TURBO
TOOLKIT

THE ULTIMATE QL TOOLKIT

by

Simon Goodwin

Published by Digital Precision, 222 The Avenue, London E4 9SE

Channel manipulation commands

Commands: CHANNEL ID, SET CHANNEL, CONNECT

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.

'Random access' file handling

Two new commands allow you to read or write any part of a file withput having to read past the rest of the file. Whole files can be accessed 'at random' - treated like enormous character arrays.

POSITION

This function returns the current position within a specified channel, e.g. PRINT POSITION(3). If present (the default is 1) 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 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.

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 SUPERCHARGE 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 127, as explained earlier.

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

The format of the SET_PRIORITY command is shown below:

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

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

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 new 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.

Cursor control

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.

Error detection and trapping

Commands: DEVICE_STATUS, DEVICE_SPACE, WHEN_ERROR, END_WHEN

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).

At Digital Precision we chose to attack this problem by providing a new 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!

Since SUPERCHARGE we have enhanced DEVICE_STATUS. The function now attempts to read the file or device specified, before trying to write to it. If the read results in an error (such as IN USE) DEVICE_STATUS returns the appropriate code at once. This change means that several new classes of error are detected quickly and unambiguously. The function also tells you the amount of free space on a disk or tape, if a new file is to be created.

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

Our long-term plan is to provide asynchronous error-trapping similar to that partially implemented on late QLs, but version-independent, secure and (hopefully) more sophisticated. For this reason two error-trapping directives have been included in the TURBO TOOLKIT; these are ignored by the interpreter, but they will allow full error trapping - on ANY version of the QL - if an appropriate compiler is used. At the time of writing parts of such compilers are in development and under test.

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.

WHEN_ERROR

END_WHEN

These statements allow asynchronous error-trapping routines to be declared in appropriately compiled programs. The statements are ignored in interpreted and SUPERCHARGEd programs.

DEVICE STATUS

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

DEVICE_STATUS This function now attempts to read the file or device specified before trying to write to it. If the read results in an error (such as IN USE) DEVICE_STATUS returns the appropriate code at once. This change means that several new classes of error are detected quickly and unambiguously; a host of functions similar to the original DEVICE_STATUS have been published, but the new DEVICE_STATUS, partly designed by Chas Dillon, is peerless in its ease of use and the amount of information it returns.

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 corrupting the contents. If the drive is write-protected or other tasks are reading, the function will return an appropriate code.

If the file does not exist, DEVICE_STATUS tries to create it. If this succeeds, the function kills the resultant file and returns with the number of free bytes on the medium, after allowing space for the 'empty' file. Serial devices, which have no real 'capacity', pretend that they are the same size as the largest possible QDOS file-structured device: almost 33.6 Megabytes!

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

```
PRINT DEVICE:STATUS("flp1_turbo_task")
-8
```

DEVICE_STATUS is still short of ideal in one respect; it is incapable of detecting a write-protected microdrive cartridge. This is because the QL's microdrive device driver cannot recognise write-protected cartridges - it starts to perform the requested operation and pretends that all has gone well but gives a 'Bad or changed medium' asynchronously a minute or so later.

It is theoretically possible to fix this, as the second processor can detect whether or not the currently turning microdrive is protected, but this can't be done by a user-program as the device driver can asynchronously switch from one microdrive to the other. The only reliable fix is to re-write the entire microdrive handler... Start petitioning Tony Tebby now!

DEVICE_STATUS returns a floating point value corresponding to a long integer, and indicating the degree of success it had in opening a channel. The numbers are tabulated below:

VALUE RETURNED	MEANING
0 or more	The device exists, and is not busy; a file with the name specified (if any) does not yet exist. The name or other parameters (if any) are valid. 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 is being read.

Table of values returned by DEVICE_STATUS.

Task and Compiler invokation

Commands: CHARGE, EXECUTE, EXECUTE_A, EXECUTE_W, LINK_LOAD

ssociated keywords: COMPILED, OPTION_CMD$, SNOOZE, DEFAULT_DEVICE

EFAULT_DEVICE

All file names used with CHARGE, LINK_LOAD or the various kinds of EXECUTE can se the 'default device'. If you don't specify a device at the start of a ame, the commands will automatically put the 'default device' name there. his name consists of five characters. When TURBO TOOLKIT is supplied it is 'MDV1 ', but you can change this in two ways: either temporarily, by typing a ew five-character string as a parameter of DEFAULT_DEVICE:

```
DEFAULT_DEVICE "flp1 "
```

- or permanently, by using the Default Editor in UTILITY_TASK.

HARGE

A new command to invoke Digital Precision SuperBASIC compilers. CHARGE must be entered from SuperBASIC. If a 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.

First CHARGE executes PARSER_TASK, loading it from the default device. You can abort (prematurely stop) it at any time, by pressing ALT and SPACE, or whatever other 'abort' keys you have chosen. When the parser has finished, if there were no errors, OPTIMISE_TASK (if selected) and CODEGEN_TASK are executed.

CHARGE comes configured to work with Supercharge, but the Default Editor (documented in the Utility manual) can alter it to work with later compilers. You must ensure that CHARGE is correctly configured for the compiler you wish to use, or strange and annoying things are likely to happen!

EXECUTE

A much-improved implementation of the EXEC command. Several compiled tasks 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 QUILL
```

inally, 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 pened for the task to use. You will get a 'bad parameter' error if the file lready exists. For instance:

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

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

lternatively 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
```

f 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!

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 loaded 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 tasks containing appropriate code. Programs compiled with Version 1 of Supercharge should be re-compiled with a later compiler if you wish to take advantage of the new features. Newer compilers will be available at special rates to existing owners of SUPERCHARGE.

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.

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.

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 with a Digital Precision compiler 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.

Editing data on the screen

Three new 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.

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, although you will need to use a Digital Precision compiler to PRINT the value accurately.

Binary input and output functions

Commands: FLOAT$, INTEGER$, STRING$, GET%, GET$, GETF, INPUT$

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 accomodate 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 preceeded 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$;
```

inary input

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

ET%

A new 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 efore the channel number is optional.

GETF

A new 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 new 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))
```

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 num<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 num<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 number*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
```

Memory management

ommands: ALLOCATION, DEALLOCATE, MOVE_MEMORY, PEEK$, POKE$, SEARCH MEMORY

uperBASIC has few facilities to handle areas of memory. Five new extensions ure this weakness, letting you manipulate memory 'en bloc' and within strings.

EEK$ and POKE$ can be used to generate pop-up (transient) windows, graphic ffects, 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.

uperBASIC 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 uperBASIC are running. ALLOCATION and DEALLOCATE do not have these problems.

LLOCATION

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 he command works perfectly when tasks are running. Indeed, ALLOCATION may be erformed 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 y LIST TASKS. These values should follow the number of bytes, e.g:

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

lemory is only released if the specified owner task terminates, OR the EALLOCATE 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 or microdrive 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.

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

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

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:

There's no easy way around this problem, other than 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.

OVE_MEMORY

his command moves a specified number of bytes (COUNT), from a source address to a specified destination. For convenience, the command can be written in two ays:

```
MOVE_MEMORY source TO destination,count

MOVE_MEMORY count,source TO destination
```

he first form is assumed if the first separator is TO.

here are no restrictions on the values, except that the destination address ust 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.

his is a quick way to fill COUNT bytes from address BASE with the value INIT; COUNT is assumed to be at least 1:

```
POKE base,init:MOVE_MEMORY count-1,base TO base+1
```

n 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 nstruction, but it can cause problems if you try to fill areas of memory this ay: a four-byte pattern will be duplicated, rather than a single byte.

You can always force a slower byte-by-byte move, by making one or other of the OURCE and DESTINATION addresses odd (as in the above example). Alternatively ou can take advantage of the optimisation and POKE the first 4 bytes to the desired value; for example, to clear COUNT long words from BASE:

```
POKE_L base,0:MOVE_MEMORY count-1,base TO base+4
```

Normally MOVE_MEMORY copies bytes from low addresses before it copies those higher in the area to be moved. This can be inconvenient if you want to move a ata-structure to a higher address without overwriting part of itself in the rocess. For this reason, MOVE_MEMORY will work 'backwards' - moving the highest addressed data first - if a negative byte count is supplied.

his example shows the use of MOVE_MEMORY with ALLOCATION and DEALLOCATE:

```
100 LIN=ALLOCATION(128):REMark Screen flipper 1
110 IF LIN<0 THEN PRINT"Out of Memory":STOP
120 FOR I=0 TO 32640 STEP 128
130   MOVE_MEMORY 128,131072+I TO LIN
140   MOVE_MEMORY 128,163712-I TO 131072+I
150   MOVE_MEMORY 128,LIN TO 163712-I
160 END FOR I
170 DEALLOCATE LIN
```

The speed is most impressive when the code is compiled: optimisations, articularly on compilers after Supercharge, make MOVE_MEMORY especially fast.

This routine uses MOVE MEMORY to store and restore the display:

```
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 new 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 to flip the contents of the entire display upside down, a line at a time:

```
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.

)KE$ 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 ode 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 efore 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 ery quickly.

EARCH_MEMORY

This function looks through any area of memory for a specified string, which ay represent machine-code opcodes, a text literal or binary data. The unction 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.

Access to SuperBASIC data-structures

Commands: BASIC_B%, BASIC_W%, BASIC_L, BASIC_POINTER, BASIC_NAME$,
BASIC_TYPE%, BASIC_INDEX%

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.

Automatic typing and command entry

Commands: 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, such as the command file TK BAS supplied with TURBO TOOLKIT. 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.

Selecting a new character font

Command: SET_FONT

t 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 hange the place from which this table is read so that a font set up by the ser can be selected.

Many programs use a POKE instruction to signal the location of a new font. his is unwise, because the address which must be POKEd varies depending upon he 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 ermitting, 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    9 rows of points
 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
```

```
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
```

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

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.

Two new fonts, called ALIEN_FONT and BOLD_FONT respectively, are supplied with the TURBO TOOLKIT. The former is a collection of characters and simple graphics for games; the latter is an attractive new font designed by Sam Goodwin for use in MODE 8. The standard font is also supplied; you can guess the file name!

Data-indirection directives

ommands: REFERENCE, GLOBAL, EXTERNAL

Associated keywords: PROCEDURE, FUNCTION

hree new commands indicate that certain variables can be accessed in special ways by a compiled program. Not all compilers support these powerful xtensions; consult your compiler manual to check this. The commands are ompletely ignored by Supercharge and the SuperBASIC interpreter.

EFERENCE

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

The new command indicates that the variable values are to be passed by eference, rather than by value. In other words, if the parameter values are hanged inside the routine the corresponding actual parameter (named in the all) 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.

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

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

he 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 etch values - it just counts the commas inside the brackets in a REFERENCE tatement.

A string name with no subscript, such as TEXT$ in the example, is assumed to efer 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 odule. 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 PROCEDURE or FUNCTION, 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%,FUNCTION,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%,FUNCTION,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$,option$,PROCEDURE,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$,option$,PROCEDURE,howl(problem%)
```

Data type and space directives

Commands: IMPLICIT%, INPLICIT$, DATASPACE

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 new 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.

DATASPACE

This is a directive which indicates the amount of dataspace to be allocated to a compiled SuperBASIC program. It is ignored by Supercharge and SuperBASIC, but works with later compilers from Digital Precision. It should be followed by an explicit number - the amount of data space required, in kilobytes.

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.

Finding the amount of unused memory

Command: 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.

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

```
PRINT PEEK_L(163856)-PEEK_L(163852)
```

That number corresponds to the number of bytes available for disk and microdrive buffering - the QL uses all otherwise-unused space for that purpose.

CONTENTS

1.0 GETTING STARTED

The Turbo Toolkit Utilities let you edit sound effects and character fonts. You can also change the 'default' values assumed by Toolkit commands.

This is how to load the toolkit and utilities:

(1) Connect your QL to a display and power-supply, then press the Reset button. Wait for the F1/F2 prompt.

(2) Insert the Turbo Toolkit cartridge into the left-hand (Microdrive 1) slot at the front of your computer.

(3) Press either F1 or F2 to select Monitor or TV display. The microdrive will run and load the Toolkit commands.

(4) Type this command to invoke the utilities:

    EXECUTE_A MDV1_UTILITY_TASK

    When they have been loaded a menu will appear.

(5) Type F1 to load the Character Graphics editor, QLUDGE.

    Type F2 to load the Sound effect editor.

    Type F3 to load the Toolkit default editor.

    Type F4 to stop the utility task and return to BASIC.

You can EXECUTE the utility package without destroying the SuperBASIC program you are working on. This can be useful if you want to try out a sound or graphical idea while programming. But the Toolkit default editor works by changing the toolkit extension file on disk or tape; you won't be able to detect the effect of changes until you re-load the toolkit.

The toolkit MUST be loaded before you EXECUTE the utility package, or the package will complain that some commands are 'Not Loaded'. To cure this problem, type:

    MERGE MDV1_TK_BAS

2.0 QL SOUND EFFECT EDITOR INSTRUCTIONS

This is a powerful sound-effect design utility. It lets you generate and modify virtually any sound effect which the QL can produce, using a 'front panel' which graphically and digitally displays the parameters of the current sound.

2.1 Joystick and keyboard control

The sound editor may be controlled either from the keyboard or from a joystick connected to the CTL1 socket. You can swap controls at any time. If you chose the joystick you must still use the keyboard to select a few operations.

By moving the joystick up and down you produce the effect of pressing the vertically pointing arrow keys, situated to the right of the space-bar. Move the joystick left and right to produce the effect of the horizontally pointing arrow keys, at the other end of the space-bar. Pressing the joystick FIRE button has the same effect as pressing the space-bar.

Wherever this manual refers to the arrow keys or the space-bar, it may be assumed that joystick controls may be used for the corresponding purpose if a joystick is connected.

Only one key (or joystick press) will be recognised at any time.

The following keys are recognised by the utility. Other keys have no effect:

Q, J, R, SPACE, ENTER, F1, and the four directional arrows.

The combination of keys CTRL, ALT, 7, will stop any program and sound-effects, due to a quirk in the electronic design of the QL. These keys halt the QL by causing an attempt to access non-existent hardware. Luckily, it is very difficult to press this combination of keys by accident. If you do press these keys, the only way to re-start the program is to reset the computer and re-load from step (3) of the 'getting started' sequence (on page 2 of this manual).

2.2 Overview of the display

The display may be divided into three areas.

(1) The right-hand side of the screen contains a column of instructions and indicators.

(2) The bottom of the screen shows the SuperBASIC command needed to generate the current effect, or 'help' for the user.

(3) The rest of the screen (under the main heading) consists of a graphic 'front panel', in the form of eight 'slider controls', like those on a mixing desk. The position of each slider (or horizontal line across the control) corresponds to an aspect of the sound being edited. Abbreviated names for each control appear beneath them.

2.2.1 Auto-echo mode

The 'Auto echo' mode is shown in the top right-hand corner of the display. If this area contains the word ON, the sound being edited will be automatically produced whenever its parameters are altered. In other words, the computer produces the new sound-effect whenever a change is made.

There is one exception to this rule. If the duration of the sound is such that you are able to make a further change before the last rendition is completed, the computer will not 'echo' the new setting until the last effect has finished. This means that, if you change a setting repeatedly and rapidly, the computer doesn't keep interrupting itself to produce the latest sound - it produces new sounds as often as it can, letting each effect run from beginning to end.

If 'continuous sound' is selected (duration is set to zero) at the same time as auto echo is turned on, new sound-effects WILL interrupt one another as settings are altered - otherwise you would never hear the effect of a change, since the computer would wait for ever for the previous effect to come to an end.

Press the ENTER key to turn auto echo mode ON or OFF. If the mode is ON, the key turns it OFF, and vice-versa. Turning auto echo OFF cancels any sound being produced.

2.2.2 Hearing the new sound

If you only wish to hear the sound you are generating intermittently, turn auto echo mode OFF and press the space-bar whenever you wish to hear the sound-effect. If you press the space-bar again while the effect is being generated, the sound will stop at once.

2.2.3 The Control Indicator

Immediately below the auto echo indicator, on the right-hand side of the display, is the control indicator. This contains the name of the slider control which is currently being altered. At any time you may press the F1 (help) key for more information about the current slider control.

2.2.4 The Instruction summary

Below the Control Indicator, for the majority of the right hand side of the display, is the Instruction summary. This is simply a list of the keys recognised by the program, with mnemonic text. Thus, the left and right arrow keys are used to select a given slider control, the R key is used to reset a slider to the position corresponding to zero, and so on. These keys, and their effects, are listed and explained later, under the heading 'sound control keys' (section 4.0 of this manual).

2.2.5 The SuperBASIC command display

This utility is not an end in itself - it is a means of selecting sound-effects for use in other programs. The bottom line of the display shows the SuperBASIC command which should be included in a program, where you want to produce the sound being edited. Some sounds (e.g. the sound of a telephone ringing) are best used in combination with PAUSE statements (for silent delays) and loops, so that the sound is repeated or punctuated by silence. One value in the SuperBASIC command appears on a light background. This is the value which is currently being adjusted.

Press the upward or downward arrow keys to increase or decrease the value. If you hold the key down the values will change ten times more quickly than if you press the key intermittently. This enables you to advance quickly to a desired value, and then 'trim' the value to a specific setting by pressing the upward or downward arrow key a few times.

The 'duration' value can only be adjusted in steps of 100 units. If all the intermediate values were displayed, the program would run very slowly. A resolution of 100 units allows you to adjust the duration of a sound in measures of less than a hundredth of a second, which should allow enough accuracy for even the most discerning ear. A duration of zero corresponds to a continuous sound.

The 'speed' value also moves in steps of more than one unit, since a change of one unit makes virtually no change to the sound generated. In this case, the setting can be adjusted to a tolerance of 10 units.

The left and right arrow keys are used to select different values for alteration. If a direction arrow key is used to select a value before the first or after the last, the last and first parameters are selected respectively. In other words, control 'wraps around' from the last parameter to the first, and vice versa.

2.2.6 Machine-code sounds

It is possible to use this program to design sound-effects for use within machine-code programs. In this case the values listed in the Command Display must be transfered to the 8049 processor (the IPC) by an operating system trap.

The program listing overleaf takes the form of a subroutine which calls the operating system. The routine was tested using the Metacomco Assembler.

Communication with the IPC is performed by @DOS. The BEEP parameters should be stored in the COMMAND table before the subroutine is called: Pitch 1 and Pich 2 are 8-bit values; Duration and Speed are 16-bit values; the others are 4-bit values.

Once the COMMAND table has been set as required, the subroutine should be called with the instruction JSR BEEP.

```
* SUBROUTINE TO CALL QDOS 'BEEP'
*
* PARAMETERS SHOULD BE LOADED INTO THE 'COMMAND'
* AREA BEFORE INVOCATION.
*
* BY SIMON GOODWIN, WITH THANKS TO LEON HELLER.
*
* EQUATES
*
MT.IPCOM  EQU      17              IPC COMMAND INDICATOR
*
BEEP      LEA.L    COMMAND,A3      POINT TO PARAMETER TABLE
          MOVEQ    MT.IPCOM,D0     INDICATE PURPOSE OF TRAP
          TRAP     #1              CALL QDOS
          RTS
*
COMMAND   DC.B     10              TELL IPC TO GENERATE SOUND
          DC.B     8               NUMBER OF PARAMETER BYTES
          DC.L     $AAAA           INDICATE PARAMETER SIZES
PITCH1    DC.B     0               1 + VALUE USED IN BASIC
PITCH2    DC.B     0               1 + VALUE USED IN BASIC
SPEED     DC.W     0
DURATION  DC.W     0
STEP.WRAP DC.B     0               TWO PARAMETERS IN ONE BYTE
RAND.FUZZ DC.B     0               TWO PARAMETERS IN ONE BYTE
REPLY     DC.B     1               INDICATE IPC NEED NOT REPLY
*
          END
```

You can find out whether or not the IPC is already generating a sound by reading the word at address 163990. The value therein will be zero if no sound is being generated, and non-zero otherwise.

It is not possible to fully discuss the interface between QDOS and assembly language in this manual. The QL operating system is fully documented in a number of books, including the QL ADVANCED USER GUIDE by Adrian Dickens. This book is published by Adder; ISBN 0 947929 00 2.

2.2.7 Front panel slider controls

The slider controls are the eight vertical bars which occupy the majority of the screen. Each bar corresponds to one of the parameters in the SuperBASIC BEEP command.

At any time one of the sliders will appear in a lighter colour than the others. This is the 'current slider control'. Press the left or right arrow keys to select the control to the left or right of the present one. The slider on the left is considered to be on the right of the right-hand control. Likewise, the slider on the right is considered to be on the left of the leftmost one. This 'short cut' provides a quick way of moving from one end of the row of controls to the other.

Use the upward and downward pointing arrow keys to adjust the position of the slider (horizontal bar) within the control. The effect of adjusting each control is explained in the section of this manual entitled 'slider control effects' (section 4.0).

Sliders may be moved up and down the control at two rates. Every time the upward or downward arrow key is pressed, the slider moves by a certain number of units across the scale. If the key is held down for more than about half a second the slider will begin to move at a steady, generally faster, rate across the scale. Release the key to stop the movement. This 'two speed' approach makes it possible to adjust the position of the slider quickly, in large steps, holding the key down; and move it to the precise position required, in smaller steps, by tapping the keys.

NOTE: The movement of sliders, or the rate of selection of controls, can be retarded if a single, very high-pitched note is being sounded. The solution is to turn off the sound and then adjust the controls before turning the sound back on. A sound can be muted by pressing ENTER, if auto echo mode was ON, or the space-bar, if auto echo was not selected.

The problem occurs because of the slow rate of data-transfer between the main QL processor and the component responsible for generating sounds. The sound generator is also responsible for reading the QL keyboard, so the problem is most noticeable if a key is held down while a note of pitch 0 is played. Steady, high-pitched notes 'clog the system', reducing the rate at which the main processor can work.

3.0 SLIDER CONTROL EFFECTS

Each slider control alters one of the eight parameters of the SuperBASIC BEEP command. The effects of each parameter are briefly documented in the 'concepts' section of the Sinclair QL Manual. That document also states that "The SuperBASIC BEEP command is best used experimentally rather than syntactically." This is the purpose of this utility.

3.1 Sound-effect development

The best way to design a sound-effect is by trial and error, but that can be a lengthy process. This utility sets out to speed the development of new sounds through the 'juggle' feature. If the letter key "J" is pressed at any time, the computer will take over control of the slider controls, randomly altering the value of each parameter of the BEEP command. As the controls are altered the resultant sound is generated, and you are finally returned to control the slider you were adjusting before 'juggle' was selected.

The juggle command can be used repeatedly until the basis of the required effect can be heard. Then the sliders can be manually adjusted to give the desired duration of effect, pace, number of repetitions, and so on.

Since few people can guess the right values of pitch, wrap, noise and so forth for a given sound, the juggle facility can be a very effective way of designing sounds. It is an especially useful option when a sound with a given 'mood' is required, such as a scolding sound or a happy sound.

The juggle command alters the setting of each slider at random. This can lead to very brief sounds (which are sometimes useful) but these can conceal more interesting, longer-lasting noises if the duration value is less than the cyclic period of the effect.

It may be useful to try each effect with a duration of zero (continuous) before discarding it in favour of another randomly-generated one, since it takes some time for the display to be updated after a juggle. This can conveniently be done by moving to the 'duration' slider, and then pressing "J", "R" (to reset duration to zero) and the space-bar (to hear the continuous sound).

3.2 Slider control summary

The following list explains the function of each control, and goes some way toward explaining the effects produced by a given change.

3.2.1 Duration control

The Duration control is the leftmost slider displayed. According to Sinclair, each unit represents 72 microseconds of sound duration. The maximum value allowed by the program is 32700 (about two seconds), and the minimum is 100, less than one hundredth of a second. In practice, the unit time seems rather less than 72 microseconds.

A Duration of zero corresponds to a sound which continues for ever. You may stop such a sound in two ways: if you press the space-bar the sound is stopped temporarily; it will recur if the space-bar is pressed again, or any arrow key is used while auto echo is ON. Alternatively, you may stop the sound by pressing the ENTER key - in this case you will hear nothing until you press the space-bar or ENTER again. The 'reset' option - key "R" - can conveniently be used to set Duration to zero without the need to press an arrow key repeatedly.

3.2.2 Pitch 1 control

Pitch 1 is the main note generated by the utility. Pitches range in value from 0 to 255, with 0 being the highest pure tone possible, and 255 the deepest. It is possible to generate deeper notes by adjusting the other controls so as to produce a 'beat' between two higher pitches.

The further up the display the slider appears, the deeper the note produced.

The values of pitch are not linearly related to the musical scale - the musical scale is a logarithmic one, while successive values of pitch on the QL decrease in frequency geometrically. However, it is possible to relate the two scales, so as to produce simple music - the table overleaf will be useful to those who wish to produce sound-effects in a (fairly!) musical sequence:

Musical Note	Pitch 1 value
C	24
B	26
A£ or Bb	28
A	30
G£ or Ab	32
G	35
F£ or Gb	38
F	41
E	44
D£ or Eb	47
D	50
C£ or Db	54
C	57
B	61
A£ or Bb	65
A	70
G£ or Ab	75
G	80
F£ or Gb	85
F	90
E	96
D£ or Eb	102
D	108
C£ or Db	116
C	124

Table of musical notes related to pitch values (approx).

Bear in mind that the relationship tabulated is only approximate, even for pure tones, and other non-zero parameters - or even a specific note duration other than zero - may affect the note generated, for a given value of Pitch 1. The machine-code used to generate sounds on the QL is concise and quite flexible, but it is not easy to use it to generate music.

The easiest way to produce accurate notes of a specific length is to use a BEEP duration of zero and stop each note with a BEEP command with no parameters. PAUSE can be used to time the delay between the start of one note and the start of the next. A brief silence between notes often helps to improve the sound.

3.2.3 Pitch 2 control

This control is used to set the secondary pitch produced by the utility. If Pitch 1 and Pitch 2 are set, the computer will play each note in turn, at a rate determined by the setting of Speed (see 3.2.4). If Step (see 3.2.5) is set to a non-zero value, the computer will play intervening notes on its way from Pitch 1 to Pitch 2.

It is possible to produce simulated chords, 'zap' sounds and many other effects by adjusting the relationship between Pitch 1 and Pitch 2. If step is zero and speed is a small value, notes deeper than a pure pitch 255 can be produced by 'beating' Pitch 1 and Pitch 2 together.

As with Pitch 1, values of Pitch 2 may vary between 0 and 255, with 255 being the deepest note, selected when the slider is at the top of its travel.

3.2.4 Speed control

The speed control sets the rate at which the computer moves between Pitch 1 and Pitch 2. Small values of Speed give 'zap' sounds and smooth changes of pitch - larger values produce a succession of distinct notes.

Speed can be set to any value between 0 and 4090. Small values correspond to a high speed. The control moves in steps of ten units, or 100 units if the key is held down.

3.2.5 Step control

The Step control sets the size of interval between successive notes. If STEP is zero, the computer alternates between Pitch 1 and Pitch 2, without playing any intermediate notes.

The Step control can be set to any value between -8 and +7. A negative Step produces a note which descends initially; a positive Step gives an ascending sequence. Consequently, if you play an effect with a Step of one, and then with a Step of minus one, the second effect will sound like the first one, but played in reverse. Some of the other controls (Wrap, Fuzz and Noise) may distort or reverse the effects of a given Step.

3.2.6 Wrap control

The Wrap control governs the behaviour of an effect once it reaches one of the specified pitches.

Values of Wrap can vary between zero and 15. If Wrap is set to zero, this causes the sound to switch to the other pitch. If wrap is 15, the sound-effect 'bounces' between Pitch 1 and Pitch 2 forever. Intermediate values cause the effect to bounce a limited number of times.

The effect of Wrap sounds like a periodic reversal of Step. High values give a longer delay between reversals.

3.2.7 Fuzz control

The Fuzz control is used to add periodic random deviation to the pitch generated.

Values range between 0 and 15; small values make little audible difference, while values of 8 and over alter the pitch randomly more and more, giving a 'sweeping' sound, or a 'wind' effect if Step is set to zero.

3.2.8 Noise control

This is another control used to make sound-effects more random. The Noise control periodically alters the value of Speed (see 4.2.4), producing a sound which moves unevenly from Pitch 1 to Pitch 2 and back again.

Values may range between 0 and 15. High values of Noise give large variations in Speed; values between 0 and 7 give little audible change (the Noise routine shares a random number generator with the Fuzz routine) but greater values can give quite gross distortion. A Noise value of 15 gives a characteristic 'buzz and chirp' sound, as individual notes are randomly emphasised.

4.0 SOUND CONTROL SUMMARY

Left and Right arrows:	Select a given slider for alteration - the current control is shown in a lighter colour than the others, and the matching BEEP value is on a light background.
Up and Down arrows:	These adjust the setting of the current slider. Press the key briefly to change the position by a small amount; prod it for about half a second if you wish to make larger changes. The slider will move steadily until you release the key or it approaches the end of its travel.
SPACE key:	Stops any sound being generated, or (if no effect was being produced) starts the generation of the currently-selected sound.
Letter key "J":	'Juggles' the slider positions, adjusting each control at random, while playing the corresponding sound-effect. The computer stores commands typed while 'juggling' takes place, and executes them as soon as it has finished.
Letter key "R":	'Resets' the position of this slider to zero (a high Pitch, infinite Duration, fast Speed or no Step, Noise, Fuzz or Wrap).
ENTER key:	Toggles auto echo mode ON or OFF. If echo is ON, sound is produced whenever an arrow key is pressed.
Function key 1 (F1):	Calls up a display of 'help'; the function of the current slider is stated at the foot of the screen.
Letter key 'Q':	Stops the Sound effect editor and returns you to the main menu.

5.0 CHARACTER FONT EDITOR INSTRUCTIONS

Q.L.U.D.G.E stands for 'Quantum Leap User Defined Graphics Editor'. Unlike the 'kludge' which was fitted to early QL hardware, this is a powerful character design program. It gives the user the ability to re-design the characters displayed by the QL, allowing alternative type-faces, symbols, or animated graphics to be quickly and easily constructed.

Either the keyboard or a joystick may be used to 'paint' the required graphic design onto a grid. Complete character sets, known as 'fonts', may be saved and re-loaded in place of the standard set. In this way graphics may be used in your own programs, in SuperBASIC or other languages.

QLUDGE is a component of the Turbo Toolkit file UTILITY TASK. To load it follow the instructions in Section 1 of this manual, and press F1 when the Turbo Toolkit utility menu appears.

5.1 Joystick and keyboard control

QLUDGE may be controlled either from the keyboard or from a joystick connected to the CTL1 socket. If the joystick is chosen it may still be necessary to use the keyboard to select some operations.

5.2 Initial file selection

When QLUDGE is first loaded, the message:

Please enter font device and file name:

will appear, with the default entry (MDV1_STANDARD_FONT) underneath. You can edit that default to match any other font file. If you want to edit a character set which has already been created, edit the default to match the device and file name of your font.

Thus, if you wished to edit the file BOLD FONT, which contains a new character set which looks much more attractive in MODE 8 than the standard QL set, you would edit the word 'STANDARD' to 'BOLD', leaving the text:

```
MDV1_BOLD_FONT
```

The microdrive will run for a while, as the font is loaded, and then the main menu will appear.

If you do not wish to load an existing file, press the ENTER key in response to the message. A copy of the standard QL font will be loaded and then the main menu will appear.

If you enter an invalid device or file-name the QL will detect this and make a sound before re-displaying the name so that you can correct it.

As with other QL packages, you can obtain a directory listing of the contents of microdrive 2 by typing a question mark in place of a file-name. Use CONTROL and F5 together to stop the display scrolling off the screen; press any other key to re-start scrolling. Once the entire directory has been displayed you must press the ENTER key to return to the initial question.

5.3 The main menu

Once you have specified the font which you wish to edit, a menu of five options will appear. Use the function keys to select any option. It is possible to return to the menu from any option.

5.3.1 F1: Help

Press the F1 key if you are unsure of the way QLUDGE is used. An abridged version of this manual will appear, in the form of three screens of information. Press ENTER to step from one screen to the next. Once all three screens have been displayed you will be returned to the main menu.

5.3.2 F2: Modify Character

Press F2 if you wish to modify the definition of a character. You will be asked:

Which Character?

Type the character which you wish to modify (press any key which produces a displayable character, shifted or unshifted) and press ENTER. Alternatively, you may specify a character by entering its ASCII numeric code, as listed in the 'Concepts' section of the QL User Guide.

If the character which you specify falls outside the range of the current QL font (as standard, codes 31 to 127, or 'error' to 'copyright') you will be asked to 'Try Again'. You must specify a valid character before you can return to the menu.

Once a valid character has been entered, an expanded display of the current definition of that character appears in the middle of the screen. A smaller representation appears alongside. Use the arrow keys or joystick to move the flashing question-marks over the definition. Press Space or Fire to change each the colour of each dot. If the dot is coloured it will be cleared - if it is cleared it will be coloured. The small display mimics the appearance of the larger one.

Beware of using the leftmost column of the character grid if you intend to use CSIZEs 2 or 3 (wide characters). A restriction in the software which drives the QL's display may cause such characters to be displayed incorrectly by early QL models.

Once you are happy with your design, press the ENTER key to store it in memory in place of the original which you began by editing.

Press ESCAPE if, for any reason, you wish to return to the main menu without storing the new design. This is usually because you have made a mess, and want to try again without having overwriten the previous design.

5.3.3 F3: Save current font

Press F3 from the main menu if you wish to save the complete font which you have designed. You will be asked to specify the device and filename under which the font is to be saved. A standard-sized font occupies two blocks on a microdrive cartridge, or three on disk.

Press the ENTER key alone if you want to give up this option and return to the menu.

As with other QL packages, you can obtain a directory listing of the contents of microdrive 2 by typing a question-mark in place of a file name. Use CONTROL and F5 to stop the display scrolling off the screen, and press any other key to re-start scrolling. Once the entire directory has been displayed you should press the ENTER key to return to the question asking you to specify a file name.

If the file you specify does not exist the QL will beep and the main menu display will be re-drawn. Correct the cause of the error, by using a different device or file name, or changing the destination cartridge, and try again.

5.3.4 F4: Duplicate character

Option F4 allows you to copy the definition of any character into the place of any other. This is very useful if you are designing an animated sequence and need to define a group of characters with small differences between them. Such a trick was used to define bugs which wave their legs and open and close their mouths in the demonstration character set ALIEN FONT.

Type the character which you wish to duplicate (press any key which produces a displayable character, shifted or unshifted) and press ENTER. Alternatively, you may specify a character by entering its ASCII numeric code (see the 'Concepts' section of the QL User Guide).

If the character which you specify falls outside the range of the QL font (codes 31 to 127, or 'error' to 'copyright') you will be asked to 'Try Again'.

Once you have chosen the character to be copied you will be asked to specify that which is is to be overwritten with the copy. The same rules apply.

If you decide, at any point after pressing F4, that you do not want to copy or overwrite any character, just specify the same character twice in succession. The character definition is be copied in place of itself - in other words, there is be no change to the stored font, and you are returned to the main menu.

5.3.5 F5: Display entire font

Press F5 when the main menu is displayed if you wish to see the present state of the entire font. Each chararacter is shown in two sizes, so that you can see the effect of dots in the leftmost column if character width is set (using CSIZE) to 2 or 3. Press the ENTER key to return to the main menu.

Character 255, in the second font, is defined as a 'space' character when displaying the entire font. You can redefine the space character yourself without messing up the display, but you should be wary of defining character 255.

6.0 QLUDGE IN PRACTICE

Characters designed with QLUDGE can be used in BASIC or machine-code programs. This section of the manual explains how fonts are associated with windows, and how animated effects can be achieved.

6.1 Using fonts outside QLUDGE

Just three instructions are needed to add an alternative font to a SuperBASIC program. You need to reserve memory for the representation of the font, load it into memory, and associate the font with a particular window. Thus, from turning on your QL, you could select the BOLD FONT for window 2 (that used for listings) with these commands:

```
BASE=ALLOCATION(2+9*97)
LBYTES MDV1_BOLD_FONT,BASE
SET FONT #2,BASE
```

The commands could be typed one by one, or contained within a program.

The first command reserves space for the font. Two bytes are needed for the 'header', which tells the QL which characters are defined therein, and nine bytes are used for the definition of each of the 97 characters in a standard QLUDGE font.

The header and definition bytes are generated in the required format by QLUDGE - all you need to do is load them into memory. This task is performed by the second command. The file BOLD_FONT is loaded from microdrive 1 into the area of memory indicated by BASE.

Finally you need to tell the QL to use the new font, rather than the standard one. The third command does this, using a Turbo Toolkit extension command. In machine-code the manager trap SD.FOUNT (TRAP 3 with D0=37) does the trick.

If you want to leave the standard windows (0, 1, 2) as normal, you should define a new window for use with graphics. This can be done by opening an extra channel, using a command such as:

```
OPEN #3,scr
```

You would re-define the font used by that channel with the command:

```
SET FONT #3,BASE
```

Notice the '3' in the place of the '2' in the earlier example.

You may use one copy of a font for more than one window. You just POKE the details of each window, as above, so that they indicate the same font. When you turn on the QL it uses one copy of the normal font for all three standard windows. You may also POKE different font details as a program runs, if you wish to use more than 97 user-defined characters in a given window.

6.2 Smooth graphic movement

You can produce smoothly-moving graphics on the QL by alternately setting the print position with the CURSOR command and PRINTing characters. CURSOR gives much finer control than the AT command.

There are two ways of moving characters around the screen. The first is to print a character in one place, and then overwrite it with a space (or other background character), then print it somewhere else, and so on. The computer prints the characters so quickly that they appear to move from one place to the next. If you use this technique, it is very important to re-print the character immediately after you have overwritten the previous image, or the display will flicker.

An alternative technique uses the command OVER -1. The command means that every dot printed takes account of the previous dot in that position. Thus, if you print a black dot over a white dot, the black dot appears. Print another black dot, and the original white returns.

With this scheme you can print a character on any background and it will be clearly delineated. To restore the background to its original appearance, you need only print the same character in exactly the same place. Thus graphics can move over a complex background without erasing it as they move.

The best way to reduce flicker is to compile your program into machine-code, so that the processing delay between the output of one character and the next is minimised.

If you need to move large shapes, or multicoloured patterns, around the screen user-defined characters are not really appropriate. You need a specialised package, such as our Super Sprite Generator.

7.0 TURBO TOOLKIT CONFIGURATION

The Turbo Toolkit UTILITY_TASK lets you change the default values assumed by Toolkit commands. This is done by altering the Toolkit code file. In case of accidents, you should always work with a copy of the code file, and never the original 'master' copy. You may need the original if you make a mistake!

The Toolkit default editor is selected by pressing F3 when the UTILITY_TASK has been loaded. The loading procedure is explained in section 1 of this manual.

7.1 How to configure the toolkit

You are first asked for the name of the toolkit file you wish to 'customise'. The default name is displayed; you can either enter this or edit it to correspond to a new name you may have chosen. If you scrub out the whole name and type XX the program assumes that you don't want to use the default editor after all, and returns you to the main menu.

When you enter the name the computer checks that the file exists and can be altered. If not, it beeps and invites you to correct the name. As before, enter XX to abort.

Once you have entered a valid name the toolkit file is opened; from this point onwards you must answer all the questions, or you could corrupt the file.

The next question asks for the name of the 'default device' - the device assumed to contain tasks and compiler overlays. You must enter five characters; normally these will be three characters of device name (e.g. FLP or FDK), a single digit drive-number (1 to 8) and an underscore.

You are then asked whether or not you want a beep to sound when errors are trapped by EDIT$, EDIT% and EDITF. Press Y for sound or N for silence.

Tasks invoked with CHARGE or EXECUTE_A can be stopped by pressing certain keys. These are usually ALT and SPACE, but you can use other keys if you wish. The next two questions determine what keys the computer looks for.

The first such question asks you to type the character, or character code, which will stop tasks. Type one character, or a numeric code. For instance £ and 96 both have the same meaning, as 96 is the code for the £ sign. A full list of codes is near the start of the Concepts section of the QL User Guide.

The next question asks whether or not you want the computer to check for the ALT key as well as the character you chose previously. Answer by pressing Y or N. It is a good idea to use ALT key combinations to abort tasks, as it is hard to enter them by accident.

Next you can set the default length of the buffer used by EDIT%, EDITF and EDIT$. The standard default is 40 characters, which is ideal for file name strings but may be rather excessive for numbers. If you make a lot of use of EDIT% you may wish to set this default to 5 or 6 characters, so that entries - by default - are confined to a narrow window. You can always specify an explicit buffer length in your programs if the default is not appropriate.

The penultimate question concerns the 'pipes' used to link tasks invoked with EXECUTE or its brethren. The Toolkit commands make these linking pipes 200 bytes long, which means that 199 characters can be written to the pipe before it becomes 'full'. Once a pipe is full no more characters are accepted until some characters have been read from the other end of the pipe.

Toolkit linking pipes may be up to 32767 bytes long. Communication is typically more efficient with long pipes, but correspondingly more memory is used.

The final question determines the behaviour of the CHARGE command. This has been designed to invoke the SUPERCHARGE compiler or future,- yet more sophisticated, products from Digital Precision. If you intend to use CHARGE to invoke SUPERCHARGE, answer the question with Y. Otherwise type N.

The computer writes information based on your replies to the toolkit file, and waits for you to press a key. You must reset the QL and re-load the toolkit commands before you can take advantage of use the new default values.

8.0 IMPORTANT NOTICE

TURBO TOOLKIT DEMONSTRATIONS

TURBO TOOLKIT comes with over 100 interesting and useful SuperBASIC procedures and functions, in the file TURBO_TK_DEMOS.

These routines can be used freely in your own programs, as long as you explicitly credit the authors and publisher on-screen and in the documentation of commercial products. The routines illustrate the power of the QL, TURBO TOOLKIT, and Digital Precision's SuperBASIC compilers.

Look through the TURBO_TK_DEMOS file. Try things. Enjoy yourself!

THE LIBRARY MANAGER

You can extract any group of routines from the demonstration file, or from any other SuperBASIC file that you wish to dismantle, with the TURBO TOOLKIT 'library manager'. In fact this is a very useful, friendly program, especially if you use like to use LINK_LOAD with the TURBO compiler.

Dismantled files can be recombined at any time (e.g. for testing) with MERGE. This process is neither faster nor slower than LOAD, so it makes sense to keep files in sections and load them individually, or all together with a MERGEd 'master' program that MERGEs the others!

You can keep useful testing and debugging routines in one file, so that you could use them with any file. It makes sense to standardise the range of line numbers used in your own programs, but it is easy enough to LOAD RENUMber and SAVE a program file. You can, of course, do this with the original TURBO_TK_DEMOS file.

The only 'snag' is that you must use procedures and functions in your programs. Minimal SuperBASIC or 'Pseudo-Fortran' is impossible to chop up safely and logically. SuperBASIC's block structure is powerful, logical and expressive. If you haven't started using it yet, perhaps it's about time you did.

The LIBRARY MANAGER is the 'missing link' in a source module management system built around SuperBASIC - it provides the complement of LOAD, MERGE and SAVE, yet it can also handle GROUPS of routines automatically, as we shall see.

Ideally the library manager should be compiled, but it will run happily, albeit slowly, under the SuperBASIC interpreter. To run the manager, type:

LRUN MDV1_LIBRARY_MANAGER (perhaps with a different device name)

You are asked to choose an 'input file' and an 'output file'. The former is where the routines will be taken from; the latter is the new file which you want to contains selected routines.

For the purposes of this experiment we'll assume you've got your backup TOOLKIT cartridge in MDV1 and an empty formatted cartridge for results in MDV2. Other device names (e.g. 'FLP1') may be substituted at will.

The input file is thus MDV1_TURBO_TK_DEMOS, and the output any name

you choose - say, MDV2_UCHOOSE. The file names are validated and you must press Y to delete an existing output file.

After this you are asked to enter a series of routine names. You can change the maximum number of names by altering line 150 of the program. In this case, we'll choose some debugging routines; FIND, HOW_COME, BASIC_SIZE and BASIC_SEARCH. Type in their names on separate lines; if you make a mistake, re-type the name on a separate line. The order of names does not matter; you will be told later if a routine cannot be found.

Type a blank line at the end of the list of names. You will then be asked if you want to save this list for use later. If you press Y you must enter the name of the new file to hold these names; unreasonable entries are detected and you are asked to try again.

All the names in such a file can be included amongst those you wish to find by specifying the name file, preceeded by an asterisk, instead of a routine name. The corresponding names are listed as the file is read.

The program then searches for the specified routines, and puts them into the chosen output file. Scanning stops as soon as all of the routines have been found. Only the first is copied, if duplicates exist. Any routines that were not found are named when the end of the input file is reached.

N.B. The LIBRARY MANAGER can be 'confused' if the END of a definition is on the same line as the start, or the END is duplicated or missing. In general you will end up with extra data in your file; less often, this stops the manager with an error message. You'll never notice this unless you try to extract a routine that is garbled in such a way.

INDEX V 1.01

This index (the thing you're reading!) details the TURBO_TK_DEMO routines including names of any other routines called, for the LIBRARY_MANAGER, result returned, if a function, and any appropriate extra comments, e.g:

CALLS is followed by a list of other DEMO routines this uses.
TOOLKIT indicates that a routine uses TURBO TOOLKIT commands.
MULTI-TASK indicates that a routine should be compiled & multi-tasked.
SNG indicates that a routine was written by Simon N Goodwin.
DCN indicates that a routine was written by David 'Newt' Newell.

We hope to expand the range of demonstrations provided with the toolkit. As usual, upgrades are available at any time, and we welcome your own suggestions. Remember, this is a LIST of EXAMPLES - please study the actual SuperBASIC source code if you want to know more about the workings of the QL, TURBO TOOLKIT, or individual routines. The first routine, "FIND", will help you locate each routine in the file.

(1) FIND - Finds loaded SuperBASIC procedures and functions.

Example: FIND "HOW_COME" prints first line of routine HOW_COME
TOOLKIT, SNG

(2) HOW_COME - SuperBASIC debugging utility.

Lists PROC/FN calls after an error; type HOW_COME to find out how an error was reached. ROM bugs may require that you type HOW_COME twice.

TOOLKIT, SNG

(3) SLAVE_SPACE - Returns space available for file slave blocks.

TOOLKIT, SNG

(4) BUFFER_SIZE - Returns size of the INPUT & COPY buffer.

TOOLKIT, SNG

(5) PROG_SIZE - Returns size of the SuperBASIC program.

TOOLKIT, SNG

(6) VARIABLE_SIZE - Returns size of SuperBASIC variable area.

TOOLKIT, SNG

(7) BASIC_SIZE - Returns TOTAL size of the SuperBASIC task.

SNG

(8) CLOCK - Trivial multitasking clock utility

TOOLKIT, MULTI-TASK, SNG - Must call PROC to start

(9) SPOOL - Simple multitasking file copier

TOOLKIT, SNG, CALLS: READ_OK, WRITE_OK, FILE_SIZE, YEA_OR_NAY

(10) YEA_OR_NAY Prompts then returns 1 for Y, 0 for N

TOOLKIT, SNG

(11) READ_OK - returns 1 if file/device (name is parameter) can be read.

TOOLKIT, SNG

(12) WRITE_OK - returns 1 if file/device (name is parameter) can have data written to it.

TOOLKIT, SNG, CALL: YEA_OR_NAY

(13) FLIP1 - rapidly reflects the screen display top to bottom.

If you find a use for this, please let us know!

TOOLKIT, SNG

(14) FLIP2 - reflects the screen display even faster.

TOOLKIT, SNG

(15) STRETCH - stretches the display to twice its normal height.

TOOLKIT, SNG

(16) ANIMATE - Simple MOVE_MEMORY pop-up window/animation.

TOOLKIT, SNG

(17) CREATE_FONT - Defines new CHR$(128) - a resistor, on channel £1.

The RESTORE will need to be renumbered if the DATA is moved.

TOOLKIT, SNG

(18) FILE_SIZE - Returns size of a file given its name (assumed OK).

Uses SET_POSITION with big numbers - maximum size for MDV is 1.6e6, (to avoid a ROM fault) but you can use 2e9 with disks.

TOOLKIT, SNG

(19) FAST_COPY - speedy file copier.

Very quick indeed but only copies files that fit in available memory. Prompts for input & output names.

TOOLKIT, SNG, CALLS: FILE_SIZE, TASK_TYPE, YEA_OR_NAY

(20) BASIC_SEARCH - scans through a program listing and finds ANY text.

Works by LISTing to a pipe. Prompts for text to be found. Alter pipe length if lines may be more than 500 characters long.

TOOLKIT, SNG, CALL: LINE_RANGE

(21) FUNCT_REMARK - Makes fn key F1 produce the string "REMark ".

TOOLKIT, MULTI-TASK, SNG

(22) NAME:- UNSHIFT - Transposes minus and underscore character input.

If you're sick of using SHIFT to get an underscore this utility will transpose '-' and '_' for you, automatically. Auto-repeat is disabled.

TOOLKIT, MULTI-TASK, SNG

(23) TO_UPPER - Converts a stream of mIxeD-cAsE data to CAPITALS (sic).

This routine 'translates' data from one channel, transmitting the result down another channel. You need the TURBO compiler for this.

TOOLKIT, MULTI-TASK, SNG

(24) NET_NUMBER - Returns this machines' QL network station number.

SNG

(25) TASK_COUNT% - Returns the number of tasks currently loaded

SNG

(26) CHANNEL_COUNT% - Returns the total number of channels currently open.

SNG

(27) SET_KEYBOARD - Sets keyboard response characteristics

Expects two parameters X%, Y%; X% = pre-repeat period, in frame times - see manual on SUSPEND_TASK. Y% = No. of repetitions/sec (up to 50).

SNG

(28) DRIVE_RUNNING% - Returns the No. of the microdrive now turning.

Result is 0 if no microdrive is turning; some QL file operations fail unless DRIVE_RUNNING%=0 when they start - the ROM's wrong.

SNG

(29) CAPS_LOCK - Force 'capitals lock' on or off.

Expects single parameter n - n<>0 forces lock on; n=0 forces off.

SNG

(30) TASK_BASE - Returns the address in memory where a task is stored.

The parameter, task_num%, corresponds to the first No. in LIST_TASKS. If task_num% = -1 then THIS TASK is assumed.

SNG

(31) BASIC_RUNNING - Returns 1 if a SuperBASIC program is running.

Useful when you want to use TYPE_IN; BASIC_RUNNING must be 0 then.

TOOLKIT, SNG

(32) CURRENT_LINE% - Returns the current line No.

Works whether compiled or interpreted. For obvious reasons you may want to renumber this. Needs a compiler that supports IF COMPILED.

TOOLKIT, SNG

(33) COMMAND_FILE - Types an entire file into SuperBASIC or any program.

Copies file to the input queue via TYPE_IN; useful for program testing. Prompts for filename - preset the value if need be. You may have to adjust LINE$ for long input lines.

TOOLKIT, MULTI-TASK, SNG, CALL: READ_OK

(34) VIDEO_OFF - Disables display but leaves program running.

SNG

(35) VIDEO_ON - Re-enables the display after VIDEO_OFF.

SNG

(36) TRACE - Prints latest line number of currently running SuperBASIC.

Adjust window £0 in the compiled task to suit your display layout.

TOOLKIT, MULTI-TASK, SNG

(37) LINE_RANGE - Prompts for a range of line numbers and an interval.

TOOLKIT, SNG

(38) SC_PROFILE - Profile utility for SUPERCHARGE or TURBO tasks.

A very powerful utility when performing hand-optimisations. Compiled programs do not maintain the line number when in ROM, so you should RENUM differently and run twice to avoid spurious indication. PROFILE assumes the 'last' line number when in ROM.

TOOLKIT, SNG, CALL: LINE_RANGE

(39) SB_PROFILE - Profile utility for interpreted SuperBASIC programs.

Only measures interpreter speed - not potential when compiled.

TOOLKIT, MULTI-TASK, SNG, CALL: LINE_RANGE

(40) TASK_TYPE - Discriminates tasks from data-files.

Expects a single parameter, the file name; returns the dataspace of loadable tasks, or an error code - see manual entry on DEVICE_STATUS.

TOOLKIT, SNG

(41) BASIC_VECTOR - Returns address of code of a command or function.

Expects a resident procedure or function name string as a parameter.

TOOLKIT, SNG

(42) FREEZE_SCREEN - Simulates CTRL F5.

SNG

(43) SCREEN_ACTIVE - Returns status of screen - non zero means 'frozen'.

Allows fast output skipping if the screen is frozen.

SNG

(44) MEDIUM_COPY - Fast copier for a complete medium.

Uses RAM to copy complete files from one medium to another. Prompts for device names, gives option to format the receiving medium.

Only works for files which will fit in available memory.

CALLS: FILE_SIZE, READ_OK, WRITE_OK, TASK_TYPE, YEA_OR_NAY, TOOLKIT, SNG & DCN.

(45) RE_SAVE - Replaces an existing SuperBASIC SAVE file.

Expects a file name as a parameter. If the file already exists prompts for delete, othewise simply saves the current program. The file name must be quoted or a string, e.g. RE_SAVE "mdv1_name"

TOOLKIT, DCN, CALL: YEA_OR_NAY

(46) TASK_PRIORITY% - Returns priority of a given task.

Expects a single parameter - the task number.

TOOLKIT, DCN, CALL: TASK_BASE

(47) TASK_STATUS% - Returns status number of a given task.

Expects a single parameter - the task number. Results:

0 = possibly active; 1 or more = delay time (in frame ticks) till re-activation; -1 = task suspended; -2 = waiting for another task to complete

TOOLKIT, DCN, CALL: TASK_BASE

(48) LIST_PROCS - Lists SuperBASIC procedures and their start lines.

TOOLKIT, DCN

(49) LIST_FNS - Lists SuperBASIC functions and their start lines.

TOOLKIT, DCN

(50) LIST_EXTRAS - Lists all machine code procedures and functions.

TOOLKIT, SNG

(51) BASIC_ATTR - Used to retrieve information about a channel.

For example calls, see routines 53, 54 etc. This call works on QLs, but may not work on 'QL-compatible' systems - it tests some machine-specific addresses.

TOOLKIT, SNG & DCN, CALL: CON_CHANNEL

(52) CON_CHANNEL - Returns true if ch% is (probably) a CON channel!

TOOLKIT, SNG

(53) INK_COLOUR - Returns ink colour for a given channel.

TOOLKIT, DCN, CALL: BASIC_ATTR

(54) PAPER_COLOUR - Returns paper colour for a given channel.

TOOLKIT, DCN, CALL: BASIC_ATTR

(55) STRIP_COLOUR - Returns strip colour for a given channel.

TOOLKIT, DCN, CALL: BASIC_ATTR

(56) BORDER_COLOUR - Returns border colour for a given channel.

TOOLKIT, DCN, CALL: BASIC_ATTR

(57) BORDER_WIDTH - Returns border width for a given channel.

TOOLKIT, DCN, CALL: BASIC_ATTR

(58) X_ORIGIN - Returns graphics X origin for a given channel.

N.B. Remember that graphics coordinates are FLOATING POINT values.

TOOLKIT, DCN, CALL: BASIC_ATTR

(59) Y_ORIGIN - Returns graphics Y origin for a given channel.

TOOLKIT, DCN, CALL: BASIC_ATTR

(60) G_SCALE - Returns the graphics scale for a given channel.

N.B. Remember that G_SCALE may be a floating-point value.

TOOLKIT, DCN, CALL: BASIC_ATTR

(61) X_SIZE% - Returns the width of a given window in MODE 4 pixels.

In MODE 8, X_SIZE% will return even values from 2-512.

TOOLKIT, DCN, CALL: BASIC_ATTR

(62) Y_SIZE% - Returns the height of a given window in pixels.

TOOLKIT, DCN, CALL: BASIC_ATTR

(63) CURSOR_X% - finds MODE 4 horizontal pixel cursor posn. for a window.

In MODE 8, CURSOR_X% will return even values from 0-510.

TOOLKIT, DCN, CALL: BASIC_ATTR

(64) CURSOR_Y% - Returns the vertical pixel cursor position for a window.

TOOLKIT, DCN, CALL: BASIC_ATTR

(65) X_MIN% - Returns the horizontal MODE 4 pixel offset of a window.

In MODE 8, X_MIN% will return even values from 0-510.

TOOLKIT, DCN, CALL: BASIC_ATTR

(66) Y_MIN% - Returns the pixel line number where a given window starts.

TOOLKIT, DCN, CALL: BASIC_ATTR

(67) CHAR_ATTR% - Returns details of output mode for a window.

Expects a channel No. as a parameter. Returns an 'attribute' byte, (the CSIZE, FLASH, OVER, UNDER settings) which can be tested as follows:

IF (CHAR_ATTR% && 1) : Underline

IF (CHAR_ATTR% && 2) : Flash (mode 8 only)

IF (CHAR_ATTR% && 4) : use a transparent background

IF (CHAR_ATTR% && 8) : exclusive-OR forground on background

IF (CHAR_ATTR% && 16) : Double text height (20 pixels)

IF (CHAR_ATTR% && 32) : Extended width text (8 or 16 pixels)

IF (CHAR_ATTR% && 64) : Double-width text (or MODE 8 set)

IF (CHAR_ATTR% && 128) : Characters have been positioned with CSIZE.

TOOLKIT, DCN, CALL: BASIC_ATTR

(68) FONT1_ADR - Returns address of first font used by a window.

TOOLKIT, DCN, CALL: BASIC_ATTR

(69) FONT2_ADR - Returns address of first font used by a window.

TOOLKIT, DCN, CALL: BASIC_ATTR

(70) SETUP_MUSIC - Loads note arrays used by PLAY_TUNE.

This routine must be called before PLAY_TUNE or PIPED_MUSIC.

SNG

(71) PLAY_TUNE - Renders a string as a sequence of notes and sounds.

With this command the QL can play simple musical tunes and jingles. PLAY_TUNE is easy to use but limited by the QL's minimal hardware.

You must call SETUP_MUSIC before using this routine.

The parameter is a string of notes and durations to be BEEPed. The format is rather like that supported by Microsoft BASIC's PLAY command. This list indicates the meaning of characters:

Seven letters, A to G, correspond to notes on a musical scale.

An optional digit may follow a letter:

Digits, 1, 2 or 3, indicate the octave of the note; default 1. The scale on the current release starts at C, so ascending notes in the scale of C Major are written:

C, D, E, F, G, A, B, C2, D2, E2, F2, G2, A2, B2, C3

With the intervening sharps and flats, this sequence includes all the notes currently (and reasonably accurately) supported by PLAY_TUNE.

The optional characters '£' or 'b' may come at the end of the specification of a note. '£' indicates a sharp note (one semitone above the default) and 'b' indicates a 'flat'. In view of the QL's meagre sound hardware, there is no support for tempering: A£ = Bb.

The 'length' of each BEEP is set, in TV frame times (1/50 or 1/60 second), with the character L followed by the required number. The default length - more properly, duration - is nine frame-times.

The 'space' between each BEEP is set, in TV frame times, using the character S followed by the required whole number, e.g. "S12". The default space is two frame-times.

The character R causes the last 'tune' to be repeated ad nauseam.

The character Q stops processing of the string; a Q is assumed if the end of the string is reached.

The character X allows the full, 'eXtra' powers of the QL's BEEP statement to be used. The character is followed by a sequence of seven numbers. These numbers correspond to the second through to last (eighth) parameter of a 'full' arcane BEEP statement. You can work out parameters for X with the TOOLKIT UTILITY_TASK sound designer. For instance, this plays a little jingle:

```
PLAY_TUNE "L30 X4,6,2190,1,0,0,0"
```

As you can see, single commas or spaces may appear after each number or note in a tune string; these keep numbers apart and help to make tunes easy to read. You should put sharps, flats, octave numbers and initial parameters straight after letters, but apart from that you can format strings as you wish. Letters may be written in upper or lower case. Remember to put a space or comma before the letter 'B' when it represents a note, as opposed to a flat. 'A2B2' is not valid - it is read as 'A2b', then an unexoected '2' - but 'A2 B2' works fine.

Play this string to hear a familiar nursery rhyme:

```
"L8 S1 CDEC CDEC EF L16 G L8 EF L16 G L4 GAGF L8 EC L4 GAGF L8 ECC
X182,0,0,0,0,0,0 L16 C L8 C X182,0,0,0,0,0,0 L16 C R"
```

Notice the way that X is used to produce the note 'G0' - a pitch outside the two-octave range that PLAY_TUNE supports by name.

The whole tune could be played an octave higher without any need for this trick, but it is included as an indication of the way that PLAY_TUNE can handle ANY sound the QL can make; you can mix

single-letter notes and effects in any combination.

PLAY_TUNE works very slowly under the interpreter; you can make it a little faster by removing spaces from your music string, but even so you will hear little pauses as processing takes place.

The way to cure this is to compile the routine, with Supercharge or TURBO. The advantage of TURBO is that you can set up a small 'background' task which will work like a device, accepting music from a pipe and playing it while the main program gets on with something else.

PLAY_TUNE stops with one of three error messages if there is something wrong with the tune string. In each case the string is printed, up to the point where the error cropped up.

Tune value expected:S10 LA ?

- A number (duration or X parameter) was expected at the last character position shown.

Pitch out of range:C3£ ?

The note shown is outside the 25 note range of PLAY_TUNE. Use X or transpose the tune in a different key.

Unexpected character in tune:L20 J ?

The last character shown is not recognised by PLAY_TUNE.

TOOLKIT, SNG, CALLS: EXTRACT_INT%, PARSE_N_PLAY, BUM_NOTE, BUMP_POSITION; Call SETUP_MUSIC before PLAY_TUNE.

VIRTUAL ARRAYS

This substantial library of routines allows very large 'arrays' to be set up on disk or microdrive, without encroaching upon free RAM.

Any free memory that does exist is automatically used to store parts of the 'array' file which are frequently used. The QDOS system fetches the 'next' block concurrently as data is read or written, to minimise device delays. This mechanism makes virtual arrays much faster than might othwerwise be expected.

Virtual array subscripts are floating-point values, so the ONLY limit on the size of such arrays is the size of your disks, and there's no need to re-type variables if you get a new drive.

Routines are provided to CREATE virtual arrays of 1, 2 or 3 dimensions, holding integers, floating-point values or characters (one dimensional virtual character arrays - single strings on disk! - are not supported, though you can always use a one-element 2D character array if you must!

The creation routines expand array files to hold all possible contents, and clear the array. You can use existing arrays with the V_OPEN routines. Several tasks can read an existing file simultaneously, but only one channel can be open to an array which may be altered OR read.

Other routines allow any data-type to be stored at or read from virtual arrays of every type.

Virtual arrays may be expanded by writing an appropriate number of zero bytes at the end of the file and adjusting the maximum first subscript you may pass, to use the extra space.

All the space in a virtual array is allocated to data. Integer elements occupy two bytes, floating-point ones six bytes, and strings two bytes plus their exact maximum length in characters. Remember to allow for the 'zeroth' element of each dimension.

You must LINK_LOAD and use TURBO 'shared arrays', in memory, if you want several tasks to be able to modify array values held in common. Full instructions appear in the TURBO compiler manual.

(72) V_OPEN_IO - Opens an existing virtual array for reading and changes.

May be used in conjunction with any of the following virtual array routines. Expects channel No. and file name as parameters.

DCN, TOOLKIT

(73) V_OPEN_IN - Re-opens an existing virtual array for reading only.

Several tasks may share access to any virtual array, as long as they all open it with V_OPEN_IN.

DCN, TOOLKIT

(74) V_EXTEND - Pre-extends a virtual array file.

Used by ALL V_DIM routines.

DCN, TOOLKIT

(75) V_DIM1% - Sets up 1 dimensional integer virtual array.

Expects channel No., filename and maximum subscript as parameters.

TOOLKIT, DCN, CALLS:- V_EXTEND, WRITE_OK, YEA_OR_NAY

(76) NAME: V_SET1% - Sets a value in a 1D virtual integer array.

TOOLKIT, DCN

(77) V_READ1% - Extracts a value from a 1D virtual integer array.

TOOLKIT, DCN

(78) V_DIM1 - Sets up 1 dimensional float virtual array

TOOLKIT, DCN, CALLS: V_EXTEND, WRITE_OK, YEA_OR_NAY

(79) NAME: V_SET1 - Sets a value in a 1D virtual float array.

TOOLKIT, DCN

(80) V_READ1 - Extracts a value from a 1D virtual float array.

TOOLKIT, DCN

(81) V_DIM2% - Sets up 2 dimensional integer virtual array.

Expects channel No., filename and maximum subscripts as parameters.

Maximum subscripts are also required by other 2% routines and may be better set as GLOBAL values. This will avoid the need to pass them as parameters to other routines.

TOOLKIT, DCN, CALLS:- V_EXTEND, WRITE_OK, YEA_OR_NAY

(82) NAME: V_SET2% - Puts a value in a 2D virtual integer array.

TOOLKIT, DCN

(83) V_READ2% - Extracts a value from a 2D virtual integer array.

TOOLKIT, DCN

(84) V_DIM2 - Sets up 2 dimensional float virtual array.

Expects channel No., filename and maximum subscripts as parameters.

Maximum subscripts are also required by other 2D routines and may be better set as GLOBAL values. This will avoid the need to pass them as parameters to other routines.

TOOLKIT, DCN, CALLS:- V_EXTEND, WRITE_OK, YEA_OR_NAY

(85) NAME: V_SET2 - Puts a value in a 2D virtual float array.

TOOLKIT, DCN

(86) V_READ2 - Extracts a value from a 2D virtual float array.

TOOLKIT, DCN

(87) V_DIM3% - Sets up 3 dimensional integer virtual array.

Expects channel No., filename and maximum subscripts as parameters.

Maximum subscripts are also required by other 3D routines and may be better set as GLOBAL values. This will avoid the need to pass them as parameters to other routines.

TOOLKIT, DCN, CALLS:- V_EXTEND, WRITE_OK, YEA_OR_NAY

(88) NAME: V_SET3% - Puts a value in a 3D virtual integer array.

TOOLKIT, DCN

(89) V_READ3% - Extracts a value from a 3D virtual integer array.

TOOLKIT, DCN

(90) V_DIM3 - Sets up 3 dimensional float virtual array.

Expects channel No., filename and 3 maximum subscripts as parameters.

TOOLKIT, DCN, CALLS:- V_EXTEND, WRITE_OK, YEA_OR_NAY

(91) NAME: V_SET3 - Puts a value in a 3D virtual float array.

TOOLKIT, DCN

(92) V_READ3 - Extracts a value from a 3D virtual float array.

TOOLKIT, DCN

(93) V_DIM2$ - Sets up 2 dimensional character array (a 1D string arrray).

Supply the maximum subscript and maximum string length as parameters. The maximum string length is also required by other 2$ routines and may be better set as GLOBAL values to save passing parameters.

TOOLKIT, DCN, CALLS: V_EXTEND, WRITE_OK, YEA_OR_NAY

(94) V_SET2$ - Places text at specified position in a 2D character array.

TOOLKIT, DCN

(95) V_READ2$ - Returns text from a 2D character array.

TOOLKIT, DCN

(96) V_DIM3$ - Sets up 3 dimensional character array (a 2D string arrray).

Supply maximum subscript and maximum string length as parameters. The maximum string length is also required by other 3$ routines and may be

better set as GLOBAL values to save passing parameters.

TOOLKIT, DCN, CALLS: V_EXTEND, WRITE_OK, YEA_OR_NAY

(97) V_SET3$ - Puts text into a 3D character array.

TOOLKIT, DCN

(98) V_READ3$ - Returns text from a 3D character array.

TOOLKIT, DCN

(99) PIPED_MUSIC - Replays music as it is printed to a pipe.

Needs TURBO's support for pipe/channel re-direction.

TOOLKIT, SNG, MULTI-TASK, CALLS: PLAY_TUNE, EXTRACT_INT%, BUM_NOTE, BUMP_POSITION, PARSE_N_PLAY, SETUP_MUSIC.

(100) EXTRACT_INT% - Returns an integer parameter for PLAY_TUNE.

SNG, CALLS: BUM_NOTE, BUMP_POSITION

(101) BUMP_POSITION - Advances through a PLAY_TUNE string.

SNG

(102) BUM_NOTE - Indicates what PLAY_TUNE failed to understand.

SNG

(103) PARSE_N_PLAY - checks and plays one note for PLAY_TUNE.

SNG, TOOLKIT, CALLS: BUMP_POSITION, BUM_NOTE

(104) FORCE_FONT - Lets you choose the main font of ANY console window.

The routine asks for a font file name (e.g. MDV1_BOLD_FONT, from the TURBO TOOLKIT cartridge). It then sets the font of EVERY open console window in turn, displaying a message (in window 1) each time. The first three windows are generally SuperBASIC's £0, £1 and £2; any others may be intermingled or in groups by task, depending upon the order in which they were opened.

In between questions you can switch windows and see the effect of each font change (e.g. by pressing F4, which re-draws the screen in many packages). Switch back to channel £1 and type Y to change a channel, N to keep the previous font. To change the font of all the current windows, just type Y in reply to every question.

TOOLKIT, SNG, CALLS: READ_OK, YEA_OR_NAY

(105) PLAIN_ENGLISH - Replaces the QL's repertoire of messages.

Needs the TRA command implemented in late (post AH and JM) versions of the QL ROM. The new messages are a little idiosyncratic, but they certainly convey more information than the originals did. You can change the text to suit yourself by altering the DATA statements.

TOOLKIT, SNG

QL TURBO TOOLKIT USER MANUAL

Published by Digital Precision Ltd, London E4 9SE.

CONTENTS

QL TURBO TOOLKIT USER MANUAL

Revision 1, published by Digital Precision Ltd, London E4 9SE.

CONTENTS

An overview of this manual

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. Many example programs and routines appear in the text, and there are many more in the demonstration file TURBO TEXAMPLES. Load this file and try out the routines therein - if the purpose of a routine is not obvious it will be explained by comments in the program just before the code.

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

Many 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.

Copying Turbo Toolkit

You can make a backup copy of Turbo Toolkit, for your own use, on any device To do this, put the original cartridge in drive 1 and type:

```
LRUN MDV1_BACKUP        (or FLP1_BACKUP if you are using disks).
```

When the program runs, type the name of the destination - the drive which you want to copy to (e.g. MDV2). Put a disk or cartridge in the relevant drive Type Y if you want the disk to be formatted, or N if it is already formatted and you do not wish to re-format it. There should be room for all the Toolkit files on the destination medium.

Some files in the Toolkit need to know the drive they are loaded from. The last question asks if you intend to use the copy in the drive presently containing the original. Type Y to obtain an exact copy of the original, or N for a version converted to run from the destination drive.

PLEASE don't copy Turbo Toolkit for others. Unless users are honest the QL software market will wither and die. If you are offered, or hear of, an unofficial or 'pirate' copy, please remember that D.P. pay substa tial reward for information leading to the successful prosecution of software thieves.

Improvements and upgrades

Digital Precision continuously develops and improves its products. Please read the Quill file UPDATES_DOC, on your Turbo Toolkit cartridge, to find out about any changes and enhancements that have been made since this manual was printed.

Error detection and trapping

Commands: DEVICE_STATUS, DEVICE_SPACE, WHEN_ERROR, END_WHEN, RETRY_HERE, ERLIN, ERNUM

Effective error trapping has been neglected in the implementation of SuperBASIC. Late models of the QL have undocumented error trapping commands, but these are bug-ridden and unavailable on most machines.

Turbo Toolkit contains an ingenious function, DEVICE_STATUS, which addresses this problem. The function 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 needed DEVICE_STATUS in order to write SUPERCHARGE properly!

The original DEVICE_STATUS simply took a string and tried to use it as a new output channel name. This worked fine, but it did not provide much information if the file specified already existed on the device.

After much lobbying by Chas Dillon (one of the world's most brilliant Supercharge users) DEVICE STATUS has been re-designed, with an extra optional parameter to make explicit your desire to read or alter a new or familiar file.

If your name is valid DEVICE_STATUS will tell you how much room there is on the device for extra data. It will also check that the file CAN be modified in case it is 'in use' or 'read only'. Advanced users should note that this check updates the 'last change' date on file-structured devices that support date-stamping, unless access-type 1 is used.

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

We have designed our own asynchronous error-trapping scheme; a refinement of that offered on late QLs, but version-independent, secure and more sophisticated. Three error-trapping directives and two functions are in the TURBO TOOLKIT. These are ignored by the interpreter, but they will allow ANY error to be trapped - on ANY version of the QL - if the TURBO compiler is used.

DEVICE_SPACE

This function expects one parameter - the number of any channel open to an file on a file-structured medium (usually floppy disk or microdrive). The channel number may be preceded 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.

File storage space is allocated in 'granules', typically 0.5K / 1.5K, not byte at a time. Unused parts of a granule are not counted as free space. Some bytes are used to store the 'header' of each new file. Also granules can be allocated to 'directory' information as a new file is created. For these reasons a little space may be allocated but unused when drives become full.

WHEN_ERROR, END_WHEN, RETRY_HERE, ERLIN, ERNUM

These statements allow a heirarchy of asynchronous error-trapping routines to be declared and managed. They need the power of TURBO to bring them to life. The statements are ignored in interpreted and SUPERCHARGEd programs.

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 corresponding 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 device - 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 be 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 SUPERCHARGE users will recognise this as the 'old' DEVICE_STATUS. The result will be a negative number if this is not allowed; otherwise it will be positive value - the number of unallocated bytes on the device.

DEVICE_STATUS (-1 , NAME$)

This is the most versatile type 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 corrupting 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 error code is returned if anything goes wrong.

DEVICE_STATUS automatically adapts to different hardware, so you can use it on basic QL system, secure in the knowledge that it will also work with floppy disks, modems, '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")      <- Command
-20                                            <- Result
```

There's still one fly in the ointment - DEVICE_STATUS cannot detect a write-protected microdrive cartridge. This is because of a fault in the QL's microdrive device driver, which ignores the write-protection when opening a file but fails when it tries to perform the requested operation, giving a 'Bad or changed medium' asynchronously a minute or so later. This is classic Sinclair 'broken belt-and-braces' programming!

It is theoretically possible to avoid this, as the second processor can detect whether or not the currently turning microdrive is protected, but this can't be done by a user program as the device driver can asynchronously switch from one microdrive to the other. The only reliable fix is to alter the ROM of the microdrive handler... so start petitioning Tony Tebby now!

It is also 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$(1 TO 3)="NET":RETurn -7:ELSE RETurn DEVICE_STATUS( 1 , NAME$ )
```

DEVICE_STATUS returns a floating point value corresponding to a long integer, indicating the degree of success it had in opening a channel. These are standard QDOS codes; corresponding messages can be printed with the REPORT keyword supported by some QLs. For example REPORT -20 prints "read only".

The numbers are tabulated below:

VALUE RETURNED	MEANING
0 or more	The device exists, and is not busy; a file with the name specified (if any) does not yet exist. The name or other parameters (if any) are valid. The value is the number of free bytes on the device, or a large number (65535 * 512) for 'endless' serial devices.
-3 or -6	The device name and parameters are valid, but the QL has not got enough free memory to open a new channel to the device.
-7	There's no device (or file) with that name on this QL.
-8	The specified file exists 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 - see the notes on DEVICE_SPACE.
-12	The device is valid, but the file name or parameters are not.
-14	Bad or changed medium; the medium in the device is faulty, or has been changed while the system was writing to a file.
-20	The specified file exists and may be read but not altered, as the device is write-protected or is being read.

TABLE OF VALUES RETURNED BY DEVICE_STATUS

ING TURBO TOOLKIT IN COMMERCIAL PROGRAMS

In order to allow software developers to use Turbo Toolkit commands in applications software, without harming sales of the Toolkit, a special 'UNTIME' version is supplied. This may be distributed without restriction, whereas all other Toolkit files are copyright and may not be sold or given to other users without specific written permission from Digital Precision.

The runtime Toolkit conveys extra advantage to developers - expecially if they are targetting their software at a standard QL - because it leaves up to 1K (depending upon what is previously loaded) more free memory than the standard version.

The runtime toolkit is in the file RUNTIME_EXTS on your original toolkit disk or cartridge. It can be loaded by MERGing RUNTIME_BOOT.

Please note that it is not possible to test programs under the runtime Toolkit by loading RUNTIME_EXTS after TURBO_TK_CODE; the commands will become multiply-defined and system behaviour will vary depending upon the QL version. Compiled programs will consistently use the FIRST (non-runtime) keyword definition, so loading the runtime version later will have no effect! You MUST reset your computer and then load the runtime version on its own - without having loaded TURBO_TK_CODE - if you want to test the performance of your code under RUNTIME_EXTS.

Runtime command support

The runtime version allows developers to call Toolkit routines from within compiled programs, but restricts interpretative use. All of the toolkit commands and functions are supported from within Supercharged or Turbocharged code, with the exception of these compiler directives:

PROCEDURE, FUNCTION, IMPLICIT%, IMPLICIT$, EXTERNAL, GLOBAL, DATA_AREA, REFERENCE, WHEN_ERROR, END_WHEN, SNOOZE, CHARGE, PROCEDURE, FUNCTION, OPTION_CMD$, ERLIN, ERNUM

These are all ignored by Supercharge and translated into compiler-specific code by TURBO, so you may still use all of these features in commercial Turbocharged programs. You must have the standard Toolkit loaded when you compile programs using these words. The resultant tasks will run under the runtime Toolkit, even though the words are not defined.

In general, any attempt to use Toolkit commands in interpreted BASIC when the runtime Toolkit is loaded will cause a BAD PARAMETER or NOT IMPLEMENTED error report. However, to allow applications to start up reasonably easily, the command-file keyword END_CMD will work as normal, and EXECUTE/_A/_W and LINK_LOAD/_A/_W will work as long as their only parameters are file name strings. Names must be in quotation marks and channel numbers, parameters and priorities are only supported in compiled programs. If these facilities are vital they must be provided through a startup task which loads others appropriately.

****

A QUILL FILE CALLED UPDATES_DOC IS SUPPLIED ON THE CARTRIDGE. IT IS IN YOUR INTERESTS TO READ IT AS IT CONTAINS INFORMATION SUPPLEMENTARY TO THAT GIVEN IN THE MANUAL...

****

* PAGE 2

Replace the line starting 'base=RESPR..' with this command:

```
MERGE MDV1_TK_BAS
```

4th Para, 2nd Line; replace start of line with this text:

```
the file MDV1_TURBO_TK_CODE & linked into
```

* PAGE 6

3rd Para - extend line:

SuperBASIC channel number. As long as you're careful, this can be an advantage!

There's a missing HASH sign in the paragraph about POSITION.

Add a paragraph at the foot of the page:

BEWARE: Current microdrive handlers fall over, with a 'Bad or Changed Medium' error, if you try to SET POSITION beyond about 1.6E7. This is a system bug.

* PAGE 34

Correct heading:

Commands: IMPLICIT%, IMPLICIT$, DATA_AREA

Further down the page, replace DATASPACE with DATA_AREA

* PAGE 35

Replacement heading:

FINDING MEMORY REQUIREMENTS

Functions: FREE_MEMORY, DATASPACE

Add this paragraph at the foot of the page:

The DATASPACE function finds the amount of data space associated with a task file. The result is the number of bytes of dataspace, or -2 (Invalid Job) if the file is not a task. This command indicates that BOOT is not a task:

```
PRINT DATASPACE("MDV1_BOOT")
-2
```

* UTILITIES; PAGE 21

Replace two POKEs in the third paragraph:

You just SET the details of each

also SET different font details

* UTILITIES; PAGE 24

Typo!!! Paint out the word 'use' near the end of the third paragraph.

Amend the last line of para. 6 to read as follows:

otherwise, to others. You may also copy RUNTIME_EXTS freely.