TURBO TOOLKIT

THE ULTIMATE QL TOOLKIT

by
Simon Goodwin

DIGITAL PRECISION

Published by Digital Precision, 222 The Avenue, London E4 9SE
Program and Documentation © 1986 Digital Precision

QL TURBO TOOLKIT
USER MANUAL
for
Version 3.38

Turbo Toolkit updated 2000 by Mark Knight & David Gilham

updated 2003 by David Gilham

updated 2005 by David Gilham

updated 2006 by George Gwilt

Manual updated 2000 by Tim Swenson

2002 by George Gwilt

2003 by George Gwilt

2005 by George Gwilt

2006 by George Gwilt

CHAPTER SUMMARY

1. Overview & Credits

QL TURBO TOOLKIT is a library of over 100 SuperBASIC commands, functions and directives.

This manual documents the new SuperBASIC instructions. Most of these may be used in interpreted or compiled SuperBASIC programs, but some are specifically designed to be used in compiled programs. Many are compatible with Supercharge, the first SuperBASIC compiler from Digital Precision. A few make use of the more advanced facilities of subsequent compilers. Similarly, some of the commands were present in the first set of compiler extensions published with Supercharge by Digital Precision; in general these commands have kept the same names, but their performance and flexibility have been improved.

There are three versions of the Turbo Toolkit supplied, more or less functionally identical. One (filename TURBO_TK_CODE), works on all known QL and compatible systems; the other (filename TURBO_SMS_CODE) works on SMSQ/E systems only (note: NOT SMSQ or SMS2 systems). The SMSQ/E only version takes up less memory and uses some of the extended trap calls in SMSQ/E to work slightly faster and save space in some areas of its code. The third version (filename TURBO_REM_CODE) contains only those commands which can be called inside a compiled program. This version can be "included" inside a task compiled by Turbo to obviate the need to LRESPR TK Code to run the task.

Turbo Toolkit can be loaded with the following line:

```
    base = RESPR(8256): LBYTES FLP1_TURBO_TK_CODE,base: CALL base
```

or

```
    LREPSR FLP1_TURBO_TK_CODE  / LRESPR FLP1_TURBO_SMS_CODE
```

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

Remember that, as with all other add-on procedures or functions, you must install the commands before you load any program which uses them. This rule applies whether the program using the commands is compiled or written in boring old ordinary SuperBASIC.

Default options

TURBO TOOLKIT is stored in RAM, rather than ROM and the fact that it is held in memory means that you can 'tailor' many of the default values it uses to suit your system.

The 'default' values are used by Toolkit commands if no explicit information is provided when the command is entered. In particular, you can set:

1. The default device name (normally FLP1_); this is added at the start of names supplied to CHARGE, LINK_LOAD or EXECUTE, unless they already begin with a device name. Compiler overlays are also read from this device.

2. The default buffer size used by EDIT (normally 40 characters).

3. Whether or not EDIT produces a sound after every error.

4. The length of pipes used to link tasks by EXECUTE (normally 200 bytes).

5. The key or keys used to abort CHARGE and EXECUTE_A (normally ALT SPACE).

The bulk of this manual consists of a discussion of the new commands, directives and functions in small groups, collected by category, with a list of names at the start of each section.

TURBO TOOLKIT has been written by a programmer, for programmers. The new facilities are concise, powerful and general. The examples will indicate some of the things you can do with TURBO TOOLKIT, but there are thousands of other possibilities. Experiment!

Credits

My thanks go to Chas Dillon for advice and encouragement in the development of this toolkit, and to Tony Tebby and Andy Pennell for help and information.

- Simon N Goodwin

Turbo Toolkit is Freeware

2. Channel Manipulation Commands

CHANNEL_ID, SET_CHANNEL, CONNECT, FWINDOW%

The QL contains facilities to handle 'pipes' - queues of characters maintained in memory by the system, where the user can either put characters into the pipe or take them out, and the first character put in is always the first to come out. The channel which puts characters into the pipe is termed the output pipe and the channel that reads characters is the input pipe.

These pipes can have any length from 1 to 32767 characters, and are often very useful when a program needs a temporary buffer. Unfortunately they have not been available to the BASIC programmer, because the standard SuperBASIC OPEN command is not sufficiently sophisticated to allow the user to specify which channel a given input pipe is to take characters from. A new command, CONNECT, solves this problem.

Another problem with channels comes when two or more tasks are linked and running concurrently. It is often useful for one task to use a channel opened by another - yet every compiled SuperBASIC program has an independent channel table, so channel numbers do not correspond from one task to the next.

CONNECT

CONNECT works rather like OPEN except it expects TWO channel numbers -the output and input pipes, respectively.

The first channel must previously have been opened to an output pipe, e.g:

```
    OPEN #4,PIPE_500
```

Any integer from 1 to 32767 can follow the underscore character - the number determines the length of the pipe.

You may now PRINT characters to channel 4; they will be stored in the pipe until it is full; once it is full no further characters will be accepted until some characters are taken out from 'the other end' of the pipe. To allow this, and to specify the channel from which characters will emerge from the pipe, Type:

```
    CONNECT 4 TO 3
```

You can put hash signs in front of the channel numbers if it makes you feel happy - they're not necessary. The above command would allow characters PRINTed from channel 4 to be read, in their original sequence, from channel 3. If channel 3 is already open, CONNECT closes it before linking it to the pipe.

To test CONNECT, type these commands:

```
    PRINT #4;"HELLO PIPE.": INPUT #3;A$: PRINT A$
```

The first command sends a message into the pipe. The second reads it into the variable A$, and the third prints the result. Unless something is very wrong with your QL, the message should be the same before and after 'piping'!

If a pipe is full, a task that tries to write characters into it will pause until there's room. If a pipe is empty a task that tries to read characters will just wait until there is something to be read. The EOF function works with pipes, but you must CLOSE the output pipe before the input channel can detect the end of the file - otherwise there would be no distinction between the end of a file and a temporary break in the stream of data.

There is one irritating but non-crucial implementation restriction upon CONNECT. The input channel number must not be the highest channel number yet used in BASIC, or you will get a CHANNEL NOT OPEN error. You can get around this in two ways. Either make sure that you 'leave a gap' for that channel when you open the output pipe, or use a channel that has previously been opened for another purpose.

CHANNEL_ID

This function allows a task to find the operating system's internal identifier for a channel. This 32 bit identifier is returned as a floating-point value, but may be stored in a long word - with POKE_L, for instance. The channel identifier can be passed to another task to allow pipes to be set up or routines in the second task to PRINT to or INPUT from that physical channel.

The function expects one numeric parameter - the SuperBASIC channel number of the channel which must be identified. That should be the number of an OPEN channel, for obvious reasons.

SET_CHANNEL

The procedure SET_CHANNEL has the opposite purpose - to associate a channel ID with a BASIC channel number. In case you're getting confused, here's a rather trivial example that allows channel 3 to be used for input and output just as if it were channel 0, the command channel.

```
    X=CHANNEL_ID(#0) SET_CHANNEL #3,X
```

Subsequent PRINT #3s and INPUT #3s will work just like PRINT #0s and INPUT #0s. An attempt to CLOSE channel 3 would close channel 0 'as well', since they both correspond to the same hardware channel as far as the operating system is concerned. In fact you should NEVER close the SuperBASIC interpreter's channels 0 and 1, because parts of the operating system assume that they will always be open and (therefore) have fixed identifiers. Unfortunately the ROM does not check to stop you doing this, and it is easy to accidentally 'hang' the machine as a result. Remember that OPEN and CONNECT may both perform an implicit CLOSE.

In general you get a weird message or your machine crashes if you use a channel once another with the same identifier has been closed. This kind of thing makes the QL very upset.

For obvious reasons, SET_CHANNEL works with system channel identifiers, rather than SuperBASIC ones. The interpreter associates some information about channels with specific channel numbers; this information is not copied by SET_CHANNEL. When using one channel with an identifier copied from another, you must not make any assumptions about:

1. The current graphics coordinates.
2. The turtle angle or pen position.
3. The cursor position and line width, for non-console channels.

This information will be stored separately and independently for each SuperBASIC channel number.

As with CONNECT, there is a small implementation restriction on SET_CHANNEL. The channel number you SET must not be the highest channel number yet used in BASIC, or you will get a CHANNEL NOT OPEN error. You can get around this in two ways. Either make sure that you 'leave a gap' for that channel when you open some other channel, or use a channel that has previously been opened for another purpose. If the channel is already open, SET_CHANNEL closes it before associating it with the new identifier.

FWINDOW%

Function to help write programs which will work on a standard QL screen but which can also use high resolution screens. This function takes the same parameters as WINDOW though the channel parameter is not optional. Returns an error code if the window can't be redefined, otherwise redefines it just as WINDOW does and returns 0. Enclosed is a suitable SuperBASIC program (ReSize_BAS), a smaller example follows:

```
1000 WindowError=FWINDOW%(#0,512,384,0,0)
1010 IF WindowError<0 THEN
1020   WINDOW#0,512,256,0,0
1030 END IF
```

3. Random Access File Handling

POSITION, SET_POSITION, GetHEAD, SetHEAD

Two commands allow you to read or write any part of a file without having to read past the rest of the file. Whole files can be accessed 'at random' - treated like enormous character arrays. Two other commands allow access to the file header.

POSITION

This function returns the current position within a specified channel, e.g. PRINT POSITION(#3). The channel number must be prefixed with a hash character, and the associated channel must be open to a file or pipe, or a bad parameter error will occur. The first 'position' in a file (as opposed to a Kama Sutra) is position 0.

SET_POSITION

This command requires two parameters: a channel number (preceded by a hash) and a position. It attempts to set the position within the channel to the value specified.

If the required position is beyond the end of the file - e.g. SET_POSITION #3,1E9 - the position is set to the end of the file. If the parameter value is 0 or less, the position is set to the start of the file.

GetHEAD & SetHEAD

The command GetHEAD lets you read a file header. For instance, it lets you check the file length and the dates when the file was last read, changed or copied. SetHEAD lets you change the dataspace of a task, or the file-type or other 'reserved' information.

The commands have two parameters - a channel number, and the address of a 'buffer' - an area of memory where the file header can be stored.

Before you can use the commands you must open the relevant file in the normal way. If you've got a toolkit you can check that the file exists first, using a function like DEVICE_STATUS or FOPEN.

You can reserve the buffer - an area of 64 otherwise-unused bytes of memory - with ALLOCATION.

As is often the case on the QL, it's easier to read information than it is to change it. For some reason best known to QDOS designer Tony Tebby the QL FS.HEADS routine only re-writes the first 15 bytes of a header, and the length of a file is always reset when the file is closed, so you can't rename a file or set the dates this way.

SetHEAD does let you alter the dataspace of a task easily. The old way to do this was to load the whole file with LBYTES, DELETE it and re-save it with SEXEC. SetHEAD does the job much faster and more economically.

You can also use SetHEAD to hide information in the 'access', 'type' and 'extra' slots. Just read the header with GetHEAD, POKE the buffer with new values, and store them with SetHEAD. Don't forget to CLOSE the file when you've finished!

4. Task Control

LIST_TASKS, SET_PRIORITY, REMOVE_TASK, RELEASE_TASK, SUSPEND_TASK

LIST_TASKS

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

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

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

```
    LIST_TASKS #0
```

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

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

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

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

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

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

The last character is a full stop, if the task is ready to run normally, or a letter "s" if the task has been suspended. Tasks may be suspended by QDOS while they are waiting for data, or by the user for any reason.

Priority treatment

Priority numbers range from 0 to 255. If a task has a priority of 0 it never gets any time at all. If a task has any other priority, the proportion of the processing time it receives will depend upon the priority of other tasks.

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

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

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

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

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

The 'priority' of QL tasks is much like the 'priorities' which you might attach to tasks at home. Fixing the gas fire might be a high priority, hoovering the carpet a lower priority and experimenting with your QL the lowest priority of all, only to be done when other tasks are not pressing. Unfortunately the author only attends to the fire when he would otherwise be suffocated or poisoned by fumes. This is not, in itself, a good policy, but it's brought you TURBO as well as TURBO TOOLKIT, so it is evidently a viable scheme of priorities!

SET_PRIORITY

This command allows you to change the priority of any task that is loaded.

The QL needs two things in order to change the priority of a task - the task identifier (the number and the tag) and the new priority. Priority values may range from 0 to 255 on QDOS and -128 to 127 on Minerva and SMSQ/E.

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

The format of the SET_PRIORITY command is shown below:

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

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

There are times when it is useful for a task to be able to set its own priority. Any program can do this by using the SET_PRIORITY command with just one parameter - the new priority. Thus:

```
    SET_PRIORITY 1
```

sets the priority of the task that executes the command to 1.

REMOVE_TASK

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

```
    REMOVE_TASK 1,1
```

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

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

SUSPEND_TASK

Sometimes it is useful to be able to 'put a task to sleep' for a while. The command SUSPEND_TASK does just that. Normally it has three parameters: a pair of integers, together making a task identifier (as with SET_PRIORITY or REMOVE_TASK), and a third integer number indicating the amount of time for which the task is to remain dormant.

This value is in units of one display 'frame' time - the time taken for the video electronics to generate a complete (but not interlaced) picture. The same units are used to specify delays generated by the PAUSE statement. This time generally corresponds to the rate of alternation of the mains electricity supply. In the UK and most of Europe this time is a fiftieth of a second; in the USA and some other countries it is 1/60 second.

As with SET_PRIORITY, you may use the SUSPEND_TASK command with a single number, in which case the period of suspension is assumed to refer to the task executing the command. You can check the frame time of your QL by entering and timing this direct command:

```
    SUSPEND_TASK 600
```

If the computer pauses for 12 seconds your display is re-drawn 50 times a second. A ten second pause indicates a frame time of 1/60 second.

It is not wise to try this experiment while other tasks are running, because the command merely sets a minimum time for which the task will remain dormant. At the end of that time the computer will treat that task as if it is competing for time just like any other task. Tasks don't necessarily start to run as soon as their period of suspension is over - it depends what else is going on.

You can put a task to sleep 'for ever' by specifying -1 as the length of pause. The task will not start to run until another task explicitly 'releases' it from suspension. The command to do this will be discussed in a moment.

There is one special rule about task 0,0, the SuperBASIC interpreter. The BREAK keys - CONTROL and SPACE - will always bring that task back to life if it is suspended, regardless of how long it was meant to wait. This is the mechanism that lets you break into SuperBASIC programs. The TURBO TOOLKIT includes a command, EXECUTE_A, which lets you break into any other task in a similar way.

RELEASE_TASK

This command is the complement of SUSPEND_TASK. It expects two parameters - a task identifier - and releases the specified task. You'll get an 'Invalid Job' error if the numbers you type do not correspond to a valid task.

5. Cursor Control

CURSOR_ON, CURSOR_OFF

Two commands are provided to turn on and off the display of a cursor in a CONsole window. Thus a window can be selected even if INPUT is not taking place. This is most useful when INKEY$ must be used from within a multi-tasking program.

CURSOR_ON

This command turns on the cursor in the specified channel. The channel number must be prefixed with a hash character, if present: the default is channel 1. N.B: unlike commands in other toolkits, CURSOR_ON always causes the chosen cursor to flash; it doesn't just turn on a static cursor.

CURSOR_OFF

This command turns off the cursor in the specified channel.

6. Error Detection and Trapping

DEVICE_SPACE, DEVICE_SPACE, WHEN_ERROR, END_WHEN, RETRY_HERE, ERLIN%, ERNUM%, DEBUG, DEBUG_LEVEL

Effective error trapping is a vital feature of any interactive program, yet it is a feature that has been neglected in the implementation of SuperBASIC. Late models of the QL have a set of undocumented commands which deal with error trapping, but these are extremely unreliable and unavailable on most QLs. Nor have they been formally specified or documented; Sinclair Research apparently persuaded Jan Jones, the author of the interpreter, to take their details out of her excellent 'QL SUPERBASIC DEFINITIVE HANDBOOK' (ISBN 0-07-084784-3).

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

A second function, DEVICE_SPACE, lets programs check whether or not there is room for data as they write.

DEVICE_SPACE

This function expects one parameter - the number of a channel open to any file on the medium (floppy disk, microdrive, or whatever). The channel number may be preceeded by a hash character. Typically the parameter will be the channel number of an output file. The result of the function is the number of unused bytes on the medium.

DEVICE_SPACE can be used as information is written to a medium, or as a check for possible errors before a file is created. NOTE: 64 bytes are used to store the 'header' of each new file. Some space for 'directory' information may also be allocated when a new file is created.

DEVICE_STATUS

This function expects one or two parameters - an integer access-type (the default is 2) and a mandatory file or device name. It returns a negative value if there is some problem opening a correspo0nding channel. Otherwise it returns zero or a positive value - usually the number of 'free' bytes on the device. Serial devices, which have no real 'capacity', pretend that they are the same size as the largest possible QDOS devic - almost 33.6 Megabytes!

The exact treatment of your file or device name depends upon the access-type parameter, which should indicate what you want to do with the device or file.

```
    DEVICE_STATUS ( 0 , name$ )
```

The 0 shows that you want to open, read and alter data in the file or device. The result will a negative number if this is not allowed; otherwise it will be the number of unallocated bytes on the device.

```
    DEVICE_STATUS ( 1 , name$ )
```

Access-type 1 indicates that you just want to read data from the file or device. The result will be a negative number if this is not allowed; otherwise it will be the rather irrelevant number of free bytes of space on the device.

```
    DEVICE_STATUS( 2 , name$ ) or DEVICE_STATUS ( name$ )
```

Access-type 2 shows that you want to create a new file or open a new device. The result will be a negative number if this is not allowed; otherwise it will be a positive value - the number of unallocated bytes on the device.

```
    DEVICE_STATUS ( -1 , name$ )
```

This is the most versatile of all. The function analyses the supplied string to find out whether or not it starts with the name of a device on the current QL. Any parameters, such as 'con_448x180a32x16', ser1EHC' or a file name are checked. If everything looks good DEVICE_STATUS tries to open the file and re-write part of it, without corruptiing the contents. If the file does not exist, DEVICE_STATUS tries to create it. If this succeeds, the function deletes the resultant file and returns with the number of free bytes on the medium, after allowing header and directory space for the 'empty' file. A negative number is returned if anything goes wrong.

DEVICE_STATUS automatically adapts to different hardware, so you can use it on a basic QL system secure in the knowledge that it will also work with floppy disks, modems, hard drives, 'parallel'printers and so on. For example, this command shows that the file 'TURBO_TASK' exists on the write-protected disk in drive 1:

```
    PRINT DEVICE_STATUS(-1,"flp1_turbo_task")
```

produces the result:

```
    -20
```

It is worth bearing in mind that DEVICE_STATUS takes a long time to get results from the network driver if there's nothing connected. This is inevitable; if it bothers you, trap it before the call with;

```
    IF NAME$(1 to 3)=="NET":RETurn -7:ELSE RETurn DEVICE_STATUS(1,NAME$)
```

The meanings of the various returns from DEVICE_STATUS are shown below:

TABLE OF VALUES RETURNED BY DEVICE_STATUS

Value	Meaning
0 or more	The device exists, and is not busy; a file with the name specified (if any) does not yet exist. The name or other parameters (if any) are valid. The value is the number of free bytes on the device, or a very large number for 'endless' serial devices.
-3 or -6	The device name and parameters are valid, but the QL has insufficient free space to open a new channel to the device.
-7	There's no device with that name on this QL.
-8	A file with the name specified exists on the device, and may be read, written or deleted.
-9	EITHER the device exists, but it is already in use and no other task may use it until the present one has finished; OR the file is being written.
-11	The specified device is full.
-12	The device name is valid, but the file name or parameters are not.
-16	Bad or changed medium; the medium in the device is faulty, or has been changed while the system was updating or writing to a file.
-20	The specified file exists and may be read but not altered, because the device is write-protected or the file is being read.

WHEN_ERROR, END_WHEN, RETRY_HERE, ERLIN% and ERNUM%

These commands and functions allow asynchronous error-trapping routines to be declared in appropriately compiled programs. See section 4 of the QL Turbo Manual for a complete discussion.

DEBUG and DEBUG_LEVEL

Compiler directives used to delimit parts of a program so that code can be included or excluded conditionally. The use of these commands is detailed in TurboS4.Txt.

7. Task and Compiler Invocation

DEFAULT_DEVICE, CHARGE, EXECUTE, EXECUTE_A, EXECUTE_W, OPTION_CMD$, LINK_LOAD SNOOZE, COMPILED, CATNAP

DEFAULT_DEVICE

All file names used with CHARGE, LINK_LOAD or the various kinds of EXECUTE can use the 'default device'. If you don't specify a device at the start of a name, the commands will automatically put the 'default device' name there. If Toolkit II, which automatically sets the default device, is not operational when TURBO TOOLKIT is loaded, the default device is set from a string within the TURBO TOOLKIT. This is initially 'FLP1_', but it can be changed by using the Default Editor in UTILITY_TASK, to another five-character string.

You can at any time change the default by typing a new string as a parameter of DEFAULT_DEVICE:

```
    DEFAULT_DEVICE "win1_sys"
```

NOTE. DEFAULT_DEVICE is equivalent to Toolkit II's PROG_USE.

CHARGE

A command to invoke Digital Precision SuperBASIC compilers. CHARGE must be entered from SuperBASIC, or the Master Basic with SMSQ. If a string parameter is supplied it is treated as the name of the task file to be produced, although SUPERCHARGE ignores this. CHARGE puts the default device name at the start of the parameter unless there's already a device name there.

If CHARGE is used with backslash as parameter (CHARGE \ ), Parser_task immediately starts just as if SPACE had been pressed when COMPILE is selected on the front panel. This only works if all the settings as displayed on the panel are acceptable.

EXECUTE

A much-improved implementation of the EXEC command. Several tasks compiled by Turbo may be specified in one EXECUTE command; 'pipes' (communications channels) are set up to allow the tasks to communicate via PRINT and INPUT. The priority of each task may be set individually as it is invoked. You can pass a different 'parameter' string to each task.

The simplest form of the EXECUTE command is like EXEC - it has just one parameter - a task file name:

```
    EXECUTE TASK
```

Optionally, you may follow any task name with an exclamation mark and an expression giving a priority (between 1 and 127) for the task. The priority of a task determines the proportion of available processing time that may be spent executing that task. The default value is 32. This command starts a task with a priority of 16:

```
    EXECUTE TASK ! 16
```

You may also associate an 'option' string of up to 32,760 characters with a task. The task will be able to read that string when it has loaded. The string or expression should follow the task name and be prefixed by a semicolon:

```
    EXECUTE DEMO ; "STRING"
```

You may set a priority for a task and specify an option string in either order:

```
    EXECUTE DEMO ; "STRING" ! 16
    EXECUTE DEMO ! 16 ; "STRING"
```

Whenever a task name appears in later examples, remember that it may be associated with a priority, or an option string, or both.

EXECUTE also allows you to specify and link several tasks:

```
    EXECUTE TASK1 TO TASK2
```

In this case a 'pipe' will be set up, linking the two tasks. Whatever the first task PRINTs on its highest valid channel number (31 in compiled BASIC) the second will be able to read on the previous channel (30 in compiled BASIC). Note that this feature only works with machine-code that expects the pipe linkage - it does not work with SUPERCHARGEd code.

Pipes are, by default, 200 characters long but you can change this length with the UTILITY TASK. Pipes work VERY quickly.

Pipe linkages can be extended to 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:

```
    EXECUTE TASK1 TO TASK2 TO TASK3 TO TASK4
```

The first task in the list 'owns' all the others. If it stops or is removed, the others will also stop. It is the responsibility of the tasks to close pipes or other channels when they stop; tasks generated by Turbo always 'tidy up' in this way.

You may start the list with a data file name, rather than a task file; the computer will open the file for input so that the first task in the list can read it as if it were a pipe.

You may also specify a SuperBASIC channel number at the start of the list, in which case the existing channel (if any) with that number is closed, and a pipe will be opened in its place. Material that is output by SuperBASIC to that channel will be received by the first task in the list. The channel number must not be the highest yet encountered by SuperBASIC, or a 'CHANNEL NOT OPEN' error will occur.

Finally, and consistently, the last task in a list may be replaced by a channel number or a the name of a file that does not yet exist, in which case the task will be able to write directly to that channel, or the file will be opened for the task to use. You will get a 'bad parameter' error if the file already exists. For instance:

```
    EXECUTE LISTFILE TO PAGINATOR TO #4
```

This will send the contents to the text file LISTFILE to a task called PAGINATOR, which will do the obvious and route output to (previously opened) SuperBASIC channel 4.

Alternatively we could paginate material as it was PRINTed to channel 3 by SuperBASIC, and output it directly to the serial port:

```
    EXECUTE #3 TO PAGINATOR TO SER1
```

If we wanted the text in capital letters as well, we could add an extra task, or 'filter', called CAPSLOCK:

```
    EXECUTE #3 TO PAGINATOR TO CAPSLOCK TO SER1
```

The possibilities for modular programming go on and on!

If EXECUTE has no parameters it looks for the last task compiled by TURBO and runs that. This is useful if you have just compiled a task and want to run it straight away. This feature applies also to EXECUTE_A and EXECUTE_W.

The complete form of EXECUTE is:

```
    EXECUTE [ch TO] prog_spec {TO prog_spec} [TO ch]
    ch = #channel | filename (type 0)
    prog_spec = prog_name [ option ]
    option = ! pr [; strg ] | ; strg | [! pr ]
    [] means 'optional' , {} means 'repeated optional' and | means 'or'
```

Machine code and EXECUTE

EXECUTE may be used with machine-code tasks that contain code to fetch channel details and parameter strings. When a task is loaded data will be placed onto the A7 stack. The first (lowest address) word contains the number of channels previously opened, followed by one long word QDOS channel ID for each channel. After this comes the option string: a word length followed by the corresponding number of characters.

The channel-handling and parameter access facilities are only available to programs containing appropriate code. Programs compiled with Version 1 of Supercharge should be re-compiled by Turbo if you wish to take advantage of the new features.

It is possible to use EXECUTE on machine code programs other than those compiled by Turbo. In this case there are severe restrictions. The forms of EXECUTE allowed are:

```
    EXECUTE prog_name, #channel
    EXECUTE prog_name option (where 'option' has the meaning above)
```

In the first case, two "channels" are put on the stack. The first one is the ID of #channel and the second is a negative number.

In the second case no channels are put on the stack.

Also, unlike EX, EXECUTE does not increase the dataspace to accommodate the extra lengths of channels and strings. Hence, care should be taken not to give a parameter string which is longer than the job's dataspace.

EXECUTE_W

This is the equivalent of the standard EXEC_W command, but you can use all the parameters allowed by EXECUTE. The task performing the EXECUTE_W - usually the SuperBASIC interpreter - is suspended until the command, and any associated processing, has finished.

EXECUTE_A

This command is identical to EXECUTE_W except that certain keys (normally ALT and the SPACE bar) are checked five times a second; if the keys are pressed the command is immediately aborted, stopping with a NOT COMPLETE error. Note that if the command had been given from a daughter basic under SMSQ the daughter basic will be aborted as well as the program which was the subject of EXECUTE_A.

OPTION_CMD$

This function returns a null string if called from SuperBASIC or Supercharged programs; if used in programs processed by later compilers it returns the option string passed to that task as a parameter of EXECUTE.

LINK_LOAD

This command loads several tasks which may share variables, procedures and functions. The GLOBAL and EXTERNAL commands specify which items are shared; these commands are explained later.

Not all compiled programs support the sophisticated LINK_LOAD communication mechanism. If your compiler allows LINK_LOAD it will say so in the manual. You will get a BAD PARAMETER error if you try to load inappropriate tasks with LINK_LOAD.

LINK_LOAD must be followed by a list of task names; the default device name is added to each name, unless a device is explicitly specified, and each task is loaded into memory. LINK_LOAD uses the option string to tell each task where to find the others, so that they can communicate through shared variables, procedures and functions.

Each task invoked by LINK_LOAD must have a different module number -set with the GLOBAL command, as explained later.

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.

LINK_LOAD_A and LINK_LOAD_W

These operate in the same way as LINK_LOAD except that the calling task is suspended until the operation is complete as with EXECUTE_A and EXECUTE_W.

SNOOZE

This command indicates that a compiled module is to stop independent execution so that another, linked module can call procedures or functions declared therein. If a module tries to call routines in another that is not 'snoozing' it will wait; it is not possible for several tasks to execute the same code concurrently, unless they each have their own copy of the code. If used in SuperBASIC, SNOOZE suspends the interpreter until CTRL and SPACE are pressed.

COMPILED

This is a function which returns 1 if a program is compiled and 0 if the program is being interpreted. Thus, for example, a program can either use the option string or prompt for input, depending upon whether it is interpreted or EXECUTEd:

```
IF COMPILED THEN N$=OPTION_CMD$: ELSE INPUT "Name ";N$
```

Similar code can be used whenever you want a program to behave in different ways when compiled or interpreted; this is often useful when you are testing a program alternately under the interpreter and compiler - for instance, you could truncate or avoid a slow initialisation sequence when using the interpreter.

OPTION_CMD$

The OPTION_CMD$ function returns a null string if called from SuperBASIC programs; if used in programs processed by TURBO, it returns the option string passed to that task as a parameter of EXECUTE.

CATNAP

The CATNAP command has been specially developed for imaginative programmers as a refinement of the SNOOZE command, which we found necessary if a routine is to be entered from another module.

SNOOZE stops a task dead; it will never come to life again, although its code may be used many times by several other 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 the CATNAPping task 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 'forever', or just CATNAP again, to wait for the next call.

Here is a simple example. 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 count
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 complex uses of CATNAP are possible, but these depend upon your system design and the imagination with which you use TURBO.

8. Editing Data on the Screen

EDIT$, EDIT%, EDITF

Three functions can be used as an improvement upon the normal INPUT statement. The maximum length of input may be specified, allowing neat display layouts, and a 'default' value may be supplied. The user can edit this default as if it had been entered 'manually'. Invalid data cannot be entered.

EDIT$

This function can have up to three parameters. The first, optional parameter is the channel number. It must be prefixed with a hash, if it is present: the default is channel 1. Then comes the 'default text' - a value, of any type, which is printed for the user to edit. If no default is required a null string ("") must be specified. The final, optional parameter is an number - the maximum length of the entry. The *default (which can be changed by UTILITY_TASK) is 40 characters. Allow room for one extra character on the screen; remember that the cursor may be moved to the space after the end of the text.

A BUFFER FULL error will occur if the maximum length specified exceeds the capacity of the SuperBASIC area BUFFER. In a compiled task, all free space is used for the buffer, so problems can be avoided by allocating an appropriate amount of dataspace. The interpreter allocates sufficient buffer space to hold the longest line previously listed or edited, or a minimum of 118 characters.

The left and right arrow keys can be used to move up and down the line, in the usual way. The Up and Down arrow keys are trapped as an error - and move the cursor to the end of the line. So does any attempt to enter more characters than will fit into the field. This draws attention to the fact that you must delete some characters before you enter more.

EDIT$ returns a string - the edited text - when you type ENTER.

A warning sound can be produced if an error occurs when using these functions - if you press a vertical arrow key, fill up the buffer or try to ENTER an invalid number. If you don't want the sound you can turn it off by 'patching' the extensions with the UTILITY TASK. You can also change the default maximum length of entries.

EDIT%

This function works like EDIT$, but the result is automatically converted into an integer - a whole number between -32767 and 32767. When you press ENTER the text is checked. EDIT% will not stop editing and return a value unless the text is a valid integer.

An example of EDIT% is:

```
    X% = EDIT%(#0,X%,6)
```

EDITF

This works like EDIT%, except the result is a floating-point number. Up to 9 digits, with or without an exponent, may be entered.

9. Binary Input and Output Functions

FLOAT$, INTEGER$, STRING$, GET%, GET$, GETF, INPUT$ LONGINTEGER, LONGINTEGER$, STRING%, STRINGF

It is often useful to be able to store numbers in a file, using a fixed number of bytes for each value; you can then jump directly to a given value using SET_POSITION, without the need to read intervening data. Only two bytes are needed to hold an integer between -32768 and 32767, and six bytes will accommodate a nine-digit floating-point value between (roughly) -1.61E616 and 1.61E616. The only snag of binary storage is that it can be hard to recover the data in a file if an error occurs - but that should not be a problem if you keep adequate backup copies of important information.

It is not efficient to file strings in a fixed-length form, but it can still be useful to be able to store a string preceded by its length, so that code to read it can reserve buffers or handle the string piecemeal if need be. The new functions also allow this.

FLOAT$

This function accepts a single floating point value and returns a six-character-long string representing the internal form in which that value would be stored. This can then be PRINTed to a file and re-read using GETF.

You should put a semicolon at the end of the PRINT statement, or a redundant line-end marker will also be transmitted - if you're not careful this could mess up the next field in the file.

INTEGER$

This function accepts an integer between 32767 and -32768, and returns a two-character long string representing the internal form in which that integer value would be stored. This can be PRINTed and re-read with GET%.

STRING$

This function accepts a string of up to 32762 characters and returns a string representing the internal form in which that string value would be stored. This can then be PRINTed to a file and retrieved with GET$. The string is NOT 'padded' to an even length:

```
    PRINT STRING$(TEXT$);
```

is the equivalent of

```
    PRINT INTEGER$(LEN(TEXT$));TEXT$;
```

Binary input

Four functions allow binary values to be read from a channel. These functions resemble Tony Tebby's GET command (built into some disk systems) but they return values as functions, whereas GET modifies its parameters and is thus incompatible with TURBO. PUT works OK in compiled programs, but it can often be re-written more efficiently by appending and PRINTing the results of the binary conversion functions FLOAT$, STRING$ and INTEGER$.

NOTE. Turbo v4.21 onwards allows machine code extensions to alter their parameters so GET can now be used in a program compiled by Turbo.

GET%

A function to read binary integers from a channel. PRINT GET%(3) reads 2 bytes from channel three and returns a corresponding integer value. A hash before the channel number is optional.

GETF

A function to read binary floating point numbers from a channel. PRINT GETF(3) reads 6 bytes from channel three and returns a corresponding floating point value. An overflow error is generated if the bytes do not represent a valid floating-point number.

GET$

A function to read binary strings from a channel. PRINT GET$(3) reads 2 bytes (the length of a string) from channel three, and then reads and returns that number of characters from the file.

INPUT$

This function expects two parameters - a channel number and an integer. INPUT$ attempts to read the number of characters specified by the integer from the channel. The default is channel 1. No cursor is displayed and the function will wait indefinitely if the characters are not available. You may find it helpful to know that:

```
    GET$(channel)
```

is shorthand for

```
INPUT$(channel,GET%(channel))
```

LONGINTEGER

Converts any string of 4 bytes into a floating point number equal to the signed long integer value of the bytes. The string must be 4 bytes long or an error message will result. Example:

```
x=LONGINTEGER(Any$)
```

LONGINTEGER$

Function to convert any floating point number in the long integer range (-2,147,483,648 to 2,147,483,647) into a string of 4 bytes. This function and the previous one are handy for putting long integer values into strings or files when database programming; example:

```
Any$=LONGINTEGER$(x)
```

Binary random access - an example

These two procedures allow you to read and write records consisting of two integers and a floating-point value. The records are stored in a file, and numbered from zero. You can read or write any record without having to read intermediate ones. Each record occupies 10 bytes - two for each integer and six for the floating-point value.

Unlike an array, there is no limit on the number of records in a file other than the capacity of your drive; the file is not kept in memory so that the data does not encroach upon the memory free to hold your program.

Values are taken from or stored in three variables: INT1%, INT2% and FLOAT$. The file should be opened for reading and writing on channel CHANNEL. If you try to write a RECORD number beyond the current end of the file the routine will extend the file with zero entries, up to the point where the new record should go.

```
DEFine PROCedure READ_RECORD(record)
  IF record <0 THEN PRINT "Before start of file!":STOP
  SET_POSITION #channel,record*10
  IF EOF(#channel) THEN PRINT "After end of file!":STOP
  INT1%=GET%(channel):INT2%=GET%(channel):FLOAT=GETF(channel)
END DEFine READ_RECORD

DEFine PROCedure WRITE_RECORD(record)
  IF record <0 THEN PRINT "Before start of file!":STOP
  SET POSITION #channel,record*10
  IF EOF(#channel) THEN
    REPEAT extend_file
      IF DEVICE_SPACE(channel)<10 THEN
        PRINT "Device full.":STOP
      END IF
      IF POSITION(channel)=record*10 THEN EXIT extend_file
      PRINT #channel;INTEGER$(0) & INTEGER$(0) & FLOAT$(0);
    END REPEAT extend_file
  END IF
  PRINT #channel;INTEGER$(INT1%) & INTEGER$(INT2%) & FLOAT$(FLOAT);
END DEFine WRITE_RECORD
```

STRING%

Returns the numeric decoding of a two-character string, i.e. 256*CODE(char1$)+CODE(char2$) A 'bad parameter' error is generated if the string is not two characters long.

```
w%= STRING%(two_chars$)
```

STRINGF

Returns the floating-point number which would be encoded internally as the six-character string text$.

```
f= STRINGF(text$)
```

10. Memory Management

ALLOCATION, DEALLOCATE, MOVE_MEMORY, PEEK$, POKE$, SEARCH_MEMORY, POKE_F, PEEK_F, SYS_VARS

SuperBASIC has few facilities to handle areas of memory. Eight extensions cure this weakness, letting you manipulate memory 'en bloc' and within strings.

PEEK$ and POKE$ can be used to generate pop-up (transient) windows, graphic effects, or to process memory contents. Strings copied from RAM can be searched, edited and replaced, for example. MOVE_MEMORY is also useful in display and 'raw data' processing applications.

SuperBASIC includes a function, RESPR, which finds and reserves an area of memory of a specified size. But there's no command to de-allocate this space (as a whole or in sections) and RESPR does not work while tasks other than SuperBASIC are running. ALLOCATION and DEALLOCATE do not have these problems.

ALLOCATION

This new function reserves a specified number of bytes for the user. The space is taken from the QDOS 'heap' - at the bottom of the QL memory map - so that the command works perfectly when tasks are running. Indeed, ALLOCATION may be performed within a running task. In that case, by default, the space will be automatically released when the task terminates. If you want the space to persist you should specify an 'owner' for it, using the pair of numbers shown by LIST TASKS. These values should follow the number of bytes, e.g:

```
    space = ALLOCATION ( num , task , tag )
```

Memory is only released if the specified owner task terminates, OR the DEALLOCATE command - explained in a moment - is used to release that specific area. N.B. task 0,0 (SuperBASIC) never terminates, so memory owned by that task remains allocated until it is explicitly released.

The result, SPACE, is normally a positive value - the address of the start of the area - just like the result produced by RESPR. However SPACE is set to -2 if the task (if specified) does not exist, and -3 if NUM bytes of contiguous (adjacent) free memory cannot be found. These negative values are QL internal codes for 'Invalid job' and 'Out of memory'. If you have a JS or MG version of the QL you can print an appropriate message with the REPORT command, e.g:

```
    REPORT space
```

DEALLOCATE

This procedure expects one parameter - the address of an area of memory which is to be released. That address should have been returned by a call to ALLOCATION. DEALLOCATE releases the number of bytes requested in that specific ALLOCATION.

How to avoid storage fragmentation

You should de-allocate memory in the opposite order to that in which you allocate it ('last in, first out') so that all the unused space is contiguous at any time; thus you will always be able to access the largest possible block of memory should you wish to. QDOS can join two adjacent 'free' blocks into one larger free area, but it cannot amalgamate blocks unless they are adjacent.

Memory is also implicitly allocated when a channel is opened, FILL is used, or a disk is used for the first time. You should try to make sure that this happens before you call ALLOCATION, or the area allocated by the system may be 'stranded', dividing the available space into two 'holes', when you release the memory you first used.

The following diagrams show how space can become fragmented. Step 1 shows the situation before any 'heap' space is allocated. All the 'heap' memory is unused.

```
                 **********************************************
    STEP 1       *                                            *
                 **********************************************
```

Then you ask for an allocation; the space is taken from the start of the unused area:

```
                 **********************************************
    STEP 2       *  YOUR SPACE  *                             *
                 **********************************************
```

If the system (or another task) now asks for space this situation results:

```
                 **********************************************
    STEP 3       *  YOUR SPACE  * SYSTEM SPACE *              *
                 **********************************************
```

If you release your space the available memory will be split into two halves, limiting the maximum area of memory which can be allocated in future:

```
                 **********************************************
    STEP 4       *              * SYSTEM SPACE *              *
                 **********************************************
```

There's no easy way around this problem, other than to minimise it by making sure that space is released in the opposite order to that in which it is allocated.

ALLOCATION and SuperBASIC extensions

SuperBASIC extension code may be loaded into memory found with ALLOCATION or RESPR, but you must ALWAYS load extensions from interpreted SuperBASIC (task 0,0), NOT into compiled task space. Generally it is best to do this in the SuperBASIC BOOT program which is executed when you turn your QL on.

You should never DEALLOCATE memory used to store extensions - there's no way to tell the interpreter, or compiled tasks, that certain commands no longer exist.

MOVE_MEMORY

This command moves a specified number of bytes (COUNT), from a source address to a specified destination. For convenience, the command is written this way:

```
MOVE_MEMORY source TO destination,count
```

There are no restrictions on the values, except that the destination address must be at least 131072, or a READ ONLY error will occur. The QL has no RAM below that address, and the check helps to avoid crashes caused by typing errors.

In a few cases, compiler optimisations may convert the MOVE_MEMORY instruction to move four bytes at a time; this uses the 68008's fastest form of MOVE instruction, but it can cause problems if you try to fill areas of memory this way: a four-byte pattern will be duplicated, rather than a single byte.

The speed is most impressive when the code is compiled; optimisations, particularly on compilers after Supercharge, make MOVE MEMORY especially fast.

This routine uses MOVE_MEMORY to store and restore the display: (Note: This works on the original QL, but may not work with newer systems or emulators.)

```
100 screen=131072:REMark start of video memory
110 all=32768:REMark size of entire display
130 buffer=ALLOCATION(all)
140 MOVE_MEMORY screen TO buffer,all
150 REMark mess up the display here
160 PAUSE:REMark wait for a key
170 MOVE_MEMORY buffer TO screen,all
180 DEALLOCATE buffer
```

Print or draw something at line 150; RUN the program then press a key.

This program 'zooms in' on the top half of the display:

```
100 FOR I=16256 TO 0 STEP -128
110   MOVE_MEMORY 131072+I TO 131200+I+I,128
120   MOVE_MEMORY 131072+I TO 131072+I+I,128
130 END FOR I
```

PEEK$

A function to read the contents of memory into a string. PEEK$(ADDR,LENGTH) returns a string LENGTH characters long, containing the same bytes as the memory addresses from ADDR to ADDR+LENGTH-1. There must be at least LENGTH bytes free when PEEK$ is called, to allow room for the 'temporary' result.

POKE$

A command to store the contents of a string in memory. Useful for display formatting, pop-up windows and 'trick' effects. POKE$ ADDR,STRING$ stores the characters of STRING$ in memory from address ADDR, onwards.

This routine uses the variable LINE$, in the task data area (rather than ALLOCATION memory) to flip the contents of the entire display upside down, a line at a time: (Note: This works on the original QL, but may not work with newer systems or emulators.)

```
100 DIM LINE$(128):REMark Screen flipper 1
120 FOR I=0 TO 32640 STEP 128
130   LINE$=PEEK$(131072+I,128)
140   MOVE_MEMORY 128,163712-I TO 131072+I
150   POKE$ 163712-I,LINE$
160 END FOR I
```

The function:

```
PEEK$(131072+START%*128,LINES%*128)
```

can be used to fetch LINES% lines of the display area, from line START% downwards, while other information is displayed there. 'Line' refers to pixel lines, numbered from 0, at the top of the screen, to 255, at the bottom; each line consists of 128 bytes of information, so you can't quite fit the whole screen - 256 lines - into a single string. The maximum string length allowed by PEEK$ and POKE$ is 32764 characters.

POKE$ can be used to restore the display later, giving true 'transient windows' - a temporary window need not corrupt the display upon which it is overlaid. You'll get strange (but not catastrophic) results if the display mode changes between when you PEEK$ as when you POKE$.

It is generally more efficient to process several short strings rather than one long one; this reduces the amount of free memory needed to hold the string before it is stored.

The new commands are not just useful for manipulating the display. They can be used in any application when you need to move, fetch or store a lot of data very quickly.

SEARCH_MEMORY

This function looks through any area of memory for a specified string, which may represent machine-code opcodes, a text literal or binary data. The function is very fast indeed, although the QL's internal memory does slow it down somewhat on unexpanded QLs. Self-modifying code (impossible in a ROM toolkit) means that the function can automatically optimise itself for any length of search-string!

SEARCH MEMORY expects three parameters: an address, a number of bytes to search from that address onwards, and a string to search for. It returns the address of the start of the string, or zero if an exact match cannot be found. This line searches the QL's 49152 byte ROM, which starts at address 0, for the name of our vanquished (anti?)hero:

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

The result will vary depending upon the version of your QL, but the name should always be found. Conversely:

```
    PRINT SEARCH_MEMORY(0,49152,"Alan Sugar")
```

will print zero.

This should come as no surprise.

POKE_F

Procedure, works as you would expect, pokes the 6-byte internal form of a floating point number into memory at the specified address. The address must be even. Example:

```
    POKE_F NewAddress,NewValue
```

PEEK_F

This function returns the value of the 6-bytes starting at the specified address and interpreted as a floating-point number. The address must be even. Example:

```
    floatnum = PEEK_F (Address)
```

SYS_VARS

This function returns the address of System Variables. On the original QL this address was always 168340. With some addon's, emulators, and SMSQ/E, the System Variables could be elsewhere. SYS_VARS is the way to find out where it is.

For example:

```
    system_var_table = SYS_VARS
```

11. Access to SuperBASIC Data Structures

BASIC_B%, BASIC_W%, BASIC_L, BASIC_POINTER, BASIC_NAME$, BASIC_TYPE%, BASIC_INDEX%, BASIC_F, BASIC_ADR

It is not normally possible to access data-structures maintained by the SuperBASIC interpreter easily or reliably, as the interpreter may move them as programs and tasks run. Seven new functions allow complete access to interpreter tables and system variables.

BASIC_B%

Returns the value of the byte at the specified integer offset within the SuperBASIC system variable area. Thus PRINT BASIC_B%(145) will display the number of the statement at which BASIC will re-start if CONTINUE is typed. Usually the value is only useful after a STOP.

BASIC_W%

Returns the word at the specified even integer offset within the SuperBASIC system variable area. Thus PRINT BASIC_W%(104) displays the number of the line being executed by the interpreter. With this trick a program can renumber itself and still know its own line-numbers!

BASIC_L

Returns the long word at the specified even integer offset within the SuperBASIC system variable area. Thus PROG = BASIC_L(16) will set PROG to the offset of the tokenised program from the start of the SuperBASIC task area. PRINT BASIC_W%(PROG) then gives the length of the first program line.

BASIC_POINTER

Returns an absolute address, computed from the 32 bit pointer at the specified even integer offset within the SuperBASIC system variable area. Thus PRINT BASIC_POINTER(16) displays the start address of the tokenised SuperBASIC program. N.B: this address may change after the value has been read, as SuperBASIC areas tend to move around if a task is invoked or terminated, RESPR is used, or SuperBASIC code is entered, run or modified using the interpreter. The function is most useful when invoked by the only task running on the QL.

BASIC_NAME$

Returns the name of the variable, procedure or function at the specified integer position in the interpreter's internal Name Table. If the position you select is outside the bounds of the table, the function will give a 'bad parameter' error. BASIC_NAME$ is not useful unless you are reading positions from some table maintained by the interpreter, so this is sensible behaviour.

BASIC_INDEX%

This function is the complement of BASIC_NAME$. It expects one string parameter, and searches for that string in the interpreter's internal list of names. Only exact matches are found - the search is case-sensitive, so 'PrINT' would not match 'PRINT', for instance.

If the name cannot be found the function returns the value -12, which machine-code programmers will recognise as the QL's internal code for 'Bad Name'. If the name is found in the Name List, the Name Table -which contains further details of each name - is searched to find out which entry is associated with that position in the list.

Under normal circumstances a match will always be found, because the QL does not create an entry in the Name List until it has made one in the Name Table. Sometimes, though, the interpreter tables can become corrupt, either because of a POKE, hardware fault, machine code failure, or (most often) a bug in the interpreter. In this case BASIC_INDEX% will detect the problem and return the value -7 - the QL internal code for 'Not Found'. Save your program at once, but don't overwrite your previous backup copy - the program in memory may well have been corrupted. Then reset the machine and re-load the program.

BASIC_TYPE%

Returns the type of the variable, procedure or function at the specified integer position in the interpreter's internal table of names. The result will be 0, 1, 2 or 4, indicating no type, string, floating-point and integer types respectively.

BASIC_F

Returns the value of the 6-bytes starting at the specified offset from the current base of the SuperBASIC area in RAM, interpreted as a floating-point number.

BASIC_ADR

This function returns the address of the machine code function or procedure at the specified integer position in the interpreter's internal Name Table. If this is out of range, or if the name is not a machine code function or procedure, BASIC_ADR returns zero.

For example:

```
    BASIC_ADR(BASIC_INDEX%("NEW"))
```

will return the address of the procedure NEW.

12. Automatic Typing and Command Entry

TYPE_IN, COMMAND_LINE, END_CMD

TYPE_IN

This command lets a program enter a string of characters, as if they had been typed by the user. All characters apart from Control F5, CAPSLOCK and Control C (or its equivalent) may be specified in the string parameter. The entry will be made in the current input window. If the string is a command to be executed at once, it should end with an ENTER character: CHR$(10).

For example, a task might want to stop and load another task to load 'on top' of itself. EXEC or EXECUTE from within the first task would not work, because a task cannot overwrite itself as it runs! But this works:

```
    TYPE_IN "PAUSE 100:EXECUTE TASK2" & CHR$(10):NEW
```

The PAUSE ensures that the first task is out of the way by the time the second is loaded from SuperBASIC.

Much more sophisticated actions are possible. A compiled program may edit an interpreted one, by typing in what the user would normally enter.

This one-line program makes the F1 key (ASCII code 232) produce the BASIC command PRINT:

```
    100 SET_PRIORITY 1:REPEAT POLL:IF PEEK_W(163978)=232 THEN
        POKE_W 163978,0 :TYPE_IN "PRINT"
```

The PEEK_W reads the ASCII code of the last key pressed.

NOTE: The SuperBASIC KEYROW command clears the keyboard type-ahead buffer - so you should avoid using it just after TYPE_IN, or the newly entered characters will be lost before they have a chance to appear on the screen!

TYPE_IN uses whatever console window is currently selected. You can direct it towards a SuperBASIC or compiled program window with CURSOR_ON - but sometimes you may want to select the SuperBASIC interpreter command line, so that you can TYPE_IN a command...

COMMAND_LINE

This is a procedure with no parameters. It causes the SuperBASIC interpreter command line to be selected. Until another window is selected, all TYPE IN strings will be directed to the interpreter's channel 0.

COMMAND_LINE is most useful when you want to TYPE_IN a command from a compiled task. This is pointless unless SuperBASIC is listening, so you should execute such a task with EXECUTE, not the W or A variants, and stop any SuperBASIC program that is running before you TYPE_IN to the COMMAND_LINE.

END_CMD

This command is used to terminate command files loaded with MERGE. A sequence of SuperBASIC commands, packed onto one un-numbered line, can be executed by MERGEing a file containing the commands. However MERGE does not close the file, which makes it impossible for you to swap that medium for another, thereafter. The solution is to put END_CMD at the end of the line; this closes the file so that all is hunky dory for the operating system.

MERGE command files are not as flexible as the copying of files to the keyboard with TYPE_IN, but the use of END_CMD has a few advantages in some cases. It can be invoked directly from SuperBASIC with no need for a copying task, and nothing appears on the screen while a command-file is MERGEd.

You should not LRUN a command file intended for MERGE; this will discard your current BASIC program and give a spurious error message when END_CMD tries to close a non-existent MERGE file.

13. Font Selection

SET_FONT

It is often useful to be able to re-define the form of characters displayed by the QL. Each display channel reads the shapes of characters from a table called a 'font'. Normally this table is in the QL's ROM, but it is possible to change the place from which this table is read so that a font set up by the user can be selected.

Many programs use a POKE instruction to signal the location of a new font. This is unwise, because the address which must be POKEd varies depending upon the configuration of the QL. The SET_FONT command is much more reliable which works correctly in compiled and interpreted programs, regardless of the presence of peripherals and extra memory, and should also work, Tony Tebby permitting, on future 'Super QL' machines.

The QL divides the range of displayed characters between two fonts. These may be different in every window, if you wish; when a window is first opened two ROM fonts are used.

The first ROM font contains the pattern for character-codes 32 (space) to 127 (copyright), whereas the second contains the pattern for 127 again, (the chequerboard) to 191 (a downward-pointing arrow). If a character-code is stored in the first font, it is used. Otherwise, the second font is consulted. If the second font doesn't cater for the code either, the first character in the second font is used.

This bizarre rule explains why there are two different patterns for code 127. The first is used when code 127 is printed; the second, chequerboard pattern, in the second font, is only used when a character code for which there is no explicit representation is printed.

Of course, you don't have to stick with the standard split-point between fonts if you design your own. For instance, you can use one font to store patterns for all codes 0-255, if you set it up as font 1 and start it with the values 0 (first code) and 255 (one less than number of characters) followed by nine bytes for each of 256 characters. If you define 256 characters in font 1, the second font will never be used.

Luckily you will not often need to know the format of a font, because *UTILITY_TASK contains a routine that lets you design or edit fonts interactively. However this manual would be incomplete without an explanation; read on if you're curious!

A font consists of a sequence of bytes, stored in reserved memory. The first byte contains the lowest character value for which there is information in the font. The second byte contains one less than the number of characters in the font. Then comes the representation of the characters.

Nine bytes are used for each character - each byte corresponds to a horizontal row of pixels in the displayed character. The binary patterns of the nine bytes determine the appearance of that character on the screen. The two least significant bits in each byte are ignored. The most significant bit is generally ignored, but may sometimes be displayed as a set pixel or cause distortion of the rest of the row. In general you should set this bit to zero.

This is a binary representation of the way in which a simple font would be stored. The decimal value of each byte is in a separate column, and the example ends with some BASIC that would set up the font, giving the QL a new CHR$(128):

```
 128    1 0 0 0 0 0 0 0     Character code
   0    0 0 0 0 0 0 0 0     Character count-1
  16    0 0 0 1 0 0 0 0
   8    0 0 0 0 1 0 0 0
  16    0 0 0 1 0 0 0 0
  32    0 0 1 0 0 0 0 0
  16    0 0 0 1 0 0 0 0     9 rows of points
   8    0 0 0 0 1 0 0 0
  16    0 0 0 1 0 0 0 0
  32    0 0 1 0 0 0 0 0
  16    0 0 0 1 0 0 0 0
```

This code sets up a one-character font containing an old fashioned (but much-loved) electronic resistor symbol.

```
    100 BASE=ALLOCATION(11)
    110 IF BASE<0 THEN PRINT"Out of memory":STOP
    120 RESTORE 140:FOR L=BASE TO BASE+10:READ V:POKE L,V
    130 SET_FONT 0,BASE
    140 DATA 128,0,16,8,16,32,16,8,16,32,16
```

You should not deallocate the memory used in a font until you close the windows using it, or stop the task; otherwise the font could be overwritten. The QL re-reads the font every time it prints a character. This accounts for the slow but flexibile performance of the display device driver.

Characters are ten rows deep when printed; an extra blank row is added to every character. Similarly, a blank column is added so that characters are six columns wide.

Larger CSIZEs are generated by adding extra blank columns, or by 'doubling up' pixels in the vertical or horizontal direction. The proportions of the characters are unchanged, but they are, in effect, made up from bigger blocks.

SET_FONT

This command is usually followed by a hash and a channel number, though the channel may be omitted - the default is channel 1. The channel should be a CONsole or SCReen window, or you'll get a NOT OPEN or BAD PARAMETER error. The channel number (if any) is followed by one or two addresses, separated by commas. These are the addresses of the first and second fonts to be used in the window. An address of 0 indicates that the corresponding standard font is to be used. The second font need not be specified if only the first one is to be altered.

14. Data Indirection Directives

REFERENCE, GLOBAL, EXTERNAL, TURBO_P, TURBO_F

Three commands indicate that certain variables can be accessed in special ways by a compiled program.

All the commands are ignored by SuperBASIC.

REFERENCE

This command should appear immediately before the definition of a procedure or function. It is followed by a list of variable names. Those variable names must all appear as formal parameters in the next procedure or function definition.

The new command indicates that the variable values are to be passed by reference, rather than by value. In other words, if the parameter values are changed inside the routine the corresponding actual parameter (named in the call) will also change. This is only allowed if the actual parameter is always a name, rather than a calculated value. The compiler checks this for you.

Array parameters must always be passed by reference. Dummy subscripts in the REFERENCE statement indicate the number of dimensions of the parameter, e.g:

```
    REFERENCE count%,address,array1(0),array2(0,0),text$
```

The values of the dummy subscript do not matter - the compiler only needs to know the 'shape' of the array, so that it can generate appropriate code to fetch values - it just counts the commas inside the brackets in a REFERENCE statement.

A string name with no subscript, such as TEXT$ in the example, is assumed to refer to a one-dimensional character array (a 'simple string').

Sharing variables and code between tasks

Two new commands indicate that certain names in a program may be accessed by external modules, or certain names in external modules may be accessed by this module. In each case the command starts with an explicit number between 1 and 100, followed by a list of procedure, function or variable names. Array names must be followed by dummy subscripts to show the number of dimensions of the array. Procedure or function names must be prefixed by the word TURBO_P or TURBO_F, and a comma, and followed by a dummy parameter list; array parameters must have dummy subscripts to indicate their dimensionality.

Tasks that use shared variables must be loaded simultaneously with the LINK_LOAD command, discussed earlier. LINK_LOAD sends messages to each task so that it knows where to find variables in other tasks.

GLOBAL

This command indicates that certain variables, procedures or functions may be accessed by other, linked tasks. There is no limit on the number of GLOBAL commands in a compiled program, e.g:

```
    GLOBAL 1,count%,TURBO_F,process(number,array2(0,0),text$),address
```

This indicates that this is module 1 in a group of tasks which share data. No two modules may have the same number.

The compiler makes sure that only one task can change the value of any global variable at a time - terrible confusion could result if two tasks tried, for example, to re-write a string concurrently. For this reason writing to GLOBAL variables is a bit slow; if you're after top speed you should try to avoid updating a global variable where an internal one would do.

EXTERNAL

This command indicates that variables, procedures or functions are called from another task. The command starts with the explicit number of the module where the variables are declared. For instance, for module 2 to access the variables that we declared as GLOBAL in the last example, this command would appear in the program for module 2:

```
    EXTERNAL 1,max%,TURBO_F,crunch(number,table(0,0),string$),pointer
```

Notice that the names MAY match, but they don't have to; but all the types and numbers of parameters and dimensions MUST correspond, or data will not be shared correctly. You must list all the variables that another module declares as GLOBAL if you want to access any of them -the compiler needs all the names in order to work out what corresponds to what. The program will not be slowed down unless you change the value of global or external variables.

A name cannot be both GLOBAL and EXTERNAL, for obvious reasons!

Tasks can change the value of EXTERNAL variables. However all manipulation of the value takes place in the dataspace of the task containing the code - not the task where the variable is GLOBAL. All channels are 'privately owned' by each task, but channel identifiers may be passed back and forth between tasks.

Sometimes it may be annoying to have to specify all of the GLOBAL variables in another module just because you want EXTERNAL access to a few of them. In this case you can divide the GLOBALs into several groups by declaring each group on a line of its own and following the module number with a literal string - the name of that group. Other modules can then refer to one or more groups by name in corresponding EXTERNAL statements, for instance:

```
    GLOBAL 1,"Databank",table(0,0),Xbound%,Ybound% GLOBAL 1,
      "User interface",command$,option$,TURBO_P,complain(err%)
```

In this case module 2 could access the Databank by including this directive:

```
    EXTERNAL 1,"Databank",BANK(0,0),W%,H%
```

and module 3, could declare an interest in the user-interface variables thus:

```
    EXTERNAL 1,"User interface",cmd$,option$,TURBO_P,howl(problem%)
```

15. Data Type

IMPLICIT%, INPLICIT$

The FOR and SELECT commands in SuperBASIC have the weakness that they will only work with floating-point values. On early versions of the QL you could type in integer or string FOR and SELECT statements, but they wouldn't work under the interpreter. Later QL's don't even let you enter such commands - you get a 'BAD LINE' error. Some compilers can handle integer FOR and string or integer SELECT (string FOR seemed more trouble than it would be worth!), but there's no way to type these in on a late QL.

Two commands let you show that a variable should be treated by the compiler as if it were an integer or a string, even though its name does not end with '$' or '%'. The commands are not compatible with all compilers - if in doubt, consult your compiler manual.

The commands only work with the names of 'simple' variables - not arrays. The interpreter won't let you use an array element as a loop counter or SELECT variable; this would be clumsy even in compiled code.

IMPLICIT%

This command should be followed by a list of scalar variable-names -not procedures, functions, or arrays. The names will thereafter be treated by the compiler as if they ended with a per cent sign (integers). Thus users of late QL models can specify integer FOR and SELECT statements in their compiled programs. These are often much more efficient than their floating-point counterparts.

For example, you might want to change this loop to use a fast integer FOR. This would speed up counting and checking and remove the need to repeatedly convert the count into an integer subscript:

```
100 FOR COUNT=0 TO 100:T(COUNT)=J(COUNT)+K(COUNT)
```

To make that an integer FOR loop, just add this line, anywhere in your program:

```
IMPLICIT% COUNT
```

NOTE: This will mean that EVERY reference to COUNT is treated as if it were COUNT%, including any local variables with that name. This will not cause you any trouble unless you're in the (bad) habit of using the same variable name for several different purposes. If you do this you may have to rename the variable you want to be treated as an implicit integer, to remove the ambiguity.

IMPLICIT$

This command should be followed by a list of scalar variable-names, as for IMPLICIT%. In this case the names will be treated by the compiler as if they ended with a dollar sign ($). Thus users of late QL models can use powerful string SELECT statements in compiled programs. For example:

```
100 IMPLICIT$ NAME
110 DIM NAME(128):REMark a 128 character string
120 INPUT "Please type your name: ";NAME
130 SELect ON NAME
140   ="":PRINT "You are either shy or lazy!"
150   ="?":PRINT "Hmm. Very enigmatic."
160   ="Simon":PRINT "An excellent name, that."
170   ="Freddy":PRINT "Not you again!"
180   =-1.61E616 TO 1.61E616:PRINT "You are not a number!"
190   =REMAINDER:PRINT "Hello ";NAME
200 END SELect
```

There's no limit on the number of IMPLICIT statements in a program, but you should not declare the same name to be a string and an integer; if you are so stupid, the compiler will use whatever type you declare last. This problem is unlikely to crop up if you keep all your IMPLICIT commands in one place; say, near the start of the program.

16. Finding Memory Requirements

FREE_MEMORY, PC_FREEMEM, DATASPACE

FREE_MEMORY

The function FREE_MEMORY returns the amount of space available to SuperBASIC - the unused contiguous memory in the system - if you call it from an interpreted program, or the amount of unused space within a task's data area, if you call it from a compiled program.

PC_FREEMEM

Sometimes you may want to know the amount of 'free space' in the machine, even from within a compiled program. In this case you can use PC_FREEMEM, a function that always returns the same value whether it is called from compiled or interpreted code.

DATASPACE

This function finds the amount of data space associated with an executable task. The result is the number of bytes of dataspace or a negative number if the file is not a task. For example:

```
    PRINT DATASPACE("FLP1_BOOT")
    -2                  [Invalid Job]

    PRINT DATASPACE("SILLY")
    -7                  [Not found]
```

17. Turbo Compiler Directives

TURBO_objfil, TURBO_taskn, TURBO_repfil, TURBO_diags, TURBO_sound, TURBO_struct, TURBO_model, TURBO_list, TURBO_locstr, TURBO_optim, TURBO_windo, TURBO_objdat, TURBO_buffersz, TURBO_objstk, TURBO_ref, TURBO_V, TURBO_DUMMY%, TURBO_DUMMYF, TURBO_DUMMY$, TURBO_DUMMYP, MANIFEST, DATA_AREA

All these are ignored by SuperBASIC except the four TURBO_DUMMY's which return not implemented. All are either directives for Parser_task or are to be used inside a compiled program.

First are the Parser_task directives.

TURBO_objfil <devn_FileName>

This sets the output filename for the compiled program.

TURBO_taskn <String>

This sets the job name of the compiled program as seen by JOBS and similar commands, to a maximum length of 12 characters; anything longer is truncated rather than rejected.

TURBO_repfil <devn_FileName>

Sets the report filename; a null string selects the display for the report.

TURBO_diags <Number> or <String>

This sets the compiler to exclude, display or include the line-number diagnostic information in the compiled program. 0 excludes it for a smaller file, 1 displays only and 2 includes it; also uses strings like "I" or "INCLUDE" to include or "D" or DISPLAY" to display but not include, or "O" or "OMIT" to omit the information.

TURBO_sound <Number> or <String>

Sets on or off sound made during comilation. "NO" or 0 sets it off and "YES" or 1 sets it on.

TURBO_struct <Number> or <String>

Instructs Turbo whether to treat the program as structured, i.e. structured is denoted by 1, unstructured by 0. Use "S" or "STRUCTURED" or "F" or "FREEFORM" instead of 0 or 1 for clarity. See Turbo manual for definitions of structured and freeform programs.

TURBO_model <Number> or <String>

Selects 16-bit or 32-bit code production mode, use "<" for less than 64kb of code (16-bit) or ">" for more than 64kb programs, (which is 32 bit). Does not affect the amount of dataspace allowed, just the maximum size of executeable file produced. Also accepts numbers, 0 selects 16-bit code, 1 selects 32-bit code.

TURBO_list <Number> or <String>

Sets listing on or off during compilation. "NO" or 0 sets it off and "YES" or 1 sets it on.

TURBO_locstr <Number> or <String>

Uses "I" or "IGNORE", "R" or "REPORT", "C" or "CREATE" to set the option for treatment of local strings; see TURBO manual for more details of the meaning of this. Also accepts 0, 1 or 2 for Ignore, Report or Create respectively.

TURBO_optim <Number> or <String>

Sets optimisation type to be used, e.g. "B" or "BRIEF" for smallest code, "R" or "REMS" for switchable control by <REMark +> and <REMark -> statements and "F" or "FAST" for fastest (but very bulky) code. Will use 0 to 2 instead but strings are clearer.

TURBO_windo <Number>

Sets the number of automatic window definitions to copy from the SuperBASIC interpreter windows. Must be in the range 0-32.

TURBO_objdat <Number> or DATA_AREA <Number>

Sets the number of kilobytes of dataspace to give the compiled program,

TURBO_buffersz <Number>

Sets the number of kilobytes of buffer space to be used by Turbo to compile the program. (This was added by David Gilham at the request of George Gwilt.)

TURBO_objstk <Number>

Sets the size of stack in the compiled program to an even number of bytes between 350 and 9998 inclusive unless the number given is less than 350 in which case the size configured in Codegen_task is used instead. (This was added by David Gilham at the request of George Gwilt.)

TURBO_ref

Tells Parser_task that string parameters can be sent by reference to machine code extensions. In this case the compiled program will only run if the first long word of system variables is "SMSQ".

TURBO_..... Settings

Name	Number/String	Meaning
TURBO_locstr	N/S	0 = Ignore$
		1 = Report$
		2 = Create$
TURBO_windo	N	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_buffersz	N	number = buffer size/1024
TURBO_objstk	N	number = stack size (bytes)
TURBO_taskn	S	Task Name
TURBO_objfil	S	Name of Compiled Program
TURBO_repfil	S	Name of Report File
TURBO_ref	-	Allows reference string parameters (SMSQ)

MANIFEST

This sets named constants for the source program to use. It will take a list or a single assignment but not expressions, and assignments should be separated by ":" characters. Example:

```
    MANIFEST : PiSquared=9.869604 : Fragment=.6
```

All through the source you may use PiSquared and Fragment instead of the numeric values and they will be compiled as if the source contained the numbers. This avoids the use of variable space and speeds up some programs while allowing clarity by using names for constant values.

The following keywords are for use inside a compiled program as is explained fully in TurboS4_txt of the TURBO Manual.

TURBO_V

This function returns the absolute address in the Vector Table of the named variable.

TURBO_DUMMY%, TURBO_DUMMYF, TURBO_DUMMY$ and TURBO_DUMMYP

These three functions and one procedure are just dummies. If they appear in a compiled program space will be reserved for them in the Vector Table. Use of this can be made by TURBO_V.

18. Final Comments

This version of Turbo Toolkit is fully ROMable (we think Turbo Toolkit has been since v3.00). You can use this version of Turbo Toolkit freely and give it away, and if running a public domain or similar library it may be sold on disk for the usual small copying fee. You are not allowed to sell this toolkit commercially. You may give it away with programs compiled with Turbo (commercial or otherwise) so users can run your programs. You should always include this text file and the other files connected with Turbo Toolkit in this set.

Turbo Toolkit is copyright the Turbo Team, that is (in alphabetical order) Charles T Dillon, Simon N Goodwin and Gerry Jackson. Additions and updates by David Gilham and Mark Knight are copyright their respective authors.

The DEMOS_BAS file is the same Turbo Toolkit demos file supplied by Digital Precision, but updated to use SYS_VARS rather than PEEKing and POKEing about in the system variables without asking the system if they have been moved.

None of this software including Turbo Toolkit version 3h27, TurboPatch, CONFIGURE, the DEMOS_BAS file or the ReSize_BAS file, nor any of the _BIN files is or can be guaranteed to be either bug free or suitable for any particular purpose. You must bear the entire responsibility for determining if it is useful to you, and if any lurking bugs may harm your system. It's free software and comes with no warranty of any kind, so there.

The above said, we think it's great stuff!

Mark Knight,

28 April 2000.

Table of Contents