meaning of names

Notice that TURBO can tell functions and arrays apart, in the last example, because arrays have dummy numeric subscripts, whereas functions always have names as arguments. The compiler only needs to know the dimensionality of arrays, and not the actual bounds; TURBO always calculates and checks these at run-time.

Array parameters appear just like other array declarations, with dummy subscripts, but within a parameter list; SuperBASIC does not support Pascal-like procedure or function parameters, so there's no need for TURBO_P and TURBO_F directives inside a parameter list, although it would be easy enough to support them in a future release.

But two confusing cases still exist - where parameters are present, it is not possible to distinguish between functions and procedures; and where there are no parameters it is not possible to distinguish scalar (un-subscripted) variables from parameter-less procedures and functions.

There's a simple answer - you must mark procedures and functions explicitly, by writing a special pre-defined identifier before them in the list:

```
EXTERNAL table(0,0),TURBO_F,size%(text$)
```

The TURBO_F identifier only refers to the next entry in the list. It is a pity that there must be a comma between it and the next name, but the interpreter doesn't allow two names to follow one another without some sort of separator.

For the sake of consistency, ALL procedures and functions must be marked in this way.

TURBO_F and a similar directive TURBO_P are implemented as functions, so that the interpreter lets them appear in a EXTERNAL parameter list. The values they return are not important.

For that EXTERNAL statement to work, another module loaded at the same time must contain a GLOBAL statement with matching types:

```
GLOBAL array(0,0),TURBO_F,bounds%(text$)
```

Notice that names don't have to match, unless you want to test the modules by MERGEing them under the interpreter. Types, number of entries and dimensionality must correspond exactly. If they don't, TURBO will (probably) complain when the tasks are LINK_LOADed - this checking is less than perfect, but it picks up most errors; any others will rapidly make themselves obvious!

The best way to make sure that these declarations match is to edit GLOBAL statements from EXTERNAL ones, or vice versa. There's no limit on the number of GLOBAL or EXTERNAL statements in a module.

As explained so far, only one module may contain GLOBAL names; the others must either declare all of those names as EXTERNAL, or declare none of them, in which case they will run concurrently with other modules but in isolation. This is not good enough.

The scheme would be more powerful if every module could have its own set of GLOBAL names, and every other module could chose whether or not to use external variables from each other task. This is easily done by adding a 'module number' at the start of each GLOBAL or EXTERNAL directive. For obvious reasons, this number must be an explicit integer constant.

With this extra feature the inter-relationship of modules may become arbitrarily complex; any number of modules can access data GLOBAL to any other module. Thus modules can be viewed as queues, nets, hierarchies, or combinations of these structures.

Module numbers don't have to be contiguous - there can be unused numbers - but only one module can have a given number. If module numbers are contiguous there will be the minimum memory overhead when the modules are loaded, but the overhead is small anyway; just 4 * (2 + Highest Module No.) * Actual No. of Modules. You should avoid using needlessly high numbers - values between 0 and 15 should be fine for most purposes.

Module 'group names'

Sometimes you may find it annoying to have to specify all of the global variables in another module just because you want external access to some of them. Of course, you don't have to use an EXTERNAL variable or routine if you don't want to, even if you declare it. Nonetheless, it is useful to be able to divide GLOBALs and corresponding EXTERNALs into several groups.

You can do this by declaring each group on one or more lines of its own, and following the module numbers with a literal string: the name of the group. The names must correspond exactly, between GLOBAL and matching EXTERNAL statements. Other modules can then refer to one or more groups by name in matching EXTERNAL statements.

Assuming these lines appear in module 1:

```
GLOBAL 1,"graphics",x,y,size
GLOBAL 1,"sound",duration,pitch
```

Other modules could contain either or both of these lines:

```
EXTERNAL 1,"graphics",w,h,max
EXTERNAL 1,"sound",length,note
```

You don't have to use group names if you don't need them, but they can be very useful in large projects consisting of three or more modules.

Declaration rules

In order to keep compilation fast and efficient, declaration details are kept in a single block, generated at the end of pass 1, just after DATA. For this reason GLOBAL and EXTERNAL statements must be at the end of the module source - after all DATA, DIMs and DEFs.

GLOBAL declarations must come before EXTERNAL ones. No name can be both EXTERNAL and GLOBAL - this would make no sense at all! Declarations may be split over several lines, but all the declarations for a given module number and group name (if any) must be contiguous.

The 'scope' of EXTERNAL names

While an EXTERNAL routine is executed, the only variables and routines it can use are ones that exist IN THAT MODULE. You must use the names that module recognises. These variables and routines include any parameters passed when the module was called, and anything declared as EXTERNAL in that module.

Arrays can only be re-dimensioned by the module in which they are GLOBAL.

Module loading

You load and execute compiled modules by specifying them all together in a LINK_LOAD command:

```
LINK_LOAD   FIRE,AIR,WATER,EARTH
```

The associated commands LINK_LOAD_A and LINK_LOAD_W correspond to variations of EXECUTE. The default device name is automatically tacked onto the start of task names, unless another device is explicitly specified.

You can set task priorities by typing an exclamation mark and a value (1 to 127) after each name. 'Option strings' are not allowed - TURBO uses the option string to pass information between linked tasks.

Each task invoked by LINK_LOAD must have a different module number, set with the GLOBAL command. If several tasks have the same number the error message 'Already exists' will appear, and none of the tasks will run.

If a task is missing, so that other tasks cannot find the data they expect, the error is 'Not found'. The report 'Bad parameter' indicates that one or more name did not correspond to a task suitable for LINK_LOADing.

The first task in the list supplied to LINK_LOAD 'owns' all the others; they will terminate automatically if the first task stops or is removed.

When the tasks load they communicate very quickly to make sure that declarations correspond. If inconsistencies are found there is a 'linking error' and all of the tasks are aborted.

An error message is displayed, showing the linking and target module numbers (J and K) and the sequence number (N) of the name that did not

match - the first name in the global list is number 1, and so on:

Mismatch between modules j & k at item n

ALL the modules stop at once, releasing memory and closing channels, if such an error occurs.

DETAILS of shared access to external variables

It is not acceptable for two tasks to try to write to variables simultaneously; nor should one task be allowed to read data while another is in the process of updating it.

Multi-tasking is momentarily disabled during all writing to global or external variables, including redimensioning, which can only be performed by the task in which the variable is global. This makes multi-tasking a little more 'grainy', but the most complex operation that will be performed in one step is the assignment of a 32K string, and TURBO can do that in less than a tenth of a second, even in sluggish internal memory.

In every other respect - including the reading of values, which is generally more common than writing - external variables are handled as if they were any other variable in the task which is manipulating them.

DETAILS of shared access to external routines

Before any compiled procedure or function is called, the values or addresses of parameters are created in the dataspace of the task containing the called routine - the 'host' - not the caller. To avoid chaos, and the need for complicated and slow 'policing' checks, only one call to a given module can be in process at any time.

If this rule creates a 'bottleneck', as modules wait for one another, you can always create multiple copies of routines for which there is contention, and link different callers to different copies.

In fact you'll probably only need to do this if called routines take a long time to run, yet the callers must appear to work concurrently. While a module waits it takes virtually no processing time, so the total amount of work that the computer can do in a given time is rarely reduced by the use of external calls.

The rule about only one part of a module running at any time also applies to the host task, which is, of course, perfectly entitled to run its own routines! If other modules are to get a look-in, the host must stop running itself. It does this by executing a special TOOLKIT command, SNOOZE.

Normally a linked task will set up GLOBAL arrays and other variables, if any, and then SNOOZE so that others can gain access to its code.

Once a module has 'gone to sleep' it cannot re-start, because it has no way of knowing whether or not other modules are using its code. Such a signal would slow down calls appreciably. In fact this doesn't really matter, because a module takes no processing time once it has fallen asleep.

We probably COULD implement something akin to the kiss of a prince (!), to bring a slumbering module back to life, but this is not likely to be useful in most cases. Remember that tasks do not have to SNOOZE to share variables, but only to share procedures or functions.

There is one potential source of deadlock - the scourge of multi-tasking systems. In theory processing could get stuck, with several modules waiting indefinitely for one another, if one module calls a routine in another, which then tries to call the first... which is busy... so the second task waits... so the first can't complete its call, and stays busy... so the second task keeps waiting... etc... etc... !

It seems perfectly reasonable to put the onus on YOU to guard against this sort of thing. If this risk really worries you, you can be sure of avoiding it if none of your modules contain both GLOBAL ** AND ** EXTERNAL calls. Keep all the EXTERNAL declarations in 'master' modules and only put GLOBAL declarations in the subsidiary modules that are called. This restriction is not necessary if you design your system carefully on paper before you code it, but if it helps you sleep at night, so much the better!

Assuming the caller does not have to wait for another task - which will generally be the case, as few users will need true parallel processing - the overhead of an external call compared with an internal one is only a few microseconds.

Channels and modules

All channels are 'privately owned' by each module. Code in a module can only use channels which have been opened implicitly or explicitly by that module, or channels which have been set up with SET_CHANNEL, using a CHANNEL_ID passed from another module.

In general you should compile modules with the 'copy windows' control set to zero, as few modules will need their own windows. If your application does demand that every module has its own set of windows, TURBO will cope quite happily.

In theory programs, tasks and modules running on a QL can open up to a total of 360 channels, which should be more than enough for any practical application! The limit is a generous 168 channels on a standard 128K QL.

Two modules may independently open and read the same disk or microdrive file, but only one channel may be open to a file which is being written. If other modules try to open such a file an 'IN USE' error results. DEVICE_STATUS can detect this error condition.

Testing modular programs

It is easy to test modules under the interpreter, even though each file is 'incomplete', because you can MERGE all of the files into one big program, as long as names correspond exactly and you choose line-numbers sensibly. Line numbers are no problem if the modules started out as one big program; if not, you can RENUMber the parts as you first MERGE them together.

EXTERNAL statements are ignored by the interpreter if they only have variable-names among their parameters, but SuperBASIC hickups if it comes across a procedure or function name used as a parameter. It's best to keep EXTERNAL and GLOBAL declarations out of harm's way; TURBO imposes various placement rules, explained later, in the interests of fast compilation - in general it makes sense to put declarations at the

end of each file.

MERGEing is no slower than LOADing. Admittedly this is not very fast, and it might be more accurate to say that LOADing is no faster than MERGEing - but the fact remains that you can test modular programs almost as easily as you can test single-file, 'monolithic' ones. It is easy to build the required MERGE statements into your program - TURBO will ignore them.

Splitting monolithic programs

Given all the advantages of separate compilation, it is quite likely that you'll want to split up some of your existing programs so that you can adapt and compile them piecemeal.

The best way to split files depends upon the way that your programs are organised. First and foremost, you should try to ensure that procedures which call one another in loops, or very often, are in the same file. This gives a small speed improvement, because processing doesn't keep switching from one task to another, but the difference is quite small.

A better reason for this kind of split is that it increases the likelihood that changes will be confined to a certain module, so you won't have to re-compile several parts whenever you make a design change. This kind of split also tends to reduce the need for GLOBAL and EXTERNAL declarations.

If the routines you want to put in a module are grouped together you can extract them easily with SAVE, which allows 'line-slices' as parameters. For instance:

```
SAVE FLP1_MODULE1_BAS, 100 TO 420, 700 TO 1300
```

will create a file containing two sections from the current program.

If the routines you want to put in a module are muddled with others you should use the TURBO LIBRARY MANAGER to extract routines; all you need is a list of the procedures or functions you want to extract - give this list to the manager and let it get on with it!

You can also use the utility program MAKE MODULES to split programs -this program, and other techniques, are discussed in Chapter G, which deals with 'Real-world problems'.

CATNAP - THE SLEEPING PRINCESS COMMAND

Chas Dillon has designed a small extension to the LINK LOAD scheme which can be very useful if you're an imaginative programmer or you're working on a complex concurrent system.

SNOOZE stops a task dead, so that other tasks can have their wicked way with its routines. The task never comes 'back to life', although its code may be used many times by several tasks.

CATNAP is a kind of 'light snooze'! It puts a task to sleep, but the task re-starts, from the statement after CATNAP, whenever another task has called and exited from one of its routines. Thus every access from another task causes a task which is using CATNAP to wake up, to poll variables and maybe to take action consequent upon the call from outside.

When the task has finished it can SNOOZE, allowing other tasks in 'for ever', or just CATNAP again, to wait for the next call.

It therefore follows that this line:

```
REPeat simulated snooze:CATNAP
```

works just like a SNOOZE, except that the task it appears inside wakes briefly (and fruitlessly) after every call from outside, wasting a little processing time. There's no real point in using such a line, but CATNAP can be very useful in a loop. A simple example follows:

You can make a task count the number of times it is called by replacing SNOOZE with this:

```
count = 0
REPeat counting
  CATNAP
  count = count + 1
  PRINT counting,
END REPeat counting
```

This causes the latest access-count to be printed to the module's default window (make sure it has one!) whenever a GLOBAL routine is called from outside.

Many more complicated uses of CATNAP are possible, but these depend upon your system design and the imagination with which you use TURBO. LINK LOAD is an extremely powerful tool, and rewards study and experimentation.

SUMMARY OF GLOBAL AND EXTERNAL DIRECTIVE SYNTAX

This is a short but comprehensive list of points to bear in mind when using GLOBAL and EXTERNAL in your programs.

* GLOBAL and EXTERNAL directives must be at the start of the text of a program, perhaps with relevant IMPLICIT statements beforehand. GLOBALs are declared first and then EXTERNALs, organised in groups for each EXTERNAL module number.

* Both GLOBAL and EXTERNAL directives MUST be followed by a module number.

* All GLOBAL directives must use the same module number, which is the module number of this task.

* EXTERNAL directives may not use the same module number as a GLOBAL directive. They should use numbers which correspond to GLOBAL

directives in other modules which will be loaded with this one by LINK LOAD.

* After a module number you may put a 'group name' - a default null name "" is assumed if you do not do this. All declarations for a given group name must be collected together on consecutive lines, or a single line. The case of characters in a group name is significant - group names must match exactly between modules.

After the group name you may put any number of procedure, array or variable declarations, separated by commas. Arrays should have dummy subscripts to indicate their number of dimensions. The type of a variable or function is indicated by IMPLICIT declarations or the last character of the name.

* Procedure and function names in a directive list must be preceded by the word TURBO_F or TURBO_P, as appropriate, with a comma to separate the two names. Any spaces are ignored.

* All parameters of shared procedures and functions must be indicated within the brackets after the name of the routine. Arrays must be followed by dummy subscripts. All values are passed by reference, apart from scalar values which have their names in brackets. For example:

```
IMPLICIT% table
EXTERNAL 3,"group",TURBO_F,tot up(table(0,0),x,(y))
```

In this case TOT UP is a floating-point function, with appropriate code in module 3. TABLE is a two-dimensional integer array. X is passed by reference and Y by value (see Chapter P).

CHAPTER S - ERROR TRAPPING AND RECOVERY, DEBUG and TURBO_V

TURBO'S GET-YOU-HOME SERVICE

TURBO TOOLKIT includes a number of keywords associated with error-trapping. Some, like DATASPACE, ALLOCATION and DEVICE_STATUS, return a special value to signal an error without disrupting the program. Others, such as EDIT% and EDITF, refuse to allow incorrect entries. All those are documented in the TURBO TOOLKIT manual.

This chapter deals with a different kind of error-trapping. WHEN_ERROR, END_WHEN, RETRY_HERE, ERLIN% and ERNUM% are the TURBO TOOLKIT commands associated with the subtle art of postponing or preventing a task from stopping AFTER an error. The commands are activated by TURBO, and work in any compiled task. They do not work fully in interpreted code because few QL ROMs support the facilities they require.

Compiled WHEN_ERROR

TURBO WHEN_ERROR works on any version of the QL, in programs compiled on any version. The code inside a WHEN block should be skipped during interpretative testing - use IF COMPILED... or IF NOT COMPILED GO TO.

When an error occurs in a compiled program TURBO checks to see whether or not a WHEN_ERROR routine has been passed. If not, the error is reported normally.

There are two 'levels' of WHEN_ERROR routine supported at present. You can use one level to trap 'local' errors, such as invalid entries or coercion problems, and another to escape gracefully from more serious problems, such as OUT OF MEMORY. You don't end up with the unwieldy, monolithic, catch-all error-trapping clauses of Microsoft or DEC BASIC.

TURBO remembers the most recently encountered block at each level. One block at a certain level 'over-rides' the previous one at that level when program execution passes the point at which it is defined - rather like a DIM. If there's no error, execution skips over the whole block. The code in the WHEN block is ONLY executed after an error.

A WHEN block consists of the TURBO TOOLKIT WHEN_ERROR directive (one word with an underscore in it) followed by a digit - presently 0 or 1. Then come a succession of program lines which are only executed when an error occurs, and which will normally attempt to diagnose, report and perhaps recover from the error, and the directive END_WHEN (again one word).

If you use the partly implemented WHEN keywords on a JS or MG version of the QL, you can omit the underscore and type in the commands as two words - in this case, WHEN ERRor (two words) corresponds to WHEN_ERROR 0. Note that RETRY and CONTINUE do not work in the same way when compiled by TURBO as when handled by the interpreter - this is mainly because TURBO doesn't take a note of the current line, statement, and bounds of various tables after every statement, as the interpreter does, but only when you indicate that this would be a good place to do so, by including a RETRY_HERE directive.

From v5.05 there are two retry points, one for a RETRY within a WHEN_ERROR 0 clause and the other for a RETRY within WHEN_ERROR 1.

The RETRY_HERE directive can be used to mark the two separate places for return. RETRY_HERE 0 will set the return position for WHEN_ERROR 0 and RETRY_HERE 1 the position for WHEN_ERROR 1. RETRY_HERE without either 0 or 1 causes the same position to be set for both WHEN_ERROR 0 and WHEN_ERROR 1.

The TURBO TOOLKIT functions ERLIN% and ERNUM% can be used inside the block, to find the last line started before an error, and the internal number of the error. Values of ERNUM% and standard reports correspond as follows:

-1 not complete

-2 invalid job

-3 out of memory/dataspace full

-4 out of range

-5 buffer full

-6 channel not open

-7 not found

-8 already exists

-9 in use

-10 end of file

-11 drive full

-12 bad name

-13 Xmit error

-14 format failed

-15 bad parameter

-16 bad medium

-17 error in expression

-18 overflow

-19 not implemented

-20 read only

-21 bad line

-23 access denied

You can use the "JS" and "MG" ROM functions ERLIN and ERNUM, if you wish, but they are not as efficient as their TURBO counterparts, especially in compiled programs.

Some of these errors, like 'bad line', are extremely unlikely to crop up as a program runs - but they may all, potentially, be returned, perhaps by extensions.

The error numbers that TURBO itself returns, and the circumstances in which they are returned, are documented in great detail by Chapter F.

How an error is propagated

When an error occurs TURBO attempts to call the most recent WHEN_ERROR 1 routine that it has passed. If there is none it tries to call the most recent WHEN_ERROR 0 routine. If it still draws a blank it reports the error 'as normal'.

If the WHEN_ERROR 1 routine is completely executed (up to the END_WHEN) without RETRYing, or if it executes a compiled CONTINUE statement, or itself fails with an error, the WHEN_ERROR 0 routine is called. If this doesn't exist the latest error is reported normally.

If a WHEN_ERROR 0 routine executes a compiled CONTINUE statement, the last error is reported normally. The same happens if an error occurs in the level 0 error-handler.

It follows that, if you only want one level of trapping, it doesn't matter whether you use level 0 or level 1.

Either type of WHEN_ERROR routine may cause program execution to continue as if the error had not happened, if it executes a RETRY statement. Processing continues at the point after the last RETRY_HERE statement.

Thus you can trap errors in the whole of a routine, or several routines, by putting a RETRY_HERE statement at the start of each and -if the error line and code seem appropriate - performing a RETRY.

There is always an implied RETRY_HERE set at the start of every compiled program, so if you don't explicitly set your own RETRY_HERE a RETRY will take you back to the start of the program.

You can jump back into a program from within an error-handler without using RETRY. For instance you might use a procedure-call, or (horror of horrors!) a GO TO - but, if you do this, you should traverse another WHEN_ERROR soon after, or TURBO will not realise that you have left the handler, and subsequent errors will be propagated to the next level, or out to the user.

Locals, arrays, temporaries and RETRY_HERE

When an error is trapped it may be that temporary data has been allocated space by the compiled task. The task would rapidly run out of memory if this was not de-allocated.

Whenever TURBO encounters WHEN_ERROR - at either level - or RETRY_HERE it saves the current limits of storage used for temporary values, LOCAL information and arrays.

TURBO also records the position of the WHEN_ERROR or RETRY_HERE directive, which it needs to perform trapping or recovery correctly.

The saved storage limits are used by the error handler, so that it doesn't immediately fail, or leave space allocated after an error.

TURBO re-stores the limit of space used by arrays after a DIM, so TURBO RUBBER ARRAYS (Chapter K) work fine in programs that use WHEN_ERROR. If an error would cause an array to expand beyond available limits, error -3 is generated BEFORE any attempt to move or clear current array values!

LOCAL variables created between the time a RETRY_HERE was encountered and the incidence of an error are de-allocated.

In general, this is a good thing, and what you'd want to happen. But avoid using RETRY to jump BACK into a block you've RETurned from: all of the LOCALs will have vanished!

RETRY and recursion

It can be useful to use RETRY and RETRY_HERE in your programs, even if you have trapped normal errors explicitly with EDIT%, EDITF, DEVICE_STATUS and so on. If your program works by 'recursive descent', and performs lots of calls while it decides what must be done, you may run into cases where you can go no further and want to stop, perhaps with a 'user error' message of your own.

In these circumstances you must usually write lots of bulky, boring code to ensure that every routine still RETurns normally, 'unravelling' any locals created on the way. If you just hop out with a GO TO or a procedure call from which you do not intend to return, the system will slowly fill up with redundant LOCALs and parameters after every error, until all memory is exhausted.

SUPERCHARGE used to do this, as the parser had to be very concise and no alternative trick was available. This sometimes made the parser stop before it had finished scanning a program.

TURBO uses RETRY_HERE to mark the place where scanning is to re-start after the compiler has 'got stuck'. A RETRY after such an error tidies up local and temporary variables with no need to ensure that all the code returns normally. This saves time, saves memory, and saves code. Such a facility is commonly required by complex analytical software, yet very few programming languages support it. In 'C', for instance, you have to use LONGJMP and SETJMP.

The interpreter can't cope with this sophistication, which is a shame as it generates a lot more 'temporary data' than TURBO does, as it works. You can still test programs by making the RETRY conditional, and using a procedure-call to recover without unravelling unless the code is compiled:

```
IF COMPILED: RETRY: ELSE SCAN
```

SCAN is assumed to be the routine with the RETRY_HERE directive -ignored by the interpreter - at its start.

Turning off error trapping

It is easy to turn off TURBO error-trapping, so that errors are reported normally. For each level of trapping that you want to disable, insert code like this:

```
WHEN_ERROR 0
  CONTINUE
END_WHEN
```

Limitations of error-trapping

Although TURBO error trapping can 'tidy up' operations performed inside the compiled program's environment, such as LOCAL allocation, it can't wind back time as far as 'the outside world' is concerned.

If an error causes a file to be closed by the relevant device driver, or a random access position to change, it is up to the programmer to recognise this case and to include appropriate code to correct it - or, at least, to recognise the futility of attempts to recover fully in such cases.

Error-trapping by example

The simple program overleaf copies files from one medium to another, using free memory as a buffer. It works nicely as a spooler when compiled, and illustrates the use of WHEN_ERROR together with normal toolkit error-trapping. Notice the superiority of RETRY_HERE over the interpreter's RETRY, which would keep re-starting sequences of doomed commands!

```
REMark Super spooler - Error trapping demo
IMPLICIT% cause
DIM in$(42),out$(42)
IF COMPILED
  WHEN_ERROR 0
    cause=ERNUM%
    SELect ON cause
    =-3: PRINT "Not enough memory to copy ";IN$: RETRY
    =-11: PRINT "Aaargh: Bad or changed medium!"
    =-12: PRINT "There's no such device.": RETRY
    =-9,-7: PRINT "Unable to load ";IN$: RETRY
    =-8,-16,-20:PRINT "Unable to save ";OUT$: RETRY
    =REMAINDER :PRINT "Oops! Error"!cause!"at"!ERLIN%
    END SELect
  END_WHEN
END IF
REPeat spool:RETRY_HERE:READFILE:WRITEFILE

DEFine PROCedure READFILE
  REPeat infile
    INPUT "Input file (x to stop): ";in$
    IF in$="x": STOP
    OPEN_IN #3,IN$
    SET_POSITION #3,1E6
    size = POSITION(#3)
    CLOSE #3
    IF DATASPACE(in$)<0: PRINT "Not data.": NEXT infile
    ptr = ALLOCATION(size)
    IF ptr < 0 : PRINT "Too big.": NEXT infile
    LBYTES in$,ptr
    EXIT infile
  END REPeat infile
END DEFine READFILE

DEFine PROCedure WRITEFILE
  INPUT "Output file (x to stop): ";out$
  IF out$="x": STOP
  SBYTES out$,ptr,size
  DEALLOCATE ptr
END DEFine READFILE
```

The LBYTES, SBYTES approach makes this routine much faster than COPY on long files. It also does not swap between drives all the time, which is desirable in a multi-tasking utility, where you may want to swap media every so often. Of course, if you're doing something like that, the error-trapping is especially important.

Of course, errors and speed may not bother you. You may be happy with the ultimate CONCISE multi-tasking spooler:

```
REPeat L:INPUT A$,B$:COPY A$ TO B$                !!
```

DEBUG and DEBUG_LEVEL

It is sometimes useful to be able to test a program in stages, building it up as each stage is proved correct. The DEBUG facility helps you to do this.

This feature, which allows a program to be selectively compiled, works as described here. The commands DEBUG and DEBUG_LEVEL can be issued anywhere in the prgram.

Syntax

```
DEBUG_LEVEL <num>
```

where <num> is an integer from 0 to 9 inclusive, sets the top level at which lines in the program will be ignored. This can be issued only once.

```
DEBUG <num>
```

can be issued several times and sets the current debug level. Any lines which follow are ignored if <num> is greater than the top level set by DEBUG_LEVEL, or until a different current level is set.

If DEBUG is given without a following number, 0 is assumed and a warning issued.

If DEBUG is given before DEBUG_LEVEL, the top level is taken as 0 and DEBUG_LEVEL thereafter has no effect.

If neither DEBUG nor DEBUG_LEVEL is issued in a program, both the top and current levels are set to 0 so that all lines are accepted.

The purpose of this is to allow successively more of a program to be compiled with successively higher numbers of DEBUG_LEVEL <num> being given, with the same DEBUGs scattered throughout.

Thus if a program is divided into sections by various DEBUG k commands, starting the program with DEBUG_LEVEL 5 will allow compilation of all sections with k up to 5, but all sections with a higher number will be ignored. All sections with k = 0 will always be compiled. DEBUG_LEVEL 9 will force the entire program to be compiled.

A special example of this is when some code is intended to be used only when the program is run in S*BASIC. This code will be excluded from the compiled program if it is headed by DEBUG 1 and ended by DEBUG 0. This is an superior alternative to the use of IF_COMPILED/END IF. Not only does this eliminate the need for code inside the compiled program for IF_COMPILED test, it also eliminates the compilation of the instructions used only in S*BASIC. This is especially useful when the S*BASIC instructions contain extension names not expected to be loaded when the compiled program is run.

TURBO_V

One of the most crucial elements of Turbo is the Vector Table. This mirrors the Name Table in SuperBASIC in that there is one entry for every name used in the program. The Vector Table entries are all 4 bytes long and contain information as indicated later. TURBO_V is intended to give a programmer access to this Table.

TURBO_V is a function requiring a single name parameter and returning a floating point value. In a compiled program the number returned is the absolute address of the Vector Table entry for the name given as the parameter (with or without quotes).

The parameter is case sensitive and must correspond with a name used in the program. If not, Parser_task will signal an error during compilation. Also the parameter must be simple. That is, it cannot be a combination of strings or a slice of a string.

The names used in a program fall into three main categories:

variables

procedures and functions

keywords

Each name has a place allotted in the Vector Table, though not all entries are equally useful. The entries for most SuperBASIC procedures and functions as well as for those SuperBASIC keywords specially dealt with by Turbo are zero. A list of the latter is given below. In a GLOBAL module the entries in the Vector Table for all procedures and functions marked by TURBO_P or TURBO_F point to their place in the compiled program.

Variables

a. Integers

The entry for each integer variable is a long word pointer to the word value. Hence at any time this value could be obtained by, for example

```
PEEK_W(PEEK_L(TURBO_V(v%)))
```

where the variable is called v%.

b. Floating Point Numbers

The entry in the Vector Table for floating point numbers is a long word pointer to the 6-byte QDOS representation of the number.

Such a number can be found as

```
STRINGF(PEEK$(PEEK_L(TURBO_V(fpno))),6).
```

This expression returns the fp value of the 6 bytes pointed to by the address in the Vector Table entry for fpno.

c. Strings

The long word entry in the Vector Table for a string points to a descriptor, such as is used for all arrays, since a string is an array of characters.

The descriptor for a simple string consists of:

A long word pointer to the string itself

A long word containing the length needed to hold the string

A word containing -1 (indicating no array dimensions)

A word holding 2 + the DIMensioned length of the variable rounded up to the higher even number

The string itself, at the address given in the descriptor, is a word length followed by the number of bytes indicated.

d. Arrays

Arrays are slightly more complicated than simple strings. As with strings, the Vector Table entry points to a descriptor which is as follows:

A long word pointer to the values

A long word containing the length needed to hold the values

A word giving the number of dimensions less one

This is followed by a number of words containing the size of each dimension, in reverse order.

For a string array there is an extra word leading the others. It contains the maximun string length + 2, as for a simple string.

Access to the individual items can be obtained from the Vector Table address for a variable.

More useful, perhaps, is the ability to switch the value of variables. This can be done by switching the pointers rather than the values themselves. Thus, if the addresses of two variables v1% and v2% are adr_v1 and adr_v2, obtained from

```
adr_v1 = TURBO_V(v1%), and
adr_v2 = TURBO_V(v2%)
```

we can switch values by:

```
spare = PEEK_L(adr_v1)
POKE_L adr_v1,PEEK_L(adr_v2)
POKE_L adr_v2,spare
```

If v1% and v2% are arrays this could be especially useful, since swapping individual values of large arrays is time-consuming.

Keywords not Specially Dealt With by Turbo

Turbo activates keywords generally by setting up a SuperBASIC environment and branching to the code implementing the particular keyword. The 4-byte entry in the Vector Table for such keywords is the absolute address of the code.

List of keywords with a zero entry in the Vector Table

```
BLOCK     CHR$         CLEAR     CODE      COMPILED  DIMN      EOF
ERLIN%    ERNUM%       FILL$     INPUT     LEN       MOVE_MEMORY
NEW       OPTION_CMD$  PEEK      PEEK_L    PEEK_W    PEEK$     PI
POKE      POKE_L       POKE_W    POKE$     PRINT     READ      RESPR
RUN       STOP         TURBO_V
```

There is a type of problem TURBO_V is especially good at solving. The problem is this. A programmer wants to use a specific extension keyword, but only if it is loaded at runtime. This may be because the program is designed to be run on several different levels of operating system, some of which will have this keyword and some not.

Enclosing the line containing the keyword in an IF ... END IF clause is not a solution, because when the program is run on a machine not having this word, it will stop with the message " .... NOT LOADED".

The solution is to use instead a different keyword which will be loaded at runtime and to use TURBO_V to trick the program into using the required keyword.

To do this we use the function BASIC_ADR which appears in Turbo TK code from version 3.32 onwards.

BASIC_ADR(num) returns the address of the machine code function or procedure whose index number in the Basic Name List is num. If num is out of range, or if the name is not a machine code function or procedure, BASIC_ADR returns zero.

BASIC_ADR can be used in conjunction with TURBO_V to make one keyword in a compiled program act as another as shown here. Thus, the keyword SOMETHING$ will be activated in a program by substituting DATE$ as follows.

```
ads = BASIC_ADR(BASIC_INDEX%("SOMETHING$"))
IF ads <> 0 : POKE_L TURBO_V(DATE$),ads
```

From now on DATE$ can be used whenever SOMETHING$ is required.

This is useful in programs which have to operate in environments where the keyword SOMETHING$ may not be loaded. If SOMETHING$ were used directly in the program but was not loaded when the program was executed, it would fail with the message

SOMETHING$ not loaded.

Substituting DATE$, which will always be there, allows the program to operate even when SOMETHING$ is not loaded.

It is necessary to choose carefully the keywords to substitute for a particular candidate. They must be of the same type as the candidate and also must allow any parameters. Although there are several possibilities, it is recommended that the substitutes be limited to

ADATE – for procedures

BASIC_B% – for integer functions

DATE – for floating point functions

DATE$ – for string functions

From version 3.33 onwards Turbo TK code contains three functions -TURBO_DUMMY%, TURBO_DUMMYF and TURBO_DUMMY$ as well as one procedure -TURBO_DUMMYP. These all give rise to "not implemented" if they are used directly, but they are very useful for reserving a place in the Vector Table to be filled with the address of the machine routine we really want to use. These can be used in place of the four keywords suggested above.

CHAPTER T - CUSTOMISING TURBO

A. Parser_task

Versions of Parser_task v4.5 and beyond have a Config Block which allows configuration either by Config (level 1) or Menuconfig. The latter shows enough information to enable a user to determine the limits allowed to any configurable item. Moreover Menuconfig allows changes of all items for as many times as a user wants before accepting the new file.

On the other hand Config gives less information and repeated attempts at configuration will be needed to determine the types of item to be configured and the limits allowed. For that reason, this Appendix lists the configurable items in Parser_task in the order in which they are presented by Config, showing the type and limits of each. The types of item are, Integer (I), Character (C), Code (D) and String (S).

Integers are changed by typing the value required, which will be forced to be within the limits laid down.

Characters are changed by typing the character on the keyboard. Note that Config will not accept any keypress containing a cursor key.

Codes are changed by cycling through the available options using SPACE and accepted by pressing ENTER.

Strings are changed by typing in the required name.

Available Configuration Settings

Description of Item	Type	Limits
Number of Windows	I	0 to 3
Default Dataspace	I	1 to 999
Default Buffer Size	I	1 to 999
Background Colour	I	0 to 255
Listing Depth	I	10 to 190 (Note: should be 10, 20 etc.)
Default String Length	I	40 to 257
Key to shrink window	C	ASCII 0 to 31 and 224 to 255
Key to expand window	C	ASCII 0 to 31 and 224 to 255
Qram on/off	D	On, Off
Produce Listing	D	Yes, No
Sound on/off	D	On, Off
Pause	D	Yes, No
Optimisation	D	Brief, Fast, REMs
Line Numbers	D	Include, Omit, Display
Local Strings	D	Report, Ignore, Create
Structure	D	Free Form, Structured
Program Size	D	< 64K, > 64K
Display x_csize	D	Normal x1, x1 1/3, x2, x2 2/3
Display y_csize	D	Normal Height, Double Height
Object file name	S	Maximum length 25
Default Report	S	Maximum length 25

Suggested configurations

(1) FAST COMPILATION (lines/minute)

This configuration is ideal for repeated compilations during a development. Set unlisted controls as you wish.

Object file device: RAM (best) WIN or FLP

Default Report: SCREEN or RAM

Default Buffer Size = as much as possible - but there's no benefit to be had from making it more than the size of the loaded SuperBASIC - you just lose the extra untilTURBO finishes compiling.

Produce Listing: NO

Optimisation: BRIEF

Line Numbers: OMIT

INCLUDE is marginally slower, but may be a better choice unless you're ABSOLUTELY CERTAIN that code is going to work. If an error crops up after you've set OMIT, don't

even bother to guess at the cause till you've re-compiled the task with INCLUDE, to find out the erroneous line number.

Local Strings:IGNORE $

(2) COMPILATION OF SLOPPY CODE

If you're compiling a program from Quanta library cartridge -22/7, or a part-listing from Global QL in the dark, you'll need to make TURBO as forgiving as can be! These settings will instruct TURBO to be as careful as possible. They will compile the vast majority of correct or nearly-correct SuperBASIC programs, and clearly indicate the major flaws of other ones. Look through the listing file to see TURBO's 'lab report' on the program!

Output file device: RAM, WIN or FLP.

Report file: SCREEN, (PAUSE: YES), RAM, WIN or FLP

For speed, DON'T pick the output file drive or device.

Buffer size: as much as possible, for fast compilation, but see (1) above.

LIST: YES

LINE NUMBERS: INCLUDE

LOCAL STRINGS: CREATE $

STRUCTURE: FREEFORM

(3) COMPILING FOR CONCISE CODE

These settings trade speed for space. If a task is more than 64K long, consider splitting it and using LINK_LOAD.

Output file: RAM disk, disk or MDV.

Report file: as you wish.

Buffer size: as much as possible, for fast compilation, but see (1) above.

LIST: as you wish.

OPTIMISATION: BRIEF

LINE NUMBERS: OMIT or DISPLAY

LOCAL STRINGS: CREATE or REPORT $, as you wish. (but see (1) above)

(4) COMPILING FOR FAST CODE

These settings will give you fast, but perhaps very verbose code. Sometimes this verbosity does not matter, but you may end up with irritatingly large files if you OPTIMISE SPEED when compiling big programs. You can get similar speed, and code more concise than interpreted SuperBASIC, if you use OPTIMISE REMs and sprinkle a few REMark + and REMark - statements around.

Output file: RAM disk, disk or MDV.

Report file: as you wish.

Buffer size: as much as possible, for fast compilation, (but see (1) above)

LIST: as you wish.

OPTIMISATION: FAST

LINE NUMBERS: OMIT

CREATE or REPORT $, as you wish. (but see (1) above)

STRUCTURE: STRUCTURED

A smattering of well-chosen IMPLICIT% statements often boosts the speed of tasks sharply, and very simply.

INCREASING TURBO'S OWN DATASPACE

If TURBO aborts a compilation with an 'out of memory' error or the report 'please increase dataspace' you can fix the problem by increasing the parser's dataspace, just as if it were any other compiled program.

The standard dataspace should be enough to compile almost any program that fits, with the parser, on a 128K QL, but a larger allocation may be required for expanded systems.

A dataspace of 60K should be enough to cater for almost all sizes of program.

The PHANTOM STRING control

This is another control that can display several messages. All of these relate to the way that un-dimensioned strings in your program are treated. These strings may appear as LOCALs or parameters, but TURBO has no way of knowing whether or not they are used in other contexts.

All strings in TURBO must be dimensioned, so that they can be processed efficiently. If a string is used, but never declared in any way, TURBO assumes a maximum length of 100 characters, and issues a warning. You can change this maximum, and get rid of the warning, by adding an explicit DIM.

Set the PHANTOM STRING control to 'Create $' if you want LOCAL and parameter strings to be dimensioned globally too, just in case. Set 'Report $' if you just want a list of such names. Set 'Ignore $' and TURBO will assume that you know what you're doing!

B. Codegen_task

Codegen_task sets the stack space for a compiled program. Recent versions of Codegen_task have set this at 350 bytes. Earlier versions set 250 bytes.

Almost all compiled programs find that amount of stack space ample. Some exceptions are those programs using QMenu. This program is very useful for finding files, but it does require a lot of stack space. To accommodate this, versions of Codegen_task from 3.15 onwards can be configured to set a stack space from 350 to 2048 bytes, for library versions 5.25 to 5.30 and from 350 to 9998 bytes for versions 5.31 and later. QMenu will need possibly 800 bytes, maybe more.

It is suggested that programs using QMenu be compiled with a version of Codegen_task configured to set a stack of 1200 bytes. The normal version setting of 350 can be used for other programs.

Another way of increasing stack space is by the use of the program ADJ_DS. This can alter the stack space of any program compiled by Turbo and also the dataspace of any executable program.

ADJ_DS runs under the Pointer Environment, so that its window can be moved and it can be put to sleep as a button. To use it, assuming that it resides in the directory PROGD$, simply type

```
EX ADJ_DS
```

To pick a file press "p" or click on "Pick a File" and you are asked to choose the file to be altered either by QMenu if it is present or by typing its name. If it is a type 1 file its dataspace is displayed. If in addition it was compiled by Turbo its stack size is also displayed. These can now be changed. A change is started by pressing "d" or "s" or by clicking on "Dataspace" or "Stack".

The new value must then be typed in. If both items were available for change you are then asked if you want to change the other. After this the program waits once more for a new file to be picked. Such a reversion also occurs if the filename given is blank, eg by pressing ESC when QMenu's screen appears.

Once a file is picked, the operation of changing dataspace or stack size can be aborted by asking for another file before pressing "p" or "s".

Note that for versions of TURBO 5.01 or later it is possible, and much easier, to set a program's stack space by using the directive TURBO_objstk or parser_task's front panel.

CHAPTER U - TUNING TURBO FOR TOP PERFORMANCE

HINTS AND TIPS

In this chapter we list and explain general techniques which will help you to get the best possible performance from TURBO.

The first half of the chapter deals with software techniques and timings. The second part discusses external influences on the speed of compiled code: devices, memory and multi-tasking.

SOFTWARE SHORT-CUTS

TURBO increases the speed of most BASIC programs by rationalising the steps needed to produce a given effect. The speed-up ratio depends upon two factors - the original steps, outlined in the program source, and the degree of 'optimisation' which TURBO performs.

TURBO is a powerful tool because it can analyse and improve virtually any program that runs under the SuperBASIC interpreter. But some improvements are not 'obvious' to the compiler; this chapter explains how you can can encourage it to produce 'better' code, by whatever standards you set. It is useful to bear these points in mind when you write new software which you intend to compile with TURBO.

Other 'improvements' may have disadvantages; they might limit the range of values which your program can use, or increase the size of the compiled task. In such cases TURBO gives you control over the option.

This is a discusssion, not a comprehensive list of TURBO's advantages over interpreted SuperBASIC. You'll find specific advice about data-types and calculation efficiency in Section 3 of this Manual.

OPTIMISATION AND IN-LINE CODE

Human programmers will produce different code depending upon the memory and other resources available. TURBO can do the same. This is called 'optimisation'.

You can tell TURBO to optimise the size or speed of your compiled task, or the help it gives when things go wrong, or the speed of compilation. Many of these options are necessarily mutually exclusive: if TURBO can think of a trick to make a program simultaneously concise AND fast, it will use it without asking you!

There are two ways to control the 'style' of code produced by TURBO. You can do it 'globally' - throughout the program - or 'locally', line by line. The first is crude but often effective, especially when testing. The second requires prior thought but gives the best possible trade-off between size and speed. Local optimisation control is recommended for 'production' software.

Optimisation controls

All of the 'top row' of TURBO front-panel controls affect optimisation, but the most important is the second last one, showing BRIEF, REMs or FAST. This is discussed in a moment, once we've got the minor ones out of the way.

FREEFORM/STRUCTURED

The leftmost panel shows either 'Freeform' or 'Structured'.

If you choose 'Structured' the program must be written so that all the code outside procedures and functions comes first. The only exception to this rule is REMarks, DATA, DEBUG and DEBUG_LEVEL. A program written this way is made more efficient by the choice of 'Structured'. If a program line does appear after a procedure or function, a warning will be issued by Parser and the line ignored.

Unless you are certain that your program obeys these rules, it would be better to choose 'Freeform' so that no program lines will be lost.

SIZE (< 64k / > 64 K)

Choosing '< 64 K' tells TURBO to minimise the size of the compiled file, by using threaded code and short (16-bit) addresses for lines and data. The use of short addresses can also give a tiny speed improvement.

This setting will not work with extremely large programs, where all the code cannot fit within a 64K area. If you fall foul of this restriction you must re-compile selecting '> 64 K'. You should find that your program is much smaller than the interpreted form, even when the default of 32 bit addresses is used.

We have compiled programs of over 2,500 lines into less than 64K, so only the most prolific will run into this restriction, and it is unlikely that they will be writing for standard 128K QLs!

LINE NUMBERS

The three options in the middle panel are OMIT, DISPLAY and INCLUDE.

Selecting OMIT or DISPLAY reduces the size of compiled tasks by four bytes for every line in your program, by removing the code that maintains the current line-number.

The snag is that all error messages from such a task are marked with line number 'zero', so you can't tell where you have run into trouble if an error does occur.

We recommend that you choose INCLUDE until you have exhaustively tested a program. You are also advised to keep a separate compiled version WITH line-numbers so that - with luck - you will be able to duplicate and find out the source of any errors that may crop up in future.

Choosing DISPLAY removes line nmbers from the program but displays them on TURBO's panel as compilation proceeds. Choosing OMIT removes line numbers from the program and suppresses printing of the line number on the screen.

Choosing OMIT increases the speed of compiled code (although this effect is marginal except in integer routines) and speeds up the compiler. It also reduces the size of compiled tasks, but debugging is rendered extremely tricky, if errors do occur.

WINDOWS

The size of compiled programs is reduced very slightly, and their memory requirement is reduced further (by about 270 bytes for each window) if you limit the number of windows that are automatically copied from SuperBASIC when you compile a program.

The right-hand control on the top line of the panel sets the number of windows, as its label indicates. You can set the number of copied windows to any number between 0 and 31, or the highest channel number open from BASIC - whichever is the smaller.

If your program does not use the display, or opens all its own windows, you can save a little code by setting the number of windows to 0. Select 1 if your program only uses the default window. Set 2 if you also use #0, the command window, and 3 if your code is badly-behaved and needs a copy of the SuperBASIC listing window!

The second to last switch allows a lot more control over the style of TURBO's code. The three possible settings are discussed below.

BRIEF

This tells TURBO to make up its own mind; in general it will only perform optimisations that minimise code SIZE and maximise SPEED, both at the same time!

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 calling each address in turn. TURBO uses an extremely fast linkage for threaded code, and the complexity of the routines in the 'library' is carefully judged so that the cost 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. SuperBASIC programs contain many repeated operations, so threaded code can dramatically reduce the size of big programs.

The relationship between the size of source code and generated code is not constant - it depends upon the actions being performed. Very short, complex programs may require most of the library 'routines' - but TURBO tasks never include routines unless they actually need them.

The original SuperBASIC version of TURBO's parser would not load into a standard 128K computer, while the compiled version leaves space for a substantial BASIC program alongside the parser. The saving is more than 50 per cent of the final code size - and TURBO stores variable values and 'return' information more concisely than the interpreter, too.

The alternative to 'threaded' code is 'in-line' code. In this code, an option on TURBO and the only possibility for other true compilers, all the routines are written out in full and executed in a continuous stream. In-line code is not used very much when you select 'no' optimisation, but it is useful in other cases...

FAST

In this case TURBO translates the entire program to in-line 68008 machine-code, with a minimum of subroutine calls. This is longer than the default, 'threaded code', but faster, particularly on integer operations. To understand this you need to know a little about the way TURBO generates code.

The second part of the compiler, the code-generator, assembles 68008 machine code for each operation as required by the first part (the parser). When 'threaded' code is being produced, the code-generator only includes each routine once, and then only if it is definitely needed, so programs can be compiled into compact tasks.

About 4K of code and pre-allocated working storage is always present -this includes routines to set up the task, communicate with the ROM, trap and handle errors, open windows and so forth. Part of this space is 'released' when the task has finished setting itself up, and becomes available as dataspace.

'Raw' 68008 machine code is, by nature, extremely verbose - it takes four bytes of code, for instance, to compare a register value with a character! If you're really keen you can improve the performance of compiled programs by the use of in-line code, but the results are only marginal and you should expect an increase in the size of the compiled task as a result.

You can force the code-generator to use in-line, rather than threaded code, throughout your program by selecting FAST, However this may result in a very large task; TURBO's in-line code is much more concise than the code generated by its predecessor SUPERCHARGE, but it's still much more long-winded than threaded code.

REMs

This is a variation on the BRIEF and FAST options which can give most of the advantages of both. The snag is that you must analyse your program to find out which parts are taking most of the time.

You may be able to pin-point the key routines from your own knowledge of the program design, but in some cases automatic help is useful.

When you know which lines are time-critical, re-code them, making the least possible use of extensions and floating-point commands - use integer arithmetic wherever possible. Then put a pair of special REMark statements around the lines, to indicate that you want them to be 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 everything after a REMark is ignored). It is a good idea to put these REMarks on lines of their own, since this makes them easy to find later. It doesn't matter whether you enter a space between REMark and the '+' or '-' sign.

N.B: There is one small restriction on the placement of these REMarks. If a FOR statement is in in-line code, the corresponding END FOR and any intervening NEXTs and EXITs MUST also be in in-line code - not threaded code. The fix for this 'bug' would cost more time than REMark + saves!

Tasks which use in-line code are larger and (in appropriate circumstances) faster than their threaded counterparts. In general REMark + is only useful on assignments and 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.

TURBO_... and MANIFEST

TURBO TOOLKIT contains a set of 13 commands such as TURBO_taskn as well as the command MANIFEST. This section indicates how they can be used to make a program compile and run faster.

TURBO_...

The front panel of Parser_task enables several switches and filenames to be set during the current run. The previous paragraphs show how to set these for increased performance. The defaults in Parser itself can be set by the executable program Config. Some of these defaults can be altered at run time by the set of commands TURBO_locstr etc. which are described here. These defaults can of course be overridden by using the options available on the front panel at run time.

The last command, TURBO_ref, is rather different from the rest. It takes no string or number and it does not appear on the front panel or Config Block. Its purpose is to indicate that string parameters are to be sent by reference to machine code routines. More information is given in Chapter P.

TURBO Configuration Commands

Name	Number/String	Meaning
TURBO_locstr	N/S	0 = Ignore$ 1 = Report$ 2 = Create$
TURBO_windo	N/S	0 to 32 windows saved
TURBO_optim	N/S	0 = Brief 1 = Rem 2 = Fast
TURBO_model	N/S	0 = >64K 1 = <64K
TURBO_struct	N/S	0 = Freeform 1 = Structured
TURBO_diags	N/S	0 = Omit Nos 1 = Display Nos 2 = Include Nos
TURBO_list	N/S	0 = No 1 = Yes
TURBO_sound	N/S	0 = No 1 = Yes
TURBO_objdat	N	number = dataspace/1024
TURBO_objstk	N	number = stack (bytes)
TURBO_buffersz	N	number = buffer size/1024
TURBO_taskn	S	Task Name
TURBO_objfil	S	Name of Compiled Program
TURBO_repfil	S	Name of Report File
TURBO_ref	-	String parameters by reference

An example of these used in a program might be:-

```
100 TURBO_objfil "ram1_test_task":TURBO_optim 0
```

NOTE: In the cases N/S, where either a string or a number can be given, the number must be one of those shown. If a string is used it must contain as its first character (without regard to case) the first character shown as the Meaning. Thus, in the example above 0 could be replaced by "brief", or "B".

The use of the TURBO_... commands in a program ensures that each time the program is compiled the same options are chosen. Also, the total time taken to compile the task is lessened since there is no need to use the fromt panel to change any settings.

Indeed, if compilation is started with "CARGE \", there is not even the need to press SPACE to start the compilation.

MANIFEST

A way of increasing the running speed of a program is to use MANIFEST.

A compiled program is faster if constants are used directly rather than being referred to by a variable name. For example, a programmer might decide to define colours by:-

```
200 black% = 0 : white% = 7  etc
```

To achieve the advantage of using direct numbers as well as having them defined by names the command MANIFEST can be used. Thus:-

```
200 MANIFEST : black% = 0 : white% = 7
```

will have the effect of 0 and 7 being substituted directly in the compiled program for black% and white% wherever these are used.

MANIFEST can also be used for strings. Thus:-

```
200 MANIFEST : open$ = "Open the door"
```

will result in the direct substitution of the string "Open the door" in place of each use of open$.

Of course the names used in MANIFEST statements cannot be used as variables. Thus, using the above example,

```
300 open$ = "Shut the door"
```

will result in the error message

**** ERROR at 300: can't assign value - MANIFEST

LARGE PROGRAMS

Long programs are often accelerated by TURBO much more than you might expect. Contrary to Sinclair's advertising claims, the SuperBASIC interpreter gets steadily slower as program sizes increase, while the speed of compiled programs remains constant, and very much faster.

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-range' reference in a large interpreted program can be 100 times slower than one which only spans a few lines.

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. GO TO, GO SUB, IF, FOR, REPeat, NEXT, EXIT, RETurn, END, procedure and function-calls are all interpreted by scanning through the program, whereas TURBO jumps directly from place to place.

The bigger your program, the greater the speed-up factor you can expect. All TURBO jumps take less time than the interpreter spends skipping a single line, so the speed-up factor can be very impressive in programs of hundreds or thousands of lines - 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 keep routines in a logical order, or to make extensive use of structured loops, procedure and function calls. They will be 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 the structured commands. These will work quickly in compiled programs, but so would their structured equivalents.

Of course there's more to program execution than jumps from place to place, but this is still a useful improvement which is not shown by short 'benchmark' programs designed to test interpreters.

D-I-Y OPTIMISATIONS

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.

Loop optimisations

In general there is little point in trying to accelerate sections of a program which wait for input or do not form parts of loops. '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
```

This is a rather gross example but a relevant one: similar cases often crop up in 'real' programs. One especially common inefficient construct is this:

```
FOR I=0 TO 32767:POKE I+131072,-1
```

This version runs almost twice as fast:

```
FOR I=131072 TO 131072+32767:POKE I,-1
```

There's no real need to substitute the value of the expression 131072 + 32767, as the start and end in a FOR statement are only evaluated once, when the loop begins to iterate. We would argue that 131072 + 32767 actually conveys more information than 163839.

Eagle-eyed readers will have spotted another possible improvement:

```
FOR I=131072 TO 131072+32767 STEP 4:POKE_L I,-1
```

This is almost four times faster again... but we can do many times better with TURBO TOOLKIT:

```
X$=FILL$(CHR$(255),128)
FOR I=131072 TO 131072+32767 STEP 128:POKE$ I,X$
```

Of course, it would be nice to get rid of the loop...

```
POKE 131072,-1:MOVE MEMORY 131072 TO 131073,32767
```

TURBO can optimise MOVE_MEMORY between even addresses so this is (probably) the ultimate version:

```
POKE_L 131072,-1:MOVE MEMORY 131072 TO 131076,32764
```

You'll need to compile this to take advantage of the optimisation. Don't blink, or you won't be able to tell the difference.

Not all program fragments are this amenable to change, but we hope you'll agree that it's a neat example!

Floating point

TURBO 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 what action is to be performed.

In this case TURBO, 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 which sum suitable series. This is because TURBO can perform the fundamental integer operations of multiplication, division and so on very quickly. Such tricks are discussed in Chapter M of this Encyclopedia, "Fast arithmetic with integers".

Minimising external calls

Alternatively, you might be able to replace function-calls with variable or array-references. This technique is useful if you use the same values of a function again and again. TURBO 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.

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. TURBO 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 always faster than SELECT, since one 'test' handles all instances.

'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 B THEN ...
```

The first test should be the one most likely to fail. You should be careful to keep END IFs matched properly.

COMPILERS AND MACHINE-CODE

Although TURBO 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. Compiled code also keeps track of the BASIC line-number corresponding to the code being executed, so that you can identify errors easily.

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 occurred. 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.

Unlike its predecessor, SUPERCHARGE, which always checked everything, TURBO does give you some control over its diagnostic checks:

(1) If you indicate in-line code with REMark + or FAST the value of the result of an integer add, subtract or multiply operation is not checked. Any 'extra' overflow bits are discarded, with no message. You should compile your program and test it in threaded code first; the interpreter does not

always detect integer overflow.

Invalid array subscripts, parameter types and integer division operands are still checked, and all string and floating-point checks are performed as normal.

(2) You can stop compiled tasks keeping track of the current line-number as they run, by compiling with OMIT Nos. This will make the task smaller and slightly faster, but it makes de-bugging a nightmare!

Compatibility

TURBO 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:

(1) 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 restricted by the use of floating-point addresses.

(2) 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 TURBO cannot make assumptions about them without running the risk of incompatibility.

(3) Since SuperBASIC is an extensible language, the compiler must support extensions intended for the interpreter. TURBO can optimise TOOLKIT facilities and standard QL commands, but other extensions must be treated in a 'general purpose' way. This limits the efficiency of compiled programs, as they must be able to 'make do' with code intended for the interpreter.

In a few minor cases extensions cannot be supported by TURBO because they make assumptions that mean they must be called from an interpreter. We have implemented the maximum compatibility that can exist without greatly compromising the efficiency of compiled code.

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 some improvement to TURBO 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 TURBO 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 TURBO 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 to 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 TURBO at a level where it offers almost all the advantages of machine-code, without the drawbacks.

EXTERNAL FACTORS AFFECTING TASK SPEED

The speed of machine-code programs that run on the QL is not just dependent upon the quality and efficiency of the code. Three other factors also affect code speed:

1. Devices and peripherals
2. Memory
3. Multi-tasking

In the remainder of this chapter we explore the effect of each of these factors on TURBO and SuperBASIC execution speed.

DEVICES AND PERIPHERALS

TURBO cannot, in general, increase the maximum rate at which peripherals can transmit or receive data. After all, printers and other serial devices would not be able to respond if TURBO cranked up the RS-232 interface speed from 9,600 baud to 192,000 baud! Even if the electronics was built to support such a speed - and it isn't -connectors and external components could not cope.

The same physical limitations stop TURBO from spinning floppy disks at 6,000 RPM or snarling up microdrive tape at 50 feet a second! But TURBO can give dramatic speed improvements on some programs that make heavy use of files and devices, because it reduces the delay between data-transfers so that data can be packed more densely.

This effect varies widely depending upon the program and the device -the only way to discover whether or not your code falls into this category is to compile it.

If you want a demonstration of this effect, bear in mind that the original, SuperBASIC version of the TURBO parser takes more than five minutes to load from disk, whereas the compiled version loads in a few seconds. The difference is nothing to do with the disk drive or interface - it just indicates that the data is processed more efficiently as it is loaded, so that much more data can be read in a given time.

TURBO's display handling speed is limited by the QL's hardware and software design. Display software is discussed in Chapter O of this Manual. Hardware problems associated with the QL's display are discussed in the next section of this chapter, 'Speed and Memory'.

SPEED AND MEMORY

The QL could run many programs almost twice as fast if Sinclair hadn't taken a 'short cut' in the machine's design. The speed of the memory is deliberately slugged to simplify the design of the computer. This hidden kludge does not affect the memory in a well-designed expansion RAM unit, so the QL's performance leaps ahead when extra RAM is fitted.

Some makes of add-on memory (such as CST's RAM Plus) allow values to be stored or read at almost twice the rate at which the QL's standard 128K can work. This can have a noticeable effect on the speed of some compiled programs.

In this section we'll discuss the reasons why external RAM can run so much faster than standard QL memory. If this latter-day industrial archaeology leaves you cold, or you only want to know the results, not the reasons, please skip the next couple of pages and continue reading from the heading 'Getting ahead'.

Not so dynamic

The Sinclair QL contains 16 standard 64K bit RAM chips, giving 128K bytes of memory. The computer uses so-called 'dynamic' memory - this is cheap and consumes little power, but it can only store information reliably for a fraction of a second. Dynamic memory uses a charge, akin to static electricity, to record information. The charge constantly leaks away, so it must be replenished regularly.

This isn't usually a major problem - dynamic memory is used in lots of other computers, from the IBM PC to the ZX Spectrum. Normally a special circuit is used to 'refresh' the memory, by cyclically reading values from every address and then writing them back. The Z80 processor contains circuitry to do this automatically, but there is no such circuit in the QL.

The QL doesn't forget all its data because all 128K of built-in memory is constantly being read and re-written by the circuit that generates the screen display.

The QL on display

An area of 32K is reserved as 'display memory'. The computer converts the pattern of bits in that memory into a pattern of dots on the TV (or monitor) screen. A TV picture is built up as a single beam scans across the screen, line by line. As the beam moves over the screen, the computer reads values from the memory. The values read determine the colour of the dots displayed.

The video memory is read (and re-written) fifty or sixty times a second, as the entire TV picture is re-drawn. This regular access is enough to ensure that the contents of the video RAM don't leak away.

Sinclair use the same circuit to refresh the memory as they use to read the video information. In other words, the whole 128K is refreshed as the beam scans over the TV tube. This reduces the cost of the computer, by decreasing the number of distinct signals flying around the circuit-board. The QL hardware took 14 months to design, and six months of that time was spent doubling-up pins on the custom chips to reduce their cost. Anyhow, the decision to re-use the video circuit has some annoying consequences.

Each memory chip can only 'talk' to one device at a time. Strange things would happen if the processor tried to set a bit to zero while the video generator was trying to 'refresh' it with the previous value, one. This is a common problem, since both devices run at the same time and the processor will often need access to video memory - to write a message, for instance, or to clear a window.

You might make the video generator wait its turn when the processor is busy fiddling with memory. But the video chip can't just wait. The beam is still zooming across the TV screen - by the time the data is ready the beam will be somewhere else, and the video chip should be reading a different bit.

To 'correct' this you could make the video chip produce a fixed colour (say, black) whenever it tried to read memory that is being used by the processor. This works, but it produces unsightly coloured streaks on the screen whenever the processor is adjusting the display (printing text, for example). This is the cause of the 'snow' which appears on the display of early IBM PCs (but not clones), and other crude micros -Acorn Atoms, TRS-80s, and many others.

This flicker is unacceptable on a posh modern computer, and there is an accepted way of getting rid of it. You use very fast memory, so that the processor AND the video generator can both read or write the same bit before the TV beam has time to move across the display.

Unfortunately the QL display uses a lot of memory. A complete 32K picture must be produced 50 or 60 times a second, and this doesn't leave much time for the processor to get a look in. If we were to fit memory that could cope with the theoretical demands of the processor AND the video generator, it would have to allow for about six million accesses a second - four million for the video (twice the top rate used by the BBC Micro) plus two million for the processor.

That's the best possible case, assuming that the video generator and the processor are synchronised perfectly. In practice the processor and video

generator don't take turns at exactly the right moment. You need even faster RAM to make up for the time wasted while the processor and video generator are working out who's turn it is.

Remember, this speed is required from ALL the memory, not just the 32K used for the video display, because Uncle Sir Clive and his merry persons decided to make the video circuit refresh the entire 128K.

Chip choices

High-speed chips would be prohibitively power-hungry and expensive in a cheap computer. So Sinclair boxed themselves into a corner. The QL has a whizzo new high-speed processor, and a whizzo new high-speed display - but the memory isn't fast enough to cope with them. There is a solution, but it's not a happy one.

The QL uses the opposite of the technique used on the old IBM PC. On the PC, whenever the processor and the video generator both want access to memory, the video generator is denied access and the processor gets right of way - leading to gaps in the displayed picture. On the QL the reverse is true. If the video wants access to a chip, the processor is 'locked out' - stopped - until the video generator has finished.

Everything looks pretty, but the machine runs slowly. The processor can spend as much as half of its time waiting for the video generator to get out of the way.

Sinclair are the victims of their own ambition. They took video generation and processing speed to the limits of the available components, and standard memories just could not keep up. The Apple Macintosh suffers from the same problem, but it only uses a monochrome screen and this helps to reduce the amount of display data that must be processed.

There is a POKE to turn the QL's video display on and off but sadly the speed of programs is unaffected. The control only prevents the output of video information - not the process of reading it from RAM, which is vital to keep the memory contents 'refreshed' and intact.

Getting ahead

The solution is to add an extra bank of memory with its own independent refresh circuitry, outside the control of the video chip. Such a RAM board can double the effective speed of the processor.

Many firms produce memory boards that work faster than the RAM inside a QL, but it takes considerable skill to design a memory board that will work reliably at the maximum possible speed. We compare many brands later in this chapter.

Many manufacturers deliberately slug their circuits with extra delays, called 'wait states', to simplify design and manufacture. In fact there is some justification for this policy. Few programs are limited entirely by the speed of the memory in which they run. Most programs stop and start reading memory many thousands of times a second; the delays occur while the processor works out where to go and what to do, and this is done at the same speed regardless of the type of memory.

Instruction and processor variations

The amount of time a program spends waiting for memory varies depending upon the machine-code instructions being performed. Without going into too much detail, this proportion can range from 100 per cent of the time taken to perform instructions which are electrically 'simple', down to ten per cent when performing a signed integer division, which really makes the processor sweat!

The QL's 68008 processor, much criticised as a 'cut down' version of the Motorola 68000 used in the Macintosh, ST and Amiga, performs such complicated operations at virtually the speed of its big brother. However the 68000 can read or write twice as much memory as the 68008 in a given amount of time.

Clock speed

The higher the 'clock speed' of the machine, the smaller that amount of time is. Most mass-produced 68xxx machines use a clock speed between 7 and 8 MHz. The QL uses 7.5MHz, which means that the basic timing pulse in the machine (the 'clock') occurs 7.5 million times a second. One byte (or a word, for a 68000) can be read or written in three ticks of the clock, when the machine is running flat out. One extra tick passes while the processor is working out the next address.

In theory almost two million bytes can be read or written by a 68008 in a second, if your memory runs at top speed and the correct instructions are used. TURBO's powerful string and memory handling routines have been checked and optimised line-by-line so that they work at close to this ultimate speed.

Fast memory is especially effective if your TURBO programs make heavy use of strings, simple integer arithmetic, in-line code, MOVE_MEMORY or SEARCH_MEMORY. Floating point arithmetic (other than adding and subtracting) is not accelerated much by fast RAM, as the work is performed by ROM routines.

Calls to ROM and extensions

Many programs call external routines while they run. TURBO tasks call device-drivers, arithmetic routines and resident procedures, usually in the QL ROM. The ROM runs at 'full speed' - up to twice as fast as standard built-in RAM.

TURBO and SUPERCHARGE compiled programs contain internal data-areas which are used to keep track of data very quickly 'on the fly' as a program runs, so they cannot run in ROM. There again, it is easy to copy a compiled task into ROM and copy it back to RAM when you want to execute it. This makes sense because the internal data-areas almost always save more time than ROM storage would, especially if you have fast memory.

Extension commands loaded into RAM may be limited by the speed of that RAM. The QL allocates internal RAM first if you use the ALLOCATION function, whereas external RAM (if fitted) is generally returned by the uncompiled RESPR statement, so it is sensible to load all extensions into

RESPR'd memory with your BOOT program, before you load any tasks.

Of course, the biggest delays of all come when tasks are waiting for disks, serial links and microdrives to process data. The speed of memory will not directly affect such programs, but it will help the processor to move values once they have been fetched. You may find dramatic speed improvements if compilation means that more blocks in a file can be loaded, in the correct sequence, at each turn of the disk or microdrive.

TURBO can speed up input and output down serial links, especially using 'PRINT ;' and INKEY$, because it can reduce the gap between transmission of one character and the next. This gap can often be longer than the time actually spent transmitting or receiving data. TURBO can reduce such delays by a large factor - perhaps as much as 100 times, in some cases.

Under some circumstances TURBO can build a sequence of characters into a group and transmit them 'en masse'. This can be much faster than passing the characters to the operating system one by one, as QL interpreters do.

The fastest way to load and save data on the QL is with SBYTES and LBYTES (or the task equivalents EXEC/EXECUTE and SEXEC). These operations are very fast because they allow data blocks to be read in jumbled order, rather than a fixed sequence. Sinclair took this feature from the ZX Spectrum microdrive handler! Saving is done all at one go, rather than 'piecemeal' as other processing continues.

All of these commands have a snag which you have probably noticed: they suspend multi-tasking while they run, so that no opportunity to transfer data is missed.

Memory speed test

We have written a short machine-code routine as a test of memory speed. The routine spends 89 per cent of its time waiting for RAM, rather than waiting upon the processor.

You can calculate the 'absolute' comparative memory speed by subtracting about 1.8 seconds, the time spent waiting for the 68008, from all the timings. In practice few real programs are likely to be more than nine-tenths dependent upon the speed of memory, so the figures bear comparison.

This is the routine:

```
100 ram=RESPR(16): RESTORE 140
110 FOR p=ram TO ram+14 STEP 2: 120 READ x: POKE W p,x
120 t=DATE: CALL ram
130 PRINT "Test complete in ";DATE-t;" seconds."
140 DATA 28771,29439,20937,-2,20936,-8,28672,20085
```

Only the time taken by the machine-code is significant - the overhead of calling the code from BASIC is negligible.

Remember that RESPR will use internal RAM, by default, if called from a compiled program. For this reason you should run the code from SuperBASIC, with no other tasks running, if you want to measure the speed of add-on RAM.

The timings for the routine, tested on several popular makes of internal and external memory, are tabulated below.

Timing (S)	Expansion system	Tester
33	Standard 128K QL	DP
32.5	QUANTA 512K DIY project	DP
27	Simplex Data 512K (1986)	DP
22	Miracle Expanderam 256K	Mfg.
22	PCML 256K	DP
22	MEDIC 512K	DP
22	Inpho-link 512K (internal)	DP
22	GBC Italiana 512K (internal)	DP
19	Miracle Expanderam 512K	DP
19	Sandy SuperQboard V 1.17	DP
17.3	Simplex Data 256K (1985)	DP
17.3	CST RAM Plus 512K	DP
17	CST THOR 512K internal board	Mfg.
17	The same program in EPROM	DP

Timings are in seconds, rounded down; fractions indicate the average of many runs. DP timings were measured by Simon Goodwin, Freddy Vachha, Chas Dillon and Melvyn Pierce. Other tests were performed by the relevant manufacturer.

NOTES ON THE TIMINGS:

1. Manufacturers keep changing models. Early Simplex Data RAM boards appear to be faster than their later, more reliable, units. If speed really concerns you, run this routine on boards you are considering buying. We use and recommend the CST RAM Plus add-on board, which is well-designed and the fastest 512K QL RAM we have tested.

2. Standard QL timings vary between 32 and 34 seconds for runs in internal RAM; it is not easy to relate this variation to the version of the machine. The type of ROM fitted is irrelevant as the code does not call ROM.

3. Repeat the tests if you want accurate results. Take an average of several timings, adding 0.5 to each: the test program truncates the result to the nearest second.

4. The fastest time the routine could take, in theory, with top-speed memory, is 15.8 seconds. In practice it will not run that fast as the machine always spends some time reading the keyboard, keeping track of the time, and arbitrating between tasks and devices. This activity is discussed in the next section, headed 'Speed and Multi-tasking'.

5. There are two different types of internal RAM upgrade on the market. Some replace the QL's standard 128K with 512K or 640K by adding 512K chips 'in parallel with' the standard RAM. These upgrades, such as the Quanta DIY project, run at the same slow speed as internal memory, because they must compete with the video circuit as if they were part of the standard 128K. More sophisticated internal upgrades generally plug into the processor socket, rather than into the standard memory sockets. In effect these work like external memory expansions, even though they are inside the QL case - they can be much faster than the other type of internal upgrade.

SPEED AND MULTI-TASKING

When the QL is running a task, that's not ALL it is doing. Every 1/50 or 1/60 of a second (depending upon your display frame time) the processor momentarily stops running your program and does some 'housekeeping' of its own. It checks the keyboard in case something has been typed, updates the DATE$ clock, checks whether or not there are 'slave blocks' that must be read and written by the file system - and switches between active tasks as priorities indicate.

All of this work takes time. To some extent the amount of time depends upon the number of tasks running, but it is usually pretty close to ten per cent of the theoretical time which your programs could use. Multi-tasking and 'housekeeping' operations are further retarded by the fact that temporary values are stored in slow memory (on the Supervisor stack) while scheduling takes place. In practice the speed difference when the values are stored in faster memory (e.g. on a THOR system) appears to be negligible.

Nonetheless, the 'scheduler overhead' means that your programs run a little slower than theory would suggest. On the plus side, you can run several at once, with 'background' handling of file operations; you can type things while the machine appears busy, and the characters are not lost; and you can switch between tasks, perhaps to stop or start certain programs.

Even if TURBO cannot dramatically increase the speed of a program it may increase its utility. It might be argued that speed is irrelevant in most cases, as compiled programs can multi-task - you can get on with something else while your compiled program runs 'in the background'.

We measured the overhead of multi-tasking by adapting our speed test routine to run as if it were part of the scheduler, with no interruptions allowed. This brought the speed of internal RAM down from 32 to 28 seconds; a system using one wait state accelerated from 22 seconds to 20 seconds, and ROM ran the test code in 16 seconds compared with 17 when interruptions were allowed.

After some consideration, we decided that TURBO should not contain any mechanism to let the user disable the QL scheduler. Many devices and commands rely on the scheduler in order to work, and multi-tasking cannot occur unless the scheduler is running.

What TURBO does do, in a few carefully-chosen cases, is turn off the scheduler temporarily while certain critical commands or operations are performed. Thus we recoup some of the lost time without disturbing the DATE$ clock or preventing effective multi-tasking.

Priorities and performance

The exact performance of the QL's scheduler, and its handling of a varying mix of tasks, is a topic which could be discussed ad nauseam. We'll try

to stop before we get that far! In principle it works very simply; relatively high-priority tasks get a greater proportion of available processing time than tasks of lower priority.

The exact ratios depend upon the number of tasks being run, the devices in use, the memory configuration, and the precise purpose of each task! It is pointless to try to analyse these ratios in great detail, because they depend upon so many factors. However, specific results can be of interest, especially if they are measured in controlled circumstances.

In this section we discuss the effect of different priority settings when several copies of the same task are executed at the same time. Load separate copies of each task.

Four specific - perhaps surprising - effects are summarised below. You'll doubtless be able to think of many other comparative tests that could be performed. In practice the information below, while interesting, is probably more detailed than most QL owners will require, although a minority will doubtless have a thirst for even more data!

Absolute beginnings

It is only the RATIO of priorities that matters - not the values themselves. For instance if two tasks run at a priority of 4, and one at a priority of 12, the higher- priority task will receive about as much time as the other two put together. The same is true if the priorities of the tasks are 40, 40 and 120. In other words, there's no such thing as an 'absolute priority'. All priorities are relative. This is as true of QL scheduling as it is of politics, although hacks would have us believe otherwise!

Real and notional differences

The ratio of timings does not correspond directly to the ratio of priorities. Timings change less than priorities.

For instance, a priority ratio of 10 (e.g. one task at priority 90 and another at priority 9) gives a speed ratio of 8 between two tasks. A priority ratio of 5 gives a speed ratio of 3.7. A priority ratio of 3 causes the task with the higher priority to run 1.9 times as fast as the lower-priority task.

Performance degradation

The number of tasks loaded does not have much affect on the total amount of processing time shared between tasks. An extra 'dormant' task (e.g. the idle SuperBASIC task, waiting for a command) slows the active task, but only by a very small amount. If the test takes 27 seconds from BASIC, four concurrent runs from tasks, with the interpreter as an idle fifth task, will take 112 seconds. This is only four seconds more than if the runs were performed sequentially by a single task, and you can type commands or use BASIC as the concurrent tasks run.

The limitations of subtlety

When three or more tasks are active, the ratio of execution timings will only bear a vague relation to the relative priorities. It takes a big difference of priorities to make a difference in the speed ratios. Tasks will tend to settle out into 'groups' which will run at very much the same speed despite small priority variations. To offer some very general illustrations: if three tasks are running at relative priorities of 1, 2 and 3 (e.g. 40, 80 and 120) the one with lowest priority will run at about half the speed of EITHER of the others. Priority ratios of 1, 2 and 4 make the highest priority task run at about twice the speed of the other two. Three tasks at relative priorities of 4, 5 and 6 will run at much the same speed!

Further experiments

If this information has not satiated you, write a simple 'count and print' loop in SuperBASIC and compile it. Run several copies of the task, displaying data in different parts of the screen, and compare their speed with their relative priorities. You need to be able to start all of the tasks at more or less the same time, if you want to compare concurrent runs. The easiest way to do this is to load the tasks with a priority of zero, so they do not run immediately after loading. You can do this with EXECUTE:

```
EXECUTE TASKNAME ! 0
```

When all such tasks are loaded you can use LIST TASKS to find the task identifiers, and SET_PRIORITY within a loop to start all the tasks at roughly the same time.

There are lots of other ways to get comparative timings, and this is a fascinating area for research. The underlying algorithms in the scheduler are quite simple, but sometimes the interplay between tasks may cause unexpected results.

CHAPTER V – THE TURBO STORY

WAITING FOR TURBO (BUT NOT GODOT)

TURBO was 'born' within a few days of the public launch of the QL, yet it took almost three years to reach the market. This chapter tells some of the story of those years.

In January 1984, having just finished work on a TV series about micros for Central ITV, Simon N Goodwin found himself at the Which Computer Show at the National Exhibition Centre in the West Midlands. At the time, Simon was finishing work on ZIP, a simple but speedy compiler for the Sinclair Spectrum's ZX BASIC. ZIP was designed as a tutorial in compiler design. The program was serialised in the magazine Your Spectrum.

At the Show, news of a new computer from Sinclair was circulating. A magazine was cadged from a publisher, ahead of its release date, and the glossy card folder advertising the QL fell out. Despite ridicule, he practiced typing on the embossed keys printed, full-size, on the card - a vital test of any Sinclair keyboard! He filled in the form -#406.95 for a 128K machine, delivery within 28 days.

At this point plans for a SuperBASIC compiler began. They remained plans for several months - although Simon had been able to obtain a 'provisional' copy of the QL's User Guide, neither a machine or further details were available from Sinclair Research. The delivery dates were extended, the usual excuses offered, and we were promised a 'free' RS-232 lead with our machines. Co-incidentally, the 'price' of the lead jumped up by a factor of 50 per cent - value indeed!

For several months punters and press became increasingly agitated and critical of Sinclair. The promised wonder- machine stayed under wraps. One of the QL's designers, Tony Tebby, left the firm, protesting at the premature launch of the machine, which was intended to 'scoop' the announcement of Apple's Macintosh computer. Tebby set up a firm called QJump.

Late in May Simon received his first QL, which worked tolerably well for 45 minutes at a time, before overheating and turning itself off. A month later a replacement arrived and work on a SuperBASIC compiler was started.

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.

The compiler should handle 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.

The compiler 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 the compiler had to be much more sophisticated than one for 'standard BASIC' on another micro. It seemed that the potential for 'bells and whistles' was almost limitless, but the design crystalised around a product designed and sold in three stages - a strong first launch, hopefully in good time for Christmas 1985, and two follow- up products (or bolt-on upgrades) designed to develop the potential of the design.

Most of the early work went into the first product, which was dubbed SUPERCHARGE, but the design was always shaped to allow future changes or even drastic alteration.

The other two products were eventually christened TURBO - a no-holds-barred complete re-design of the QL programming environment, built around a 'SUPER-SUPERCHARGE' compiler - and LIGHTNING - a low-cost, ultra-fast SuperBASIC subset compiler. TURBO you have now; LIGHTNING is largely coded, but needs much more work and may never become cost- effective for launch.

Back in 1984 this was only a rough plan - more immediate problems were presenting themselves. 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 interpreted program.

We approached Sinclair Research for advice and encouragement. They replied by telling us 'SuperBASIC is inherently uncompilable'. They must have thought better of this after a while, as they tried to buy our design - just in case?

Sinclair were ignored, and work on the 'parser' of the compiler soon got underway, even though the format of stored programs had to be determined by PEEKing almost at random! The parser 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. TURBO uses an extension of this idea - the T.U.G or TURBO UNIVERSAL GRAMMAR. This is handled by a sort of compiler-compiler, which generates SuperBASIC descriptions of valid syntax, to be compiled into the parser, by scanning its own English-language specification file!

An intricate expression evaluator was developed to keep track of the QL's powerful but potentially inefficient 'coercion' features - this forms the core of TURBO, SUPERCHARGE and LIGHTNING, although each version has its own variations.

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. A set of SuperBASIC utilities, later compiled, handled all the indexing and conversion. Gerry Jackson, who had just written Super Forth, was given the job of writing the code generator and utilities. 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. At this point we began to realise just how sloppy the SuperBASIC interpreter was when presented with an incorrect program - this realisation prompted the design of TURBO's 'auto-corrector' and the writing of much of this encyclopedia!

In late summer 1985 the compiler was asked to prove itself, by compiling itself into machine code. After many false starts and overnight runs, the parser ran for five and a half hours and compiled itself without finding - or assuming - any errors. The resultant intermediate file needed to be tinkered with before the code-generator would swallow it, but it was up and running a couple of days later. It compiled itself again to prove this, taking only 20 minutes now that it was running in machine-code, and became much faster as the code was 'polished'.

SUPERCHARGE version 1.09, with attendant utilities and 100- page manual, was launched in November 1985 by Digital Precision. The trade loved it and thousands of copies were sold. SUPERCHARGE was used to develop many best-selling QL commercial packages. During the development of TURBO one DP staff member even spotted a '**** Supercharged BASIC halted...' message on a British Rail departure display.

Soon after the launch of SUPERCHARGE Simon and Gerry were looking for something to do next - serious work on LIGHTNING, the low-cost top-speed entry-level compiler, occupied much of the first part of 1986. In April, after many troubles, Sinclair abandoned the computer market and sold out to Amstrad for a tiny sum; after a three-week hiatus Amstrad pronounced the QL dead.

There was chaos in the QL market for a while. Everybody wanted to carry on - for most of them had made their way despite, rather than with the help of Sinclair Research - but no one was sure whether or not the market would remain as supplies of the computer dried up. Even as we write this question is not answered. The market has survived for eight months; the crux will come when the last QLs are retailed, though Digital Precision is committed to support the QL at least until the beginning of 1988.

Against this background it was hard to decide whether to go ahead with LIGHTNING or TURBO, or to stick stubbornly to our original plans. Reluctantly, the 'cheap and cheerful' compiler had to be put on ice to ensure that TURBO - the complete SuperBASIC programming environment, or 'encyclo- compiler' could be finished as soon as was possible.

TURBO does capitalise on several features developed for LIGHTNING - in particular it uses adapted LIGHTNING code for fast string and integer handling, calls to extensions, FOR loops, very large tasks and the 'front panel' display. TURBO's floating-point maths code is based upon that of SUPERCHARGE, although it incorporates several new and useful optimisations.

The rest of the TURBO specification was honed down into two essential parts: a TOOLKIT, extending the functions and facilities of SuperBASIC, and a new PARSER, developed from SUPERCHARGE but radically different in presentation and performance. Hundreds of letters about SUPERCHARGE were scoured, to find out what users really wanted.

A public meeting of Quanta, the Independent QL User Group, in Cambridge provided the chance to ask users for their views, face-to-face. We found that 'linking' and communications between tasks were considered a high priority by most users, which explains why TURBO has the most efficient and flexible linking scheme of virtually any compiled language.

Programmer Dave Newell joined the TURBO team in the summer, to work on the TURBO TOOLKIT utilities and demonstration code. The TOOLKIT was out on time, in September 1986, but the rest of TURBO - including this manual - was sliding down the schedule, as it turned out to be even more complex than we had expected.

The TURBO development team knew, by this stage, that TURBO would have three or four times the raw 'power' of SUPERCHARGE, but they could not see how to bring this under control in a few weeks!

Most of the pressure at this point fell on Simon Goodwin, who was expected to design, adapt or write, and test the new code library (more than 7,000 intricate lines), and the TURBO parser, while simultaneously writing all the documentation. This had been almost impossible for one person during the SUPERCHARGE production phase. TURBO demanded the synchronised onslaught of several brains, so Digital Precision MD Freddy Vachha took charge.

Gerry took over responsibility for testing the library, which Simon had virtually completed by the beginning of October. Chas Dillon - a keen SUPERCHARGE user and tester - joined the TURBO team to work on the new parser and grammar, in close liaison with Simon, who continued to write the TURBO Manual.

Compared with the SUPERCHARGE development, TURBO was less of a technical thrill, because we knew that 'it could be done' - Sinclair notwithstanding! Even so, TURBO was a great challenge - it goes far beyond SUPERCHARGE in many areas, and is a much more 'complete' and coherent product. It was hard work to test TURBO because we knew that users would expect this virgin product to show all the robustness of its parent, from birth.

Most important of all, though, we knew that TURBO was probably our last big chance to shake up the QL world - it had to be absolutely, instantly, obviously right!

A peek over the horizon

The original timescale for Digital Precision's SuperBASIC compiler products stretched up to Christmas 1987. TURBO will be supported and developed through this time and beyond.

THE AUTHORS

SIMON N GOODWIN BSc developed Business and Computer Aided Design systems for three years before he went freelance. He is the author of 14 micro packages, ranging from compilers to utilities and games. He has lectured on compiler and interpreter design. Simon is a prolific communicator, author of about 150 articles for computing and electronics magazines since 1979, and an experienced broadcaster on TV and Radio. At present Simon is a 'benchtester' and 'agony aunt' for Personal Computer World, and contributes a regular technical column to the Spectrum magazine Crash. Simon considers himself a journalist, rather than a programmer.

GERRY JACKSON MSc has worked for 16 years in the field of computer electronics. He is one of the few Britons to design and make an original microprocessor chip and a 2,000 integrated-circuit minicomputer. Gerry melds theory with his practical expertise as a Computing Course Tutor with the Open University. In the field of software he has written processor simulators and implementations of the Forth programming language for the Dragon and QL computers.

FREDDY VACHHA BSc graduated from Bombay University with first class Honours in Physics and Mathemetics, rather than a degree in Chemical Engineering from the University of Bhophal, as is often assumed. He then spent two years teaching engineering and doing postgraduate mathematics research. Freddy is the author of several commercial software releases, ranging from Spectrum SUPERCODE to complex financial systems to run on larger computers. Freddy founded Digital Precision when he became bored of winning the British Mensa Chess Championship, and has been on the 'phone to the other TURBO programmers ever since.

CHAS DILLON is a computer consultant with encyclopedic experience of the industry. He left university to join Austin Motor Co as a programmer in the mid '60s, and became a consultant ten years later. While his career has developed into the areas of large systems, database design and project management, he has always retained an interest in programming and programming langugages. Chas has lectured on programming and program design. He wrote THE EDITOR and the BETTER BASIC EXPERT SYSTEM for Digital Precision.

DAVE NEWELL studies Computer Science at Wolverhampton Polytechnic, and is a Black Country institution.

BACKGROUND & FURTHER READING

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, published by McGraw Hill, is recommended to all serious SuperBASIC programmers.

There are many books on the QL's operating system, and Tony Tebby gets royalties on all except the one published by Sinclair! Andy Pennell's 'Sinclair QDOS Companion' is the best, not least because Andy has taken the trouble to explore the system and explain what he found, rather than just copy out chunks of Tony's technical notes.

TURBO

TURBO was designed by Simon N Goodwin between January 1984 and December 1986; much additional work was done by Gerry Jackson, Chas Dillon and Dave Newell. The publisher and photographer was Freddy Vachha. Production was handled by Gus Chandler. Graphics were drawn by Marilyn and Suzanne Southey.

Thanks are also due to Andy Pennell, Helmuth Stuven, Tony Tebby, Simone Vachha and many other QL enthusiasts throughout the world.

TURBO was developed on standard QL systems using disk interfaces and RAM-Plus memory made by Cambridge Systems Technology (CST) of 24 Green Street, Stevenage, Herts SG1 3DS. These add-ons are fast, reliable and recommended.

Table of Contents