Q_LIBERATOR USER MANUAL

Second edition February 1988
Updates for Release 3.3 March 1990

Valid for Budget Release 1.0 and Release 3.3

Whilst all reasonable care has been taken to ensure that Q_Liberator does not contain errors and that the documentation is accurate, in no circumstances will Liberation Software be liable for any direct, indirect or consequential damage or loss arising out of the use or inability to use the software or its documentation.

Liberation Software has a policy of constant development and improvement of its products. Updates to this manual will be available for later releases.

Liberation Software
43 Clifton Road
Kingston upon Thames
Surrey
KT2 6PJ

IMPORTANT NOTE

In chapters 1 to 14 of this manual, reference is made to Release 3.2 of the compiler. All the information is also valid for Release 3.3 except where the text is marked with a vertical line in the right margin. This indicates that 3.3 contains enhancements to the Q_Liberator specification. These are explained in detail in chapter 15. Please note that the index currently excludes references to this chapter.

Thank you for buying Q_Liberator. Please complete the following registration form and return it to :

Liberation Software
43 Clifton Road
Kingstopn upon Thames
Surrey KT2 6PJ

Name _____ Q_Liberator Release _____

Jress _____ Q_L ROM Version _____

_____

_____

_____

Please describe your hardware (disks, memory etc)

Where did you hear of Q_Liberator ?

Do you read QL WORLD regularly ?

Are you a QUANTA member ?

Which other Q_L compilers do you use ?

w do you think Q_Liberator compares ?

Which features do you like and why ?

Which don't you like ?

How useful did you find the manual ?

Do you use QLOAD and QREF ?

Please use the space overleaf to write any other comments on Q_Liberator. We are particularly interested in features would you like to see in future versions.

Chapter 1 Introduction

WHY A SUPERBASIC COMPILER ?

SuperBASIC is an elegant, flexible language designed by programmers for programmers. It is a considerable advance on other implementations of BASIC and contains some unique features. It is ideally suited to be the QL's native language.

It is also somewhat slow, and gets slower as programs increase in size. Programs can take an age to load and there is no possibility of running more than one SuperBASIC program simultaneously.

To solve these problems we decided to write a SuperBASIC compiler. We did not wish to deny the programmer any of SuperBASICs more exotic features and so the major design aim was to adhere rigidly to the SuperBASIC syntax. There seemed little point in supporting a subset similar to that of our competitors.

The result was the first version of Q_Liberator - now available as our budget version. It is a sophisticated tool which produces compiled programs (known technically as object programs) from a SuperBASIC program. Object programs load in a fraction of the time and execute many times faster than the interpreted original. Furthermore such programs can be multitasked ie several programs can be run simultaneously and all are secure from prying eyes because Q_Liberator programs are indecipherable when examined.

With a few well documented (and obvious) exceptions Q_Liberator can compile virtually any SuperBASIC program. There is not normally any need to change the original program.

EXTRA FEATURES

Q_Liberator is much more than just a tool to create faster programs; the SuperBASIC extensions also supplied provide access to facilities within the QL which until now have been denied to SuperBASIC. In particular full error trapping can be included in programs regardless of ROM version, and the interesting possibilities of inter-job communication through various means including pipes can be explored.

SYSTEM REQUIREMENTS

Q_Liberator has been designed to be fully useable on any QL compatible hardware. Q_Liberator and the programs which it compiles will work with all extension disk systems which adhere to the standard QL format and will happily coexist alongside any well behaved software.

Special provision has been made in the budget version to ensure that large programs can be compiled and run on an unexpanded QL, but if extended memory is available then Q_Liberator will exploit it.

Q_LIBERATOR RELEASE 3.2

This version of Q_Liberator includes all the features of the budget compiler but with many improvements in speed and functionality. It has a completely different menu driven 'front end' and adds a major new facility - the ability to build libraries of compiled procedures or functions. Such procedures can be loaded in a variety of different ways and can be called by either interpreted or compiled programs.

Release 3.2 is larger than the budget version and is best used with at least 256k of memory. An alternative for 128k users is to buy the ROM version of 3.2. This 16K ROM cartridge contains the QLIB runtime system and most of our ancilliary SuperBASIC extensions. This frees RAM for larger compiled programs, and gives the advantage of even faster execution. In all other respects it is identical to an entirely RAM based version.

PACKAGE CONTENTS

The Q_Liberator package comprises this manual, a registration form and a master microdrive (or floppy) containing all the Q_Liberator software. The budget master can be identified by a green label whereas the release 3 master has a red label. The ROM version also includes a 16k ROM cartridge which plugs into the rear of the QL. (Power off first!)

The master copy should be kept in a safe place and used only for creating working copies as described below. Replacement, except in cases of faulty materials, is chargeable. If you ever order an upgrade from us, please return only the master with your remittance (ie NOT the manual).

The files on the master vary between releases and are fully described in appendices A and B.

MAKING WORKING COPIES

Unlike the earliest versions of Q_Liberator, there is no copy protection on current releases. You can freely copy the files to any media but please, for your own use only. There is a CLONE program on the master which will copy the QLIB files and optionally the DEMO and BOOT files to a device of your choice (including RAMdisk) to produce a working copy.

CLONE is supplied as a BASIC program and is straightforward in use. It expects the target media to be be pre-formatted (but not necessarily empty). If a microdrive copy is being produced it is wise to format the cartridge in the device in which it will be used to minimise bad media errors.

CLONE first prompts for the name of the device which contains the master and shows the default entry mdv2_ in parenthesis. This can be accepted by hitting ENTER, or an alternative drive name entered.

After specifying the name of the target device in similar way, CLONE asks for the name of the load device for the part of the compiler called QLIB_OBJ. Enter the name of the device in which this Q_Liberator copy will be used. This will probably be the drive the copy is being produced in. This entry is necessary because Q_Liberator is organised in 2 parts. The first part needs to know where to find the second part is.

You will then be asked if you want to copy the demonstration programs contained on the master. Do so for your first working copy so that you can work through the examples in this manual.

The last question inquires if you want to create a standard BOOT program on your working copy. It is wise to do this for your first experiments. Later you may wish to adapt an existing BOOT program to include loading the Q_Liberator system. If you have a ROM based Q_Liberator then the BOOT program is superfluous as the extensions which BOOT loads are already resident in the ROM.

CLONE then proceeds to copy each of the files from the master to the new working copy. When it is complete, reset the QL and try loading from it.

CREATING A RAM DISK COPY

If you want to configure Q_Liberator to load from a RAM disk, put the master in mdv2_, set the working copy as mdv1_, and the load device as ram1_. After the CLONE is complete, you can modify the BOOT to load QLIB_OBJ into the RAM disk on start up.

CONFIGURING RELEASE 3

The configuration process which takes place in the CLONE procedure, can be further modified with the procedure QLIB_USE described in chapter 14 for Release 3 users only. This should only be attempted when you are completely familiar with Q_Liberator.

USING THIS MANUAL

This manual describes both Budget Q_Liberator and Release 3.2. All information is relevant to both releases unless otherwise stated. Minor differences are explained in the text, but where there are major differences there is a separate section describing the features of each version. An entire chapter is devoted to the unique features of release 3.2.

We intended this manual to be suitable both for those who are unfamiliar with the concepts of compilation and multitasking, and the more advanced user, who will hopefully find many stimulating ideas.

A working knowledge of SuperBASIC is assumed. Throughout the text there are many examples; you are encouraged to try these to aid your understanding.

COMMERCIAL USE

Those who wish to market programs compiled with Q_Liberator are free to do so provided that :

Credit is given to Q_Liberator and Liberation Software within the program or accompanying documentation.

Liberation Software is notified of all such programs.

Any Liberation Software extensions such as those in QLIB_EXT, are linked to the object program. (see chapter 9.

It is preferable to link the runtime system QLIB_RUN to the object program to produce a stand-alone file (see Chapter 4). If however the product comprises several compiled programs this can be inefficient in memory usage. For such cases the runtime system can be supplied as a separate file.

The parts of Q_Liberator contained within commercial programs remain the intellectual property of Liberation Software at all times.

No other part of the Q_Liberator system may be distributed in any form.

REFERENCE MATERIAL

The best book for those who wish a fuller description of SuperBASIC than that provided by the QL User Guide is:

QL SuperBASIC - The Definitive Handbook
by Jan Jones, designer and writer of the language.
Mc Graw Hill 1985 ISBN 0-07-084784-3

This book proved indispensible during the creation of Q_Liberator. The language described therein is followed precisely except where documented in this manual.

CREDITS

The original Q_Liberator was designed and written in many long evenings between April 1985 and September 1986. It was a joint project and lent itself well to creation by a team of two.

Adrian Soundy was mainly responsible for the compiler, which itself was written in SuperBASIC then compiled, whilst I, Ian Stewart wrote the runtime system and the manual.

Thanks are due to Leon Jaeggi for relentless bug hunting and much support, to Tony Tebby for useful tools and challenging test material and to my wife Julia.

The Toolkit referred to throughout this text is the QJUMP Toolkit II, available from CARE Electronics. Many of the features mentioned are present on the earlier QL Toolkit from Sinclair.

Chapter 2 Getting Started

The aim of this chapter is to teach you enough about Q_Liberator to compile a short SuperBASIC program and to run the compiled version.

First, if you have not already done so, create a working copy of Q_Liberator with all demo programs present by loading and running the CLONE program. Now reset your QL, place the working copy in MDV1_ (or FLP1_) and press F1 or F2 as you see fit. The BOOT program on the microdrive will automatically load all the necessary Q_Liberator files. In channel 0 you will see the Q_Liberator copyright message briefly appear. Your system is now ready to compile a program.

If you have a ROM based Q_Liberator the BOOT program is unnecessary. Simply insert the ROM with the QL powered off, then turn it on. The copyright message will be displayed on the initial screen above the F1 F2 prompt. You will still need the working copy in MDV1 to load the remaining parts of the compiler.

Q_Liberator takes as its starting point a working SuperBASIC program which has been LOADed into memory. This is referred to as the *source program*. For the demonstration we will use a small program supplied on the master which sorts integers, strings or floats.

Type LOAD MDV1_SORT_DEMO

and wait for the cursor to reappear. Now type RUN and watch the screen. All being well you should see the demonstration sort program being put through its paces. Wait until it is complete and make a note of the times which are displayed.

Now we are ready to see Q_Liberator in action.

COMPILING A PROGRAM

Q_Liberator compiles programs in two distinct *phases*. The first phase does some initial checking and produces a *work file* for use by the second phase.

The second phase does all the detailed work of checking the program for errors and produces an *object program*. An object program when executed behaves in the same way as the original source program, but loads and runs much faster. Furthermore it can *multitask* ie run concurrently with other programs.

The two compiler phases can be run independently of each other or, providing there is enough memory, they can run automatically one after the other. We shall use the automatic mode for the first demonstration.

Type LIBERATE MDV1_SORT_DEMO,

Take care to type in the comma at the end as it is this which causes the two phases to follow each other. If you did forget it, don't worry; just retype the line.

You should now see the message "Creating work file" in channel 0 and hear MDV1 spinning. The work file (its name is MDV1_DEMO_SORT_wrk) will occupy much the same amount of space on the microdrive as the source program. Once it has been created, the source program is no longer necessary for Q_Liberator to complete its job.

After a few seconds you will see the message "Loading Q_Liberator" in channel 0. The second phase which does most of the work is now being loaded. Phase 2 is itself a multitasking Q_Liberator program, so while it is running you will still see a cursor in channel 0 and can continue to use SuperBASIC if you wish.

THE Q_LIBERATOR SCREEN

When loading is complete you will see the main Q_Liberator screen. At first the top window will contain only the product name, but shortly you will see the SuperBASIC line number which Q_Liberator is currently processing displayed in the top right hand corner. Shortly after this number has reached the maximum line number in the program the compilation is complete.

The results of the compilation are displayed in the lower window. Here you will see the size of the program, the amount of data area required, the highest channel used and the compile time (phase 2 only). The demonstration program as supplied compiles perfectly (of course) but if there had been any errors then they too would have been displayed in the lower window.

RUNNING THE COMPILED PROGRAM

After compilation , you can try running the object program which Q_Liberator has produced. The object program name has the extension '_obj' appended to it.

Type EXEC_W MDV1_DEMO_SORT_OBJ

to load and execute the compiled sort program. After a very brief loading time you should see the sort program running again, but this time much faster. We used EXEC_W because this ensures that *only* the sort program is running. You could also use EXEC to start the program in which case both the compiled program and the SuperBASIC interpreter will run simultaneously. You can switch the keyboard between the two programs by pressing control-C. This operation which will already be familiar to many users is explained in more detail in the next chapter. When EXEC is used the times for the compiled program are slightly longer, because the SuperBASIC interpreter is still active and using some processor time.

INTRODUCTION TO QX

The procedure QX offers an easier way to load and start object programs, because there is no need to specify the extension '_obj'. In other respects it behaves similarly to EXEC. Its companion, QW is similar to EXEC_W.

Try QX MDV1_DEMO_SORT

Whilst the sort is running, try typing some SuperBASIC commands to show that the interpreter and the compiled program are indeed running simultaneously. QX and its companion QW have other uses, explained fully in chapter 10.

SEPARATING PHASE 1 AND PHASE 2

When there is insuffucient memory to hold both the source program and phase 2 in memory simultaneously, the program can be compiled in separate phases. The first phase is started by typing

LIBERATE MDV1_DEMO_SORT

This simply creates the workfile then stops. You can now type NEW to clear all the memory used by the interpreter before starting phase 2 by typing just

LIBERATE

This has the effect of loading Q_Liberator which will then wait to be told what to do. First press control C to switch the cursor to the compiler window if it is not already there. What you do next to start compilation differs between the 2 compiler versions.

With the BUDGET release the compiler will be expecting a command line to be typed. In its simplest form this is just the name of the program to compile. Type

MDV1_DEMO_SORT

and you will see the budget compiler running as before.

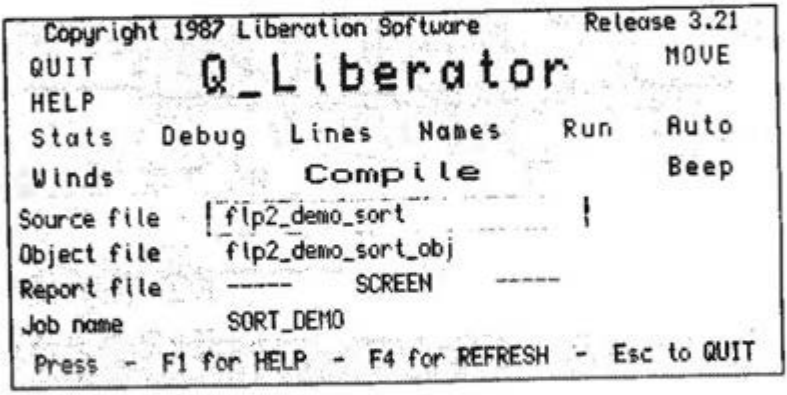
With RELEASE 3.2 you will see a more complex screen with a number of menu item boxes. The box marked 'source file' on the screen will have a highlighted outline. Press SPACE to select this box, type the file name MDV1_DEMO_SORT terminated by ENTER into the box, then press C (for Compile) to start compilation. If you find this confusing, don't worry. All will be explained in chapter 4.

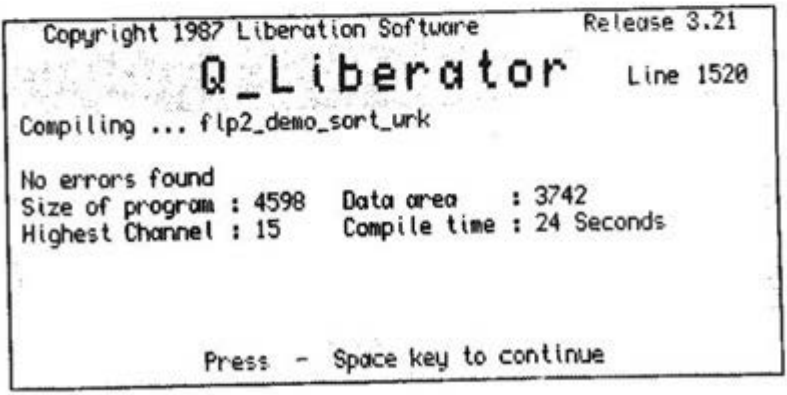
With either release, when Q_Liberator is finished you have the opportunity to compile another program or to end Q_Liberator. With the BUDGET version, just enter an empty line to end; with RELEASE 3.2 press the ESCape key.

The workfile created by phase 1 is automatically deleted at the end of a compilation. If however the compile fails because of lack of memory, it will remain on the microdrive. It is not changed in any way during phase 2 and so can be resubmitted to the compiler when more memory is available.

You should now have enough information to begin compiling your own programs, but beginers in particular should read the next chapter which gives some guidelines to ensure they will run successfully in a multitasking environment. Detailed compiler operating instructions are in chapter 4, while chapters 5 and 6 describe the error messages which you may encounter.

Release 3.2 owners may find it instructive to play with the menu system and read some of the online HELP information which is available through the F1 key before proceeding further.





Chapter 3 Fundamentals

The Motorola 68008 chip inside your QL can only execute its own machine code instructions; it cannot execute SuperBASIC programs directly. Therefore, before a program can be executed, the SuperBASIC instructions must be translated into another form. There are two types of program which can perform such a translation; *interpreters* and *compilers*. This chapter explains the essential differences between them. It also contains an introduction to multitasking and advice on writing programs designed to execute in the multitasking QDOS environment.

THE SUPERBASIC INTERPRETER

When a SuperBASIC program is LOADed, the interpreter translates the program text which it reads from the microdrive into an internal program format. The names of all variables, procedures and functions are put into a name table and memory is allocated for the program to use.

This process takes time and is the reason why SuperBASIC programs take a long time to load.

When you LIST a program the interpreter converts the internal format back to a text format which can be displayed on the screen.

When you type RUN the interpreter starts to translate the program in memory line by line. Executing a simple statement can involve many hundreds of machine code instructions, most of which are spent determining just what is to be done. The actual operation, accounts for relatively few instructions.

If a statement is placed inside a FOR or REPeat loop then each time round the loop the interpreter must retranslate the statement.

The interpreter keeps track of the location of each procedure, function, loop etc, by means of the number at the start of each line. Finding a line number involves searching from the current line all the way to the target line. This process gets progressively slower as program size increases.

THE Q_LIBERATOR COMPILER

In contrast, Q_Liberator takes the internal form of the program and translates it once at *compile time*, creating a new file called an *object* file.

In the object file, references to line numbers, procedures, loops etc are absolute, ie the program knows where everything is and searching is unnecessary.

During compilation, Q_Liberator performs all the work of deciding what has to be done to execute a given statement. Thus when the program is executed at *runtime* it runs much faster.

The object program can only run in conjunction with the *run time system.*

This is either pre-loaded by a BOOT program or linked to the object program at compile time.

Q_Liberator programs also load much faster than SuperBASIC programs because no translation takes place during loading.

It is important to realise that Q_Liberator in no way replaces the interpreter. In fact they complement each other, resulting in a more sophisticated working environment. The interpreter becomes the ideal program development tool offering the advantages of interactive operation, whilst Q_Liberator ensures that the finished product loads and runs efficiently.

MULTITASKING

The QL is rare amongst low cost micros in that its operating system, QDOS, is inherently *multitasking.* This means that more than one program can run on the machine simultaneously. A multitasking program in QL parlance is termed a *job.* Q_Liberator identifies jobs by their number or their name.

The SuperBASIC interpreter, together with the program it is interpreting constitute job 0. Job 0 is unique in that it can grow or shrink in size as necessary, and can never be removed.

QDOS manages jobs by allocating each job some processor time in turn while the job is active. The amount of time which a job gets is determined by its *priority* . The priority can range from 0 to 255; 0 means that a job is inactive, and gets no processor time. When changing priorities it is the relative difference in priority between two jobs that matters, not the absolute priority.

If a job is just waiting for a keyboard input, then it is not using processor resources.

JOB CONTROL

The Q_Liberator package contains a number of procedures to manage jobs. These let you display which jobs are running, remove jobs and change the priority of jobs. You may already have similar procedures and know how to use them. If not, you may find it useful to read chapter 12 so that you can experiment with the procedures within the example which follows.

You can, for example, see the effect that changing a job's priority has on the rate at which it counts. Try this when there is more than 1 job running.

A MULTITASKING EXAMPLE

The following short program is contained on the Working copy. It is useful for illustrating some aspects of multitasking.

```
100 REMark MULTITASKING DEMONSTRATION
105 :
107 REMark DEMO_MULTI
108 :
110 j=Q_MYJOB: REMark see chapter 12
120 REPeat loop
130   AT j,0:PRINT FILL$(" ",20)
140   AT j,0
150   INPUT ("JOB "&j&" >");a
160   IF a=0 THEN STOP
170   FOR x=1 TO a
180     AT j,15: PRINT x
190   END FOR x
200 END REPeat loop
```

This program simply prints its job number and prompts for a number to be entered. If this number is 0, the program ends; otherwise it counts from 1 to the number given, whilst displaying the current figure on the screen. The position on the screen is determined by the job number.

Type QX MDV1_DEMO_MULTI

to start a copy of this program. You will see the prompt "JOB 1 >" at the top of the screen with a non flashing cursor beside it. The job is waiting for keyboard input. Note that there is still a flashing cursor in channel 0, and you can still use SuperBASIC.

There is only one keyboard on the QL, but there may be many jobs waiting for keyboard input. QDOS provides a mechanism whereby you can effectively attach the keyboard to different jobs as required. This is done by pressing Control-C (hold down CTRL and press C). Doing this makes the flashing cursor move to the next job which is awaiting input. If you keep pressing Control-C you can select any job which is awaiting input.

Select the cursor for job 1 and type in a number, say 1000. You will see the program count from 1 to 1000. Now select the SuperBASIC cursor in channel 0 and start a few more copies. Use Control-C to select each in turn and set them all counting simultaneously. Notice how the rate of counting slows down as more and more jobs are started.

Try changing a job's priority to see the effect this has on the rate at which it counts, particularly when there is more than 1 job running.

When you are finished, end each program by entering 0.

ADAPTING PROGRAMS TO MULTITASK

Not all programs will be suitable for mutitasking because when several programs are running simultaneously they compete for the QL's resources. You need to take this into account when a program is intended to multitask.

KEYBOARD HANDLING

There are three different ways of reading the keyboard in SuperBASIC, and each behaves in a different fashion when multitasking.

If a program uses an INPUT statement, there is no problem because INPUT always puts a cursor on the screen.

The same is not true for INKEY$. You can only divert characters to a program using INKEY$ if you have first enabled a cursor on the channel used by INKEY$. You can do this either by placing an INPUT statement for the same channel at the start of the program, so you can switch the keyboard to it then, or you can use the Q_CURSON procedure (or equivalent) as described in chapter 12.

It is important to consider this point when compiling interactive games which use INKEY$. The alternative is to use EXEC_W or QW to start the the job so that it runs on its own. Then you are guaranteed sole use of the keyboard.

The final method of reading the keyboard is to use the KEYROW function. KEYROW does not care which job the keyboard is currently attached to. It bypasses this mechanism and reads the keyboard directly. Care is necessary when using KEYROW as the program will treat all keystrokes as its own making it difficult to type characters intended for other jobs. It is best to either run such programs on their own, or use obscure keystrokes to minimise interference.

SCREEN HANDLING

When you have several programs all using the same screen, the result can be chaos because each job is free to overwrite another job's windows. In such cases it is useful to separate the windows on the screen. In practice, at any given time most jobs will simply be waiting for keyboard input. If possible, try to include a routine in the program which can redraw the screen when necessary.

FILE HANDLING

Microdrive files can be shared by several jobs, so long as all the jobs open the file using OPEN_IN. Other forms of OPEN grant a job exclusive use of the file; subsequent OPEN attempts by other jobs will cause an 'in use' error. The error trapping procedures described in chapter 11 let you catch such conditions.

The same is true for devices such as a printer attached to ser1 - only one program can normally OPEN it. (In fact you can get round this problem of exclusive files easily, by sharing channels as described in chapter 10).

Chapter 4 Using Q_Liberator

This chapter gives a detailed description of how to use both Q_Liberator compilers. You will already have seen how easy this can be if you have worked through the demonstration run.

As stated earlier, Q_Liberator compiles programs in two distinct phases. The first phase produces a work file, generated from the SuperBASIC program which is currently loaded. The second phase processes the work file to produce an object file. Phase 1 is implemented as an assembler procedure while phase 2 is in fact a Q_Liberator object program.

Remember that an object file requires that the runtime system, QLIB_RUN is present before it can be executed. QLIB_RUN can either be loaded once into the resident procedure area and shared by several object programs simultaneously, or an individual copy can be linked to the object program.

PHASE 1 - PRODUCING A WORKFILE

Before attempting a compilation, you must load the SuperBASIC source into memory and ensure that your program will run correctly in its interpreted form. Q_Liberator cannot be expected to fix programming errors for you! If you have an unexpanded machine, it is wise to type CLEAR to free any available memory before starting compilation.

Q_Liberator phase 1 is started by using the LIBERATE command in the following form:

```
LIBERATE filename
```

'Filename' specifies the name Q_Liberator will use when forming the work file name and later in phase 2, the object file name. The work file name will be 'filename_wrk' and the object name 'filename_obj'. Normally you will have to specify filenames in full unless you have the Toolkit 2 extensions to support default directories.

LIBERATE prints the message "Creating workfile" on channel 0 while it is busy. When the cursor reappears the work file is complete. The work file is a complete representation of the BASIC program in an internal format. After this, the original BASIC source program is no longer needed for the compilation process and can be removed if required by typing NEW.

As an example, to produce a work file of the program MDV1_DEMO_SORT on MDV2, you would type:

```
LOAD MDV1_DEMO_SORT
LIBERATE MDV2_DEMO_SORT
```

If you want to compile a really large program on an unexpanded system,

then you need only load the file QLIB_BIN to the resident procedure area when you boot the system. This contains the LIBERATE procedure and occupies only about 2k of memory. After producing a workfile, clear the program using NEW (or reset), load the runtime system QLIB_RUN and then start phase 2 as described below to complete the compilation.

LIBERATE must always be typed as a direct command. It is not meaningful to use it within a program. If you attempt this you will get the error 'bad name'.

PHASE 2 - PRODUCING AN OBJECT FILE

Phase 2 of the compiler needs to know what work file it is to process and any special instructions regarding compilation. It obtains this from a command line. A command line is a string of characters corresponding to the following format:

Work_file_name [option_list]

At a minimum it contains just the name of a work file as produced by phase 1. This should be specified without the '_wrk' suffix. The option_list specifies which particular compiler features are to be enabled or disabled. The complete list of options is described later in this chapter.

There are several ways of passing a command line to phase 2. One method is to execute the compiler and pass the command line in a command string. Command strings are described in detail on page 10.2. For example,

```
QX MDV1_QLIB,"MDV2_DEMO_SORT"
```

would complete the compilation of the workfile produced in the earlier example. Phase 2 would be loaded from MDV1 and then would process the file "MDV2_DEMO_SORT_wrk" producing the object file "MDV2_DEMO_SORT_obj". At the end of the compilation the work file is deleted.

When no option list is present, a default set of options is used. As an example of an option list, the option -OBJ could be used to change the name of the object program. For example,

```
QX MDV1_QLIB,"MDV2_DEMO_SORT -OBJ MDV1_FREDDY"
```

would create an object version of the sort demo called MDV1_FREDDY_obj.

COMBINING PHASE 1 AND PHASE 2

When there is sufficient memory to hold both the original program and phase 2 of the compiler in memory simultaneously, the two phases of compilation can be combined. This is initiated by placing a comma after the filename in a LIBERATE command. If required an option list can be placed in a string following the comma.

Thus the format is

```
LIBERATE filename,
LIBERATE filename,"option_list"
```

Think of the first form as having an empty option list. Internally, LIBERATE first produces the workfile as before, then combines the filename and the option list to produce a command string. Phase 2 is then executed and passed this command string.

While phase 2 is being loaded the message "Loading Q_Liberator" is displayed on channel 0. The device from which phase 2 is loaded is configured when you make a working copy with the CLONE program. With release 3.2 it can also be configured by the QLIB_USE procedure described in chapter 14. Alternatively the release 3.2 compiler can be made resident in which case no loading is necessary.

For example,

```
LIBERATE MDV2_TEST,"-OBJ MDV1_FREDDY"
```

would produce the file MDV1_FREDDY_OBJ in one operation from whatever SuperBASIC program was currently loaded.

COMMAND LINE ERRORS

If you make an error in a command line, Q_Liberator will print the bad line on the listing channel with an arrow pointing to the part in error. You will see one of the following self explanatory messages:

```
No source file
Option name expected
Parameter expected
Invalid option
```

INTERACTIVE CONTROL OF PHASE 2

So far, each time we have used phase 2 it has compiled one program according to the command line passed to it, then aborted. An alternative method is available by starting phase 2 without passing a command line. This can be done by typing LIBERATE with no parameters whatsoever or QX MDV1_QLIB without a command string. In either case phase 2 loads, opens its windows then awaits your command. At this point there is a major difference in the way Release 3.2 is controlled from the budget version. Please read the appropriate section.

BUDGET Q_LIBERATOR PHASE 2

When the budget compiler is started without any parameters, it simply prompts for a command line to be entered. The command line has precisely the same syntax as earlier defined, but you should not put any quotes around the command string when it is entered. For example,

Command : MDV1_DEMO_SORT -OBJ MDV2_FREDDY_OBJ

would compile the demonstration program as before.

At the end of the compilation, you will be asked if you want to compile another program. Answering no terminates phase 2. You can also terminate Q_Liberator by entering an empty command line.

When you have several programs to compile on a microdrive based QL it may be more productive to create all the workfiles first then compile them one after the other with phase 2. Phase 2 is then only loaded once.

Normally the workfile is deleted at the end of a compilation. If however phase 2 fails because there is insufficient memory, the workfile remains intact. You should reset the QL, reload Q_Liberator form the working copy then repeat phase 2.

You should now skip to the end of this chapter where compiler options and directives are described in detail and further examples are given.

```
Q_Liberator                                          40
Budget SuperBASIC Compiler 1.0-10000 Copyright 1987 Liberation Software

Command : flp2_tst1
Compile another program? y/n:

No errors found
Size of program : 468
Data area       : 2504
Highest channel : 15
Compile time    : 3 seconds
```

THE Q_LIBERATOR MENU SYSTEM

Q_Liberator Release 3.2 is equipped with an entirely different user interface. In place of the simple command line of the budget compiler, there is now a menu driven front end which lets you easily configure the compiler to suit your own purposes. The menu system exploits, but is not dependent upon the QPTR interface as used in the QRAM utility package from QJUMP. If you own this package, preferably with a mouse, then the menu system will be entirely familiar. If not, you can drive the program from the keyboard and still have access to QRAM like facilities such as the ability to reposition the compiler window on the screen.

To see the new menu system in action, boot the Q_Liberator system from a working copy and type LIBERATE, with no parameters. After a short loading time you will see the Q_Liberator menu screen populated by a set of boxes, mostly containing function names. These are the menu items. With QRAM installed, you can select these items with the pointer in the usual manner. If you don't use QRAM then a non flashing cursor will be visible in one of the boxes. Press control C to switch the keyboard to this job (page 3.3) and the cursor will be replaced by a highlighted outline around the box in which it sat. This is the Q_Liberator cursor. It can be moved back and forth between the various menu items with the 4 cursor keys up, down, left and right. If you ever switch to another job with control C the outline cursor will again be replaced by a conventional QL cursor.

A menu item can be selected or deselected by hitting the space bar when the cursor is over it. A selected item is shown with a red background, a deselected item has a black background. In addition, the items with function names in lower case can be toggled by entering the first character.

Help can be obtained about any item by positioning the cursor over it and pressing F1. Provided that the file QLIB_HELP is present, a page of relevant information will be displayed. Please read the help information for each entry as it is designed to complement this manual. If you wish you can COPY the file QLIB_HELP to a a printer or the screen, since it is stored in a readable form.

Two other keys have a special function. F4 redraws the entire Q_Liberator screen and ESCape has the effect of QUITting the compiler.

MENU ORGANISATION

The box marked 'Compile' in the centre of the screen will start Q_Liberator compiling when it is selected providing that a source file name has been entered as described below.

In a row above 'Compile' and on either side of it the compiler options are displayed as described in chapter 4. Options can be selected or deselected as required with the spacebar.

The long horizontal boxes in the lower part of the screen are used to enter file or job names. By positioning the cursor around one and pressing space the contents of the field can be edited in the normal QL manner. An edit is terminated by ENTER or the up or down keys. Note that entering the source file name automatically defaults the object file name and the job name. They can be altered if required.

The top right corner contains the function MOVE. MOVE is used to change the position of the Q_Liberator window on the screen. When MOVE is selected a cursor appears which can be moved around the screen. When ENTER is subsequently pressed, the window is redrawn so that the top right hand corner is as close as possible to the cursor with the whole window still visible. The position which Q_Liberator last occupied is stored between calls to the compiler. See the description of QLIB_USE in chapter 14 for details.

COMPILING A PROGRAM

Before Q_Liberator can start to compile a program, it must be given the name of the source file to process. The source file must be either a Q_Liberator work file as created by the LIBERATE command or a file which has been produced by QSAVE. There is normally no need to type the extension when you enter a source file name. Q_Liberator will first append _wrk to the name you give it and try to compile a temporary work file. If this does not exist then it attempts to compile a QSAVEd file (_sav). If you want to force the compiler to compile the QSAVEd file, then enter the name in full ie including the extension _sav. The only difference in the treatment of _wrk and _sav files is that the former are automatically deleted after compilation.

Thus if we wish to compile DEMO_SORT on mdv1, we could compile it in the following different way (Type the UPPER CASE only):

LOAD MDV1_DEMO_SORT	load program to memory
LIBERATE MDV1_DEMO_SORT	phase 1, creates 'mdv1_demo_sort_wrk'
LIBERATE	phase 2, call up the menu system
	wait till the screen clears
press CONTROL C	an outline cursor appears
press SPACE	select the source file name box
MDV1_DEMO_SORT	enter the source file, the others default
press C	start compilation

COMPILER OPTIONS

Compiler options turn various compiler features off or on. One or more options, separated by spaces can be specified in a string after the filename in a LIBERATE command or in a command line. Each option consists of a short mnemonic name preceded by a minus sign. Options can be specified in upper or lower case in any order. Some require a parameter which must immediately follow the corresponding option, again separated by 1 or more spaces. The complete list of options is summarised below. Some relate to topics discussed in detail elsewhere. Further information is contained in the Release 3.2 HELP file.

-NOLINE	Suppress generation of a line number table. This makes the object program shorter, but any runtime errors will not contain a line number. If your program includes a GO TO expression (eg GO TO x*10) or other statements which require a line number to be calculated, then the compiler will always generate the line number table because the runtime system requires it.
-STAT	Print memory usage statistics at end of job. The format of the statistics is described in chapter 7.
-OBJ filename	Use filename as the name of the object file. This lets you create the object file on a different device from the work file. Note that _obj will still be appended to the filename.
-NAME jobname	Change the name of the job. This is the name used to reference the job whilst it is running. It cannot contain spaces and is best kept short.
-RUN[device]	Link a copy of the runtime system to the compiled program. The object program can then run in stand alone mode ie without the runtime system loaded. Such programs will of course be longer than programs compiled without this option. The device parameter specifies where the runtime system QLIB_RUN is to be copied from. eg mdv1_ . It MUST be present when using the BUDGET compiler and MUST NOT be present when using release 3, which always obtains the runtime system from memory.
-LIST filename	Divert the error listing to the specified device or file. This can be a printer, a disk file etc.

The defaults when no options are present are:

Line number table included, no statistics, no runtime linkage, listing to Q_Liberator window. The object name and job name are derived from the filename.

RELEASE 3.2 COMPILER OPTIONS

The following options can only be used in command lines if you have Release 3.2. It is often more convenient with this release to turn options on or off interactively via the menu system.

-NONAMES — Normally Q_Liberator includes the name of all variables which are used as procedure parameters in the object file. This is necessary to support procedures such as OPEN which can use a variable's name as a parameter. For example, OPEN #3,MDV1_DATAFILE. If this option is selected, then such names are not included, and the object file is correspondingly shorter. Statements such as the example above then have to be rewritten with the file name inside quotes, eg OPEN #3,"MDV1_DATAFILE".

-AUTOF — When this option is used, all FOR variables in the program are treated as integer variables if possible. See chapter 8 for further details.

-DEBUG — This option tells the compiler NOT to obey the compiler directive $$off described later in this chapter. If any debugging routines are included within the program, they will always be compiled when -DEBUG is used.

-WINDS — If this option is present then a compiled program will have channels #0, #1 and #2 already open when it starts. If the option is off then the program must open all its own windows.

The defaults for the above options are:

Names generated, -AUTOF on, debug mode off and -WINDS on.

COMPILER DIRECTIVES

These are special REMark statements inserted into a SuperBASIC program to instruct the compiler how to compile specific parts of a program or about special storage requirements at runtime.

A line containing a directive must start with a REMark followed by 2 dollars then the first directive.

Each directive consists of a 4 character name followed by an equal sign and then a parameter. There must not be any spaces separating these items. More than 1 directive can be placed on a line by separating each with a comma.

SETTING THE DATA AREA

The following 3 directives affect the size of the object program data area. They are only necessary when the default values are inadequate or over generous. When present, they are best placed at the start of the program, where they can be easily seen. Explanations of the parameters which are changed can be found in chapters 7 and 10.

REMark $$heap=SIZE	Set size of initial user heap allocation. Default 2048, minimum 32, maximum 512k
REMark $$stak=SIZE	Set size of the working stack. Default 800, minimum 128, maximum 512k
REMark $$chan=MAX	Define maximum channel number to be used. This reserves space for the channel table. See chapter 8 under CHANNELS for more information

LINKING ASSEMBLER EXTENSIONS

REMark $$asmb=FILENAME,INIT,TABLE

This directive causes SuperBASIC extensions written in assembler or with release 3, compiled libraries, to be linked into the object program during compilation. It may be specified up to 8 times. Each library can contain any number of procedures or functions. See chapter 9 for details of how to use this directive.

OPTIMISATION OF CONSTANTS

REMark $$i — Turn on integer mode.
This directive instructs the compiler to generate integer constants whenever possible. This will reduce the size of the object code and give increased performance when integer variables are used. Integer constants are also 4 bytes shorter than floating point constants.

REMark $$f — Turn on floating point mode.
This directive instructs the compiler to generate floating point constants, thus optimising the code for floating point work.

The default case is equivalent to $$f and is suitable for general use. Where space is at a premium, using $$i gives space savings of around 10% on average programs. When maximum speed is required, these directives can be used any number of times within a program to turn on the appropriate optimisation for specific routines.

SETTING THE INPUT BUFFER SIZE

When data is read from a device using INPUT, it is placed in a temporary buffer. This buffer has a fixed size of 128 bytes in ROM versions AH and JM. If the input data exceeds this size then a 'buffer overflow' error will occur. Page 11.4 shows how to trap such a condition with Q_Liberator.

JS and later ROMS have a dynamic buffer which expands as necessary. If you wish to compile programs which INPUT more than 128 bytes then you must use the $$buff directive described below to set the maximum buffer size required.

REMark $$buff=size — Set INPUT buffer to size specified.

$$buff gives no advantages with AH and JM ROMS.

RELEASE 3.2 COMPILER DIRECTIVES

The following directives are only available with the release 3.2 compiler.

REMark $$off — This causes the compiler to ignore all subsequent program lines until a $$on directive is encountered. Its purpose is to suppress compilation of test and debugging routines included within the program. $$off is ignored if the -DEBUG option has been used.

REMark $$on — This turns compilation back on after a $$off directive has been used.

REMark $$external — Informs the compiler that the procedure or function definition which immediately follows this directive is to be accessible from other programs. (see Chapter 14)

REMark $$ext_all — Informs the compiler that every procedure and function is external and accessible from other programs.

Chapter 5 Compiler Messages

This chapter explains all of the messages which can occur when you are compiling a program wth Q_Liberator. It concentrates on those messages which pertain to errors or inconsistencies in your program. However both phase 1 and phase 2 can encounter errors when accessing microdrives eg 'drive full' or 'file not found'. These messages are self explanatory.

MESSAGES DURING PHASE 1

Phase 1 will give the error 'bad name' if you try to use the LIBERATE procedure within a program and 'invalid Job' if there is no program to compile. You can also get 'bad parameter' if the name you have chosen for the object corresponds to one of your procedure or function names.

MESSAGES RELATING TO STRUCTURE CHECKS

A correctly written SuperBASIC procedure or function should have only one END DEFine statement. However the interpreter will tolerate and correctly handle multiple END DEFines.

eg
```
DEF PROCedure TEST (x)
IF x=1 then END DEFine
PRINT X
END DEFine
```

Q_Liberator always checks that there is only one END DEFine for each procedure or function. If a procedure or function contains multiple END DEFines then Q_Liberator changes all but the last END DEFine into a RETurn, which is the correct way to exit prematurely from a procedure.

Thus the above example would become:

```
DEF PROCedure TEST (x)
IF x=1 then RETURN
PRINT X
END DEFine
```

BUDGET Q_Liberator performs such checks on program structure during phase 1 and may issue some of the errors below. RELEASE 3.2 performs all structure checks during phase 2 and issues more explicit warnings or errors. It is also capable of compensating for a greater range of errors.

END DEFine error — Budget QLIB only

This means that you have either nested DEFinitions or an END DEFine has been found outside of a procedure. The rules concerning DEFine and END DEFine are listed in chapter 8.

END DEFine altered — Budget QLIB only

One or more conditional END IFs has been changed to a RETURN. The BUDGET compiler actually makes this change to your source program in memory during phase 1. They can be seen when the program is listed. This message is only issued once, regardless of how many RETurns had to be inserted. See chapter 8 for more details.

MESSAGES FROM PHASE 2

Phase two reports errors on the screen or other listing device as they are encountered. Q_Liberator continues to process your program after an error has been found, but will not generate any object program, since it would be unusable.

Some conditions generate warnings rather than errors. These happen because of subtle differences in the way in which the interpreter and Q_Liberator work. Q_Liberator recognises a problem and takes corrective action. In such cases an object file is generated and will often run correctly. You are advised however to examine your source program to understand the warning, then make the necessary corrections and recompile.

All warnings and errors are preceded by a line number and the statement number within the line. eg Line 100,3 is the third statement on line 100. The line number is the line at which the error was detected. This will usually be the line which needs changing. Sometimes however the real error may lie elsewhere, usually earlier, in the program.

If your program is still in memory you can examine it and correct errors while phase 2 of Q_Liberator is running.

To draw your attention to these messages, Q_Liberator gives a short high pitched beep when warnings are issued and a low pitched beep for errors.

Warning..END IF without IF

The compiler has spotted an END IF where one is not needed. It simply ignores it in the same way as the interpreter does.

Warning..END IF missing

Any IF statements within a procedure or function ought to have a corresponding END IF within the same procedure or function. The interpreter is not so fussy about this as the compiler and will quite happily use the next END IF which it finds. This will almost certainly not be what you intended. Thus if the compiler arrives at an END DEFine with one or more unterminated IFs outstanding, it will insert them automatically in the object program just before the END DEFine and give you a warning. It does not change your source program; that is your responsibility.

Warning..Procedure cannot be compiled

You have used a procedure which makes no sense in a compiled environment. Rather than forcing you to remove it, Q_Liberator simply ignores it. See chapter 8 for further explanation. The illegal procedures are:

AUTO CONTINUE DLINE EDIT LIST LOAD LRUN MERGE MRUN NEW RETRY RENUM SAVE

Warning..Variable used for channel number

This message is printed once at the end of compilation if somewhere in the program you have specified a channel number in a variable. Q_Liberator does not know how big to generate the channel table and generates the default size (0 to 15). It may be necessary to insert a $$chan directive to increase this.

The following 4 messages relate to program structure errors as described on page 8.2 and at the start of this chapter.

Warning..Conditional END DEFine , RETURN assumed	**Release 3.2 only**
Warning..Nested DEFINEs , END DEFINE inserted	**Release 3.2 only**
Warning..END DEFine missing - inserted	**Release 3.2 only**
Error....END DEFine without DEFINE	**Release 3.2 only**

Error....Not a Q_Liberator work file.

The file which phase 2 is processing is unrecognisable as the output of phase 1. Either you have been tampering with the work file or a corruption has occurred. Repeat phase 1.

Error....Unrecognised symbol

The line being processed starts with an unrecognised character. Normally such errors are trapped by the SuperBASIC editor which flags them as a MISTake. The likely cause is that the work file is corrupt. Repeat phase 1.

Error....Unsupported statement

The line contains a statement which is not supported by Q_Liberator. In practice this means SuperBASIC has recognised an error and inserted a MISTAKE, or you are trying to use the constructs in JS and later ROMS for error trapping ie WHEN ERROR etc.

Error....Too many nested IFs

Each time you use an IF statement within an IF statement the compiler needs space to keep track of this nesting. It can do this up to 32 times; beyond this it gives up with this error. If you get this error then it most probably means that your program needs restructuring. If a limit of 32 really causes you a problem it can be increased. Write to us.

Error....Too many nested SELects

This is similar to the nested IF error described above. The maximum nesting is again 32. Note that there is a separate storage area for administering SELects and IFs.

Error....SELect missing

The compiler has found a SELect clause (eg ON x=1 or simply =1) but there has been no previous SELect which must precede such a construct. Your program must be corrected as the interpreter's behaviour in such circumstances is to go searching through your program for the next END SELect (any one will do!) then continue execution after this point. An END SELect without a prior SELect will also give this error.

Error....ELSE without IF

An ELSE statement has been found outside of an IF construct. When the interpreter encounters this it searches down your program until it finds the next END IF then continues execution at that point. This is a good source of bugs. If no END IF is found the interpreter just stops. This is an example of how using Q_Liberator can help to track down problems in your program.

Error....END SELect missing

In a correctly structured SuperBASIC program, every SELect must have a corresponding END SELect. Furthermore they should both be contained within the same procedure. If the compiler finds itself at an END DEFine with an unfinished SELect then it issues this error.

Error....END REPeat missing

Each REPeat started within a function or procedure should be terminated with an END REPeat within the same procedure. If this rule is violated then this error will be given at the end of the procedure.

Error....Ambiguous name

A name has been used to represent more than one entity. eg as a variable and as a procedure or function. You will also get this error if you try to make an assignment to a function. Programs containing such errors will usually be rejected by the interpreter with a 'bad name' error.

Error....Too many assembler routines

A maximum of 8 assembler extensions can be linked to an object module using the directive $$asmb.

Error....Cannot open assembler routine

The assembler extension cannot be found on the device which you stated in the directive $$asmb.

Error....Keyword must be at start of program Release 3.2 only

GLOBAL, DEF_INTEGER, EXT_PROC and EXT_FN (in any order) must be placed before the first line of your program (excluding REMarks).

Error....Syntax error (in REM $$ directive)

Directives are explained in chapter 4.

Chapter 6 Runtime Errors

When an error occurs within a running object program it is termed a runtime error.

You will already be familiar with many of the runtime errors because they are identical to those generated by the interpreter. However, the interpreter is often vague about the exact cause of an error with some of the messages being used to cover more than one situation. Q_Liberator improves upon this with more explicit messages. Furthermore the chance to recover from certain errors is included as a standard feature.

THE ERROR WINDOW

When a runtime error occurs, Q_Liberator opens a 3 line error window in which to display it. This window stays on the screen until you select the cursor within it by pressing control C. Now you must acknowledge the error with any key or if prompted, answer the Retry question. The error window will then disappear. Note that when memory permits, this window is transient, ie when it is closed it restores what was present on the screen at the time it was opened.

The name of the job that caused the error is always printed in the top left hand corner of the error window. The rest of the error information depends upon the category of runtime error. There are three categories, initialisation errors, QDOS errors and QLIB errors.

INITIALISATION ERRORS

Initialisation errors occur immediately after an object program is loaded if something essential to support the program cannot be found.

The first thing a job looks for is the runtime system. If this is not found or has the wrong version then you will see

Runtimes missing!

on channel 0. The error window cannot be used because it is controlled by the runtime system! The runtime system, QLIB_RUN should be loaded by a boot program prior to running an object program, or the program should have its own copy linked to it. The release 3.2 runtime system will support programs compiled by the budget compiler but NOT vice versa.

The second thing that a Q_Liberator object program does is check that any extension procedures or functions which it needs are present. The action taken if any are not found differs between the compiler releases. The BUDGET compiler produces a list of the missing names in the error window. The program can go no further and aborts.

For example if you had a game which required 2 assembler procedures and you forgot to load them with a boot program, and had not linked them during compile time, you might see:

JOB : Spacegame ZAP EXPLODE missing!

with the BUDGET compiler.

RELEASE 3.2 does NOT report missing procedures at initialisation - they aae assumed to be overlays which will later be loaded and are marked as currently undefined. If they are called when they are undefined a QDOS 'bad name' error is reported. Chapter 14 explains overlays in detail.

QDOS ERRORS

QDOS errors are the standard error messages which are also used by the interpreter. They are listed in the Concepts section of the QL User Guide, and will be familiar to most users. There is a procedure described in chapter 11 which contains every QDOS error.

Only some of the QDOS errors are actually used. The only occasion which can result in a QDOS error is when a machine code procedure or function returns an error code. The only exception is 'bad name' as described above for Release 3. Q_Liberator has its own messages for other circumstances. For example trying to position the cursor outside of a window results in a QDOS 'out of range' error. Trying to access an array element which does not exist gives a QLIB 'Index out of range' error.

QDOS errors are reported along with the line number at which they occurred providing that you have not suppressed generation of the line number table by a compiler option. Following the line number the name of the offending procedure is printed, then the text of the QDOS message.

QDOS errors are usually input/output errors, ie they occur in procedures which move data to and from devices. Often such errors will be recoverable. For this reason Q_Liberator always lets you retry when a QDOS error occurs. The point at which the retry restarts is immediately before the procedure name which caused the error. For example the program TEST might contain the following:

```
10 CLS: OPEN_IN #3,MDV2_TESTDATA
```

If you ran this with the wrong tape in mdv2 then you would see the following in the error window:

```
Job 1 TEST   Line 10   OPEN_IN
not found
Retry Y/N
```

Note that this means the procedure OPEN_IN has reported error 'not found'. It does NOT mean that OPEN_IN itself cannot be found.

Placing the correct tape in mdv2 and answering 'Y' to the retry question would result in the program restarting just after the CLS procedure and continuing successfully.

If you answer 'N' then the program will print its runtime statistics and abort.

Note that RETRY may not be possible under some circumstances, and of no value in others, but at least you can try. In addition to this standard form of error recovery, QDOS errors can be trapped using Q_ERR error trapping, explained in chapter 11.

QLIB ERRORS

QLIB errors are more serious. They indicate either a programming problem or a lack of memory. Whereas the error messages generated by the interpreter are often ill defined and unhelpful, Q_Liberator has many explicit runtime messages to shed light on where an error really lies. Some of these are related solely to Q_Liberator's internal workings while others are used to replace an ambiguous QDOS message.

Each QLIB message has 1 or more error numbers associated with it which can sometimes convey additional information. Where this is the case, details are given after the explanation of the message in the complete list below. QLIB errors are always fatal; no retry is possible.

No heap space

The job has requested more data storage from the common heap but has been unsuccessful. There are too many jobs running, the heap is fragmented or you have exceeded the memory capacity of your QL. Possibly you have written a program which runs riot and grabs more and more memory. Read the section on memory organisation for further details.

No stack left

There is insufficient stack space to continue. Allocate more stack using QLIB_PATCH or include a **$$stak** directive in the source program and recompile.

6 Occurred within runtime system
12 Occurred within procedure

Variable undefined

You have referenced a variable which has not been assigned a value. SuperBASIC would give 'error in expression'.

String too long

The maximum string size is 32767 characters. This error is generated when concatenating two strings (eg a$&"ABC") produces a string which exceeds this limit.

Array too big

The dimensions of the array when multiplied together are too large. See the section on arrays for further details.

Array not DIMed

You have tried to access an array which is currently undefined. Place the DIM statement before this point in the program.

Indices wrong

You have specified too many or too few indices for an array or string, or the 'array' is actually a variable.

19 Type is wrong
20 Number of indices wrong
27 Occurred in a procedure parameter

Index out of range

An index is negative or greater than the dimension.

23 Occurred during a slicing operation
28 Occurred in a procedure parameter
35 Occurred during array or string access

Slice not allowed

You have attempted to perform a slicing operation on the wrong sort of data. This can happen if you pass a simple variable to a procedure which expects to work with arrays, or if you don't specify enough indices to uniquely identify an element of an array. Note that only slices of strings or string arrays can be used within an expression, but any array slice can be used as a procedure or function parameter.

7 Occurred within an expression
24 Occurred when storing data into a variable eg a(2 to 4)=1
26 Occurred in a procedure parameter

Array not allowed

You have attempted to use an array when a simple variable was expected. This happens when an array is passed to a procedure or function which can only deal with simple variables.

Division by 0

This is of course illegal in both floating point and integer form.

25 Integer operation
37 Floating point operation

Overflow

If floating point overflow, you have exceeded the range of QL floating point arithmetic or more likely, divided by zero. If integer overflow then an integer has exceeded the range -32768 to +32767. This can only happen when making an assignment to an integer variable. When evaluating an integer expression, Q_Liberator will automatically switch to floating point if integer overflow occurs.

13 Integer overflow
36 Floating point overflow

String is not numeric

You have tried to perform a calculation on, or set a variable to, a string which does not contain a valid number.

Cannot retry

The error is too severe for retry to work. This is unlikely to occur in practice.

Unresolved reference

Your program is trying to go to an undefined place. This may be caused by EXITing from a FOR loop which has no END FOR.

RETurn missing in function

Every function should RETurn a value. This error occurs if the program reaches the END DEFine of a function.

Out of DATA in READ

The READ procedure has run out of DATA statements. Use EOF to test for this condition prior to calling READ.

GO TO out of range

You are attempting to GO TO a line number beyond the last line in the program.

Internal

Oh dear, you should never see this! An error has occurred inside Q_Liberator. If it really happens to you, check that its not a spurious corruption, that you are not violating any rules and that your program works correctly under the interpreter. If the error persists, please write to us including if possible a concise demonstration of the error.

The following errors can only occur with Release 3.

FORtype error

The wrong type of variable has been used in a FOR loop. The compiler must allocate extra storage for FOR variables. This error happens if the FOR control variable is a formal procedure parameter and the actual parameter is a normal float or integer.

```
eg.  10 DEFine PROCedure BADFOR(X)
     20   FOR X=1 to 10: PRINT X
     30 END DEFine
     40 A=1
     50 BADFOR A
```

A 'For type error' is reported at line 20 because variable A is substituted for X in the procedure. The interpreter would give a 'bad name' error under these circumstances.

Overlay table full

You have attempted to use too many external files simultaneously either as overlays or resident procedures in one program.

Global missing

A GLOBAL variable referenced in an external is not present in the root.

Chapter 7 Memory Management

SuperBASIC is privileged among QL jobs in that it is the only one which is allowed to shrink and expand its entire area to suit its needs. EXECutable jobs such as Q_Liberator object programs have to make do with a fixed job area allocated when they are started. If they require more storage then they must use the *common heap*.

The common heap is an area of memory which QDOS administers. When a job asks for some memory, QDOS splits off an area of the common heap for the job to use. When a job is removed, any heap it has borrowed is returned to QDOS. If many jobs are running each using common heap, a problem called heap fragmentation can occur. This is when the heap is split into many small parts none of which are big enough for a given job to use.

Q_Liberator is flexible about memory organisation. Object programs can be tailored to confine all data within a job's boundaries, or they can expand into the common heap as required. The choice can be made before a program is compiled by using a compiler directive or after compilation using the utility program QLIB_PATCH.

OBJECT PROGRAM STRUCTURE

A Q_Liberator object program consists of a code area and a data area.

The code area contains the compiled form of the program and parameters associated with it. If you have linked in the runtime system or any assembler routines, then they too are contained within the code area.

The data area contains various control areas described below and storage for variables. Note that it is only the code area that occupies file space on a microdrive. However when the program is loaded into memory, there must be enough space to accomodate both the code and the data.

The sizes of the code and data areas are printed at the end of a successful compilation. They can also be obtained by using QLIB_PATCH, whilst the total area occupied by a job in memory can be displayed by the procedure QJ.

DATA AREA

The following parts of the data area are of interest to the user because their size can be modified:

Channel table

The size of this table dictates the highest channel number that can be used within a program. It is sensible to keep channel numbers low because a 40 byte entry is reserved for all channels up to the highest which you specify. (This is also true for the interpreter).

The number of channels is normally set automatically by the compiler. The minimum number of channels is 3 and the default if variables are used for channel numbers is 16. This can be changed by using the $$chan directive. Note that an attempt to open a channel with a number higher than the table size allows will probably result in a system crash.

Stack

The stack area is a general work area used to store return addresses, local variables, procedure parameters and miscellaneous control information. The amount of stack used depends very much on individual programs. Deeply nested procedure calls or recursive routines will require a large stack to run successfully, as will machine code routines which manipulate large strings. If a program runs out of stack then it will normally stop with a Q_LIB error.

Occasionally the stack shortage occurs within a machine code procedure which cannot handle the condition. This is likely to cause a crash.

The default size for the stack is 512 bytes which is generous for small programs. It can be changed by placing a $$stak directive in the source program.

Heap area

The heap area is the section of the job's data area used for the storage of dynamic data types (ie those that grow or shrink in size during runtime). All strings and arrays and any associated descriptors are stored here. When program is compiled, Q_Liberator cannot tell how large these items might become and so it simply reserves space in the data area to be administered at runtime. If this space proves to be too small then an object program will automatically request one or more areas from the common heap and expand into them. Thus a program will never crash because of a heap shortage until the whole of the common heap is exhausted.

The default size of the heap area is 512 bytes. It can be increased up to a maximum size of 512k. When extra storage is requested from the common heap, it is allocated in 512 byte chunks at a minimum. To avoid any possibility of common heap fragmentation you should obtain the statistics for a given job and set the heap area high enough so that no common heap requests are necessary.

RUNTIME STATISTICS

Most programs will run correctly with the default parameter settings, but they will not be making the optimum use of memory. To assist in setting the stack size and data size parameters, the runtime system can produce statistics. These are produced when a job ends if the -stat option was selected during compilation or subsequently turned on by using QLIB_PATCH. The statistics are always produced when a job terminates with an error.

The statistics appear in the error window in the form:

Data aaaa bbbb cc Stack dddd eeee

where aaaa gives the size of the heap area within the job, as set by the $$heap directive.

bbbb gives the total number of bytes requested from the common heap.

cc is the total number of common heap requests.

dddd is the size allocated to the stack as set by the $$stak directive.

eeee is the amount of stack which was actually used.

If you compile a program which uses strings or arrays using the standard defaults, then the first time that it is run you will see that bbbb and cc are non zero ie the job has 'spilled over' into the common heap. By setting the heap size to a figure slightly greater than the sum of aaaa and bbbb the entire user heap can be confined to the job's data area.

Similarly the stack area can be reduced by setting the stack size to a figure closer to eeee. It is wise to always leave some spare.

QLIB_PATCH

The program QLIB_PATCH, supplied in object form on your working copy, can be used to change parameters after a program has been compiled. It may be used interactively by loading it with the command: QX QLIB_PATCH The presentation of QLIB_PATCH on the screen varies between the budget version and release 3.2.

BUDGET QLIB PATCH

With this version you will first be asked for the object name which you want to change (no need to append _obj). The current parameters are displayed and you can overwrite them if necessary. If you decide not to patch the file you can QUIT before the changes are applied.

Budget QLIB_PATCH can also be started by passing it a command string in a similar format to the LIBERATE command. The first parameter in the command string is the name of the file to be patched. It should be followed by a list of options separated by spaces or commas. All options expect a parameter except for -stat and -nostat. The options are as follows:

-chan number	change the size of the channel table
-stak number	change the size of the stack area
-heap number	change the size of the job's user heap area
-name jobname	change the name of the job (NOT the object name)
-stat	turn on statistics
-nostat	turn off statistics

Example

```
QX mdv1_qlib_patch,"mdv1_demo_sort -stak 400 -chan 4 -stat"
```

If a parameter is out of range then QLIB_PATCH enters the interactive mode to allow the error to be corrected. If the patch is successful the message "QLIB_PATCH complete" is printed on channel 0.

RELEASE 3.2 QLIB_PATCH

This version is controlled by the pointer interface in the same way as the compiler itself. New values for the parameters can be typed into the appropriate box. The changes are applied when PATCH is selected. There is no command line interface, but the utility can be made resident or overlaid and called up with the procedure name PATCH.

PATCHING QLIB_OBJ

In the unlikely event of Q_Liberator itself running out of stack when compiling, it is possible to increase its stack with QLIB_PATCH. The other option parameters should not be modified.

Chapter 8 Interpreter/Q_Liberator Comparison

Q_Liberator was designed to provide maximum compatibility with the SuperBASIC interpreter. There are however areas where a compiler must by its nature do things differently from an interpreter. Furthermore there are SuperBASIC keywords which are meaningless in a compiled environment.

This chapter compares the operation of Q_Liberator with the interpreter and documents deviations, enhancements and restrictions. A number of rules are formulated which if applied will help to ensure that your programs compile without errors. These rules should not be regarded as restrictions; they are all really part of the syntax of SuperBASIC and are therefore built into Q_Liberator. The interpreter is less rigourous in its interpretation (of the rules) and can be made to disregard them by bad programming.

COMPATIBILITY

Q_Liberator was designed to support the version of SuperBASIC present in JM and AH ROMS as documented in reference 1. The additional keywords present in JS and subsequent ROMS are not supported as they are incomplete and not formally documented. The Q_ERR form of error trapping is adequate compensation for their omission and has the advantage of being useable with all ROMS.

Compiled programs are fully portable across different ROM types.

Compatibility means that a Q_Liberator object program should behave identically to the corresponding SuperBASIC program running under the interpreter. This is generally true providing that the first rule is met:

Rule 1: The source program must run correctly under the interpreter.

Compiling programs which conform to the SuperBASIC syntax but give rise to serious runtime errors can produce unpredictable results; there is no guarantee of identical behaviour in such cases.

Sometimes, however, it can be enlightening to compile a program which is behaving strangely, because Q_Liberator's more explicit error messages may pin down the problem either at compile time or runtime.

PROGRAM STRUCTURE

SuperBASIC, in contrast to earlier BASIC implementations is well equipped with constructs which add structure to a program. PROCedures, FuNctions, REPeat loops, FOR loops etc, simplify a program and make it easier to read. Programs can be well structured or badly structured. We shall not attempt to formally define 'well structured' but will simply state that a well structured program would already obey all the rules presented here, would probably be indented to reveal the underlying form and would compile without problems.

Badly structured programs will result in compilation errors and warnings, and may well be impossible to fathom.

During compilation, Q_Liberator has to ascertain the structure of an entire program as it reads it from top to bottom. The interpreter however tackles a program's structure as it encounters the keywords at runtime. It is quite possible to exploit this phenomenon to produce ill-structured programs which will nevertheless run. As an extreme example consider

```
10 bad_practice: STOP
20 END DEFine bad_practice
30 DEFine PROCedure bad_practice
40    PRINT "breaking the rules"
50    GO TO 20
```

The interpreter does not care that the procedure seems to end before it starts. At runtime it sees a DEFine then an END DEFine which is all it requires. Of course Q_Liberator cannot predict the order in which statements will be executed and so would reject the above program during phase 1.

DEFine...END DEFine

The rules relating to procedure definitions are simple:

Rule 2: Every DEFine statement must have a corresponding END DEFine later in the program.

Rule 3: DEFinitions cannot be nested inside each other.

It is also a bad habit to have more than one END DEFine in a procedure or function. Some programmers use this as a method of escaping prematurely from a routine. Q_Liberator tolerates this by changing such END DEFines into RETurns. This is performed during phase 1 with the budget compiler and in phase 2 with Release 3.2.

FOR...END FOR

The SuperBASIC FOR...NEXT...END FOR construct is a vast improvement over the FOR...NEXT loop present in earlier BASIC implementations. However some of the books purporting to teach SuperBASIC fail to make clear exactly how it operates and how it should be used.

With the exception of the single line (inline) form, each FOR statement ought to have a corresponding END FOR statement. This is the point at which the loop ends.

If you wish to prematurely process the next item whilst within a FOR loop, the NEXT statement should be used. This passes control back to the line containing FOR.

If you wish to prematurely escape from the entire loop, then the EXIT statement should be used. The program jumps to the statement after the END FOR.

For example,

```
10 FOR x=1 to 10,20,30,40
20   IF x=a THEN NEXT x        : REMark skip print if x=a
30   IF x=b THEN EXIT x        : REMark abort loop if x=b
40   PRINT x
50 END FOR x
```

In practice, often due to experience of earlier BASICs, programmers will use NEXT in place of END FOR. Q_Liberator supports this usage and such programs will compile without errors. They will also run without problems with two exceptions:

If an EXIT is attempted, the QLIB error "Unresolved reference" will be reported.

An empty FOR loop (eg FOR x=2 to 1) will cause the same error because the program expects to continue after an END FOR, and none is present.

In these cases, the interpreter would simply stop or, worse still, use the next matching END FOR which it could find.

The inline form of a FOR NEXT loop has an implied END FOR at the end of the line. If a superfluous END FOR (or NEXT) is present, it is simply ignored.

```
ie   10 FOR x=1 TO 10: PRINT x
and  10 FOR x=1 TO 10: PRINT x: END FOR x
```

are equivalent.

The control variable of a FOR..END FOR loop cannot be a formal procedure parameter or an error will occur.

FOR loops can be nested to any desired depth; there is no stack penalty. FOR loops ought not to be nested as shown below, but Q_Liberator will in fact handle such nesting in precisely the same manner as the interpreter.

```
10 FOR x=1 TO 10
20 FOR y=1 TO 10
30 PRINT x,y
40 END FOR x
50 END FOR y
```

With Release 3.2, integer FOR loops are possible. This is described in chapter 14.

REPeat...END REPeat

This construct has no counterpart in earlier BASICs. Consequently there is no excuse for not obeying the rules.

Rule 4: Every REPeat should have a corresponding END REPeat later in the program.

Rule 5: REPeat loops started within a procedure or function must be terminated inside that procedure or function.

The use of NEXT as a substitute for END REPeat is not supported because such a loop cannot be EXITed. (EXIT causes a jump to the statement after END REPeat). Q_Liberator will generate the error 'END REPeat missing' with the line number of the END_DEFine statement where it was detected. There is of course no restriction on the use of NEXT within the body of the loop.

A superfluous END_REPeat at the end of an inline REPeat is ignored.

REPeats can be nested to any desired depth.

SELect ON...END SELect

It is regretable that the interpreter only permits floating point numbers as the variable which is tested in a SELect construct. It is in fact possible to enter and run a program containing a SELect on a string or integer, but it will not give correct results with the interpreter. It will, however, run correctly when compiled. This is a construct well worth using if you can put up with the inconvenience of not being able to test it with the interpreter.

```
eg  10 SELect on a$
    20    ON a$="STOP": print "stopped"
    30 END SELect
```

IF..THEN..END IF

With the exception of the inline form, each IF should have a corresponding END IF within the same procedure or function. Missing END IFs detected at the end of a routine will automatically be inserted immediately prior to the END DEFine, and a warning will be issued. You are strongly advised to check that this is the correct place for the END IF.

Superfluous END IFs are always ignored.

THE DREADED GO TO

GO TO in all its forms is fully supported. If you use a computed GO TO and end up beyond the last line of a program then you will get an error. Use of computed GO TOs requires that a table of SuperBASIC line numbers is included in the object program. This is also true for GO SUB expression and RESTORE expression.

You should never use GO TO to jump into or out of a procedure or function. This can cause problems for both interpreted and compiled programs.

PROGRAM SIZE

There is no restriction on source program size other than the memory size of your QL. For all but the shortest program the object produced will be smaller than the source. This is particularly noticeable on very large programs where the savings can approach 50% when the option to suppress line numbers is used.

The workfile is typically slightly larger than the source program. It is important to ensure that there is enough space on microdrive or disk for both the workfile and the object file before starting compilation. A useful rule of thumb is that an area approximately twice the size of the source program should be available. When space is at a premium, it is possible to place the workfile file on one device and produce the object on another by using the compiler option -OBJ. The fastest results will be obtained when a RAM disk is used.

UNSUPPORTED KEYWORDS

If any name from the following list is used within a program then Q_Liberator will ignore the entire statement, issue a warning and continue compilation.

AUTO, DLINE, EDIT, RENUM and LIST	because they are of use only during program development with the interpreter.
CONTINUE and RETRY	which are designed for interactive use. They can be replaced by Q_ERR error trapping.

LOAD, LRUN, MERGE, MRUN, NEW and SAVE	because they relate only to the source form of a program. They are replaced in part by QX and QW which load and run object programs.

Note that other procedures concerned with program development contained within some toolkits will also be unsuitable for compilation.

DATA TYPES

Q_Liberator always stores and manipulates data in a manner compatible, though not necessary identical to SuperBASIC. This is necessary to provide maximum compatibility for additional assembler procedures. In general the storage requirements of an object program at runtime will be less than that used by the corresponding source program, due to more efficient packing of numeric variables.

FLOATING POINT NUMBERS

Floating point numbers (floats) occupy 6 bytes. The range supported is identical to that of the interpreter. Arithmetic operations on floats are fully compatible with those performed by the interpreter, but are often faster.

INTEGERS

Integers occupy 2 bytes. The interpreter provides very little support for the use of integers. Simple integer variables occupy as much space as floats (8 bytes, the minimum storage allocation) and, with the exception of DIV and MOD operations, the interpreter always converts integers to floating point before performing any calculations. This conversion makes working with integers actually slower than working with floating point.

Integer variables are normally identified by the % at the end of their name, but with release 3.2, other names can be treated as integers by using DEF_INTEGER as described in chapter 14.

When presented with 2 integer quantities Q_Liberator will use 16 bit twos complement integer arithmetic for the arithmetic operations +, -, *, DIV and MOD. Note that division, / , always produces a floating point result. Such arithmetic is much faster than floating point arithmetic.

Integers should be used wherever possible to achieve maximum execution speed. The $$i directive described in chapter 4 will ensure integer constants are generated. Making all array indices integers is particularly beneficial.

If integer overflow occurs when evaluating an integer expression, then both integers are converted into floats and the calculation is repeated, this time giving a floating point result. Integer overflow errors can only occur when attempting to store an out of range number in an integer variable.

STRINGS

Strings are stored within the user heap (see memory organisation). They have the same format as in SuperBASIC ie one word length followed by the string characters. Q_Liberator supports both strings and string arrays of one or more dimensions. The subtle differences in the way in which the interpreter handles strings from one dimensional string arrays is reproduced precisely.

If a program manipulates large strings then a stack area larger than the longest string is needed for some machine code procedures to run properly. Furthermore the job's heap area also needs to be large (use statistics to see how large). For some applications, DIMensioning all strings will reduce the memory requirement. The strings then become one dimensional string arrays and always occupy the same area in memory.

Many string operations are actually performed by manipulating pointers to strings rather than the actual strings. This increases speed, but leads to a very minor restriction. If a string variable is used two or more times within an expression and its value changes between these occurences, then Q_Liberator will use the latest value throughout the expression, leading to a false result. This is best illustrated by an example:

```
10 a$="old"
20 print a$&test(a$)
25 :
30 DEFine FuNction test (s$)
40 s$="new"
50 END DEFine
```

Under SuperBASIC line 20 prints "oldnew", whilst Q_Liberator prints "newnew" because a$ is changed within the function test. Note that

```
20 PRINT a$;test(a$)
```

correctly prints "oldnew". Here the a$ and test(a$) do not occur within the same expression.

In practice, this problem will rarely, if ever be encountered.

ARRAYS

All of SuperBASIC's powerful array handling features are fully supported. Thus slices can be made of arrays to produce sub-arrays, and arrays or sub-arrays can be passed as parameters to procedures.

Arrays can be DIMensioned dynamically at runtime. eg DIM a(x,y). ReDIMensioning an array is a fast way of clearing all elements to zero.

The maximum size of an array in both SuperBASIC and Q_Liberator is determined by three things:

a) The memory available
b) The restriction that an index can have a maximum value of 32767
c) The SuperBASIC array descriptor, which limits the multiplier for a given dimension to an unsigned word.

To determine if a numeric array satisfies (c), write down the dimensions of the array then add 1 to each dimension (to allow for the zeroth element). Now starting from the second dimension, multiply all remaining dimensions together. The result must be less than 65535 for the array to be viable. The calculation is similar for a string array, but the final dimension should first be increased by 2 then rounded up to an even number.

For example, on an expanded system

```
10 DIM a%(2,4,13106)
```

is acceptable because (4+1) * (13106+1) = 65535.

```
10 DIM a%(2,4,13107)
```

causes an error because (4+1) * (13107+1) > 65535.

Since the first dimension plays no part in this calculation, making it the largest dimension can eliminate such problems.

Thus `10 DIM a%(13107,4,2)`

is entirely acceptable.

The total storage required for an array can be calculated by taking the result from the calculation above and multiplying it by the first dimension (incremented by 1), then by the size of the array element. This will be 6 for a float array, 2 for an integer array and 1 for a string array.

CHANNELS

SuperBASIC will quite happily let you open channels with numbers such as #50, but this is in fact a very wasteful practice. The channel table contains a 40 byte entry for each channel from 0 up to the highest used. You can obviously save memory by keeping your channel numbers low.

At compile time, Q_Liberator allocates a similar channel table large enough to accomodate the maximum channel number used. This can only be established when all channel numbers are literals. If any channel number is a variable, then Q_Liberator issues a warning and allocates either a default table which supports channel numbers from 0 to 15, or a larger table if the highest literal channel number exceeds 15. You can override the default by using a $$chan directive as described in chapter 4.

The minimum size of a channel table is 3 entries, for channels 0,1 and 2.

Note that attempting to access a channel number higher than the table accomodates will probably result in a total system crash.

INITIAL WINDOWS

When a SuperBASIC program starts, channels 0, 1 and 2 are usually open. The size and location of the associated windows, the paper and ink colours etc, are as left behind by the last program. It is therefore wise to always redefine these windows in the program.

When a Q_Liberator object program starts to run, the windows for channels 0, 1 and 2 are identical to the default windows present after the system is reset. If the screen is in 8 colour mode then the windows correspond to those set up when F2 is pressed; 4 colour mode corresponds to F1.

The initial windows are overridden if a channel is replaced by a channel passed to the job. This subject is discussed in chapter 10.

With release 3.2 the -WINDS option must be enabled for channels 0,1 and 2 to be opened. If this option is off then the program must open all its own windows.

Take care when calling compiled externals from the interpreter not to close channel 0 as this will prevent you from entering any further commands.

Chapter 9 Using Assembler Extensions

One of the major advantages of SuperBASIC is its extensibility. New procedures and functions can be written in assembler and linked to SuperBASIC, whereupon they behave as if they were an integral part of the language. Many such extensions are in existence. Some are designed to be of general use, such as the error trapping facilities supplied with this package, whilst others are specific to a given application.

The exceptional compatibility of Q_Liberator means that the vast majority of extensions will operate correctly in compiled programs. This includes those that access interpreter data structures like the name table, or alter variable values using the utility routine BP_LET. Any which try to manipulate the internal form of the program will of course be doomed to failure.

The rule when using assembler extensions is that they MUST be resident at the time your program is compiled, and they MUST be present in some form when the object is run. Q_Liberator will give you a runtime error if this is not the case.

LOADING ASSEMBLER EXTENSIONS

Normally a program which uses extensions will be started by a BOOT program of the form:

```
10 base=RESPR(size)                        : REMark reserve space
20 LBYTES mdv1_extensions_code,base : REMark load the file
30 CALL base                               : REMark add the new names
40 LRUN mdv1_mainprogram                   : REMark load the main program
                                             (MERGE might also be used)
```

The BOOT program is separate from the main program so that all the new procedure names are recognised before the main program is loaded. Such boot programs CANNOT BE COMPILED by Q_Liberator for the following reasons:

a) The standard function RESPR gives an error if any jobs are running and so has been modified (see below).

b) Each file of extensions contains a small piece of code to link the new names into SuperBASIC's name table which is designed to grow as necessary. The Q_Liberator name table is of a fixed size, determined during compilation.

c) LRUN is an illegal procedure as far as Q_Liberator is concerned. (see chapter 8)

This is not a serious restriction since BOOT programs are usually short and are only executed at the start of a session. By simply changing line 40 in the program above, it can be used to load extensions prior to running a Q_Liberator object:

```
40  EXEC mdv1_mainprogram_obj
```

TREATMENT OF RESPR

The function RESPR is designed to create a permanent space at the top of memory for new resident procedures. It is, however, often used in programs to reserve memory for other purposes. This is a practice which should be avoided if possible, because each time such a program is run, more and more memory is reserved. RESPR cannot do its work if any job is running and so is given special treatment by the compiler.

In an object program RESPR will allocate an area of common heap. This area is owned by the object program and will be released when the job ends.

For applications which need a permanent storage area to run correctly, memory can be allocated using RESPR in a SuperBASIC program. The address can then be passed to the job in a command string (see chapter 10).

WRITING ASSEMBLER EXTENSIONS

The rules for this are of course the same as those for SuperBASIC. Be careful, however, not to hard code any values into your code which relate only to the interpreter and remember that the job number will not be 0. Channel identifiers should always be taken from the job's channel table, accessible relative to A6.

A procedure can tell if it is running in a compiled form by testing the long word BV_TGBAS(A6). This will be 0 for a compiled program and non zero when interpreted.

BV_CHRIX can be used to reserve space on the arithmetic stack, but the stack will never actually be expanded. If there is insufficient memory then a runtime error will occur. The current stack pointer is in BV_RIP(A6), the lower limit is stored in BV_RIBAS(A6).

Functions which work with large strings will require a correspondingly large stack area.

Unlike the interpreter, Q_Liberator permits addresses passed relative to A6 to be converted to absolute addresses. Some routines can be speeded up considerably when they are not restricted to the doubly indexed addressing mode. Note that A6 itself must never be changed.

LINKING ASSEMBLER ROUTINES DURING COMPILATION

The compiler directive $$asmb can be used to permanenty link SuperBASIC extensions into an object program. This removes the need to use a boot program and gives the benefit of not filling the interpreter's name table with names which it does not need. To use this feature you need to understand the structure of such extensions and should preferably have access to the source.

The directive $$asmb may reference up to 8 modules containing extensions. Each module can contain any number of procedures or functions.

The format of a $$asmb directive is:

```
REMark $$asmb=FILENAME,INIT,TABLE
```

where

FILENAME is the full name of the module eg MDV1_EXTENSIONS_CODE

INIT is the address in module of any initialisation routine.
If present it must end with RTS and MUST NOT contain a call to BP_INIT.
If there is no routine let INIT=0.

TABLE is the address of the SuperBASIC procedure / function table as used by the ROM routine BP_INIT.

IT IS ESSENTIAL THAT SUCH EXTENSIONS ARE ALREADY LOADED WHEN THE PROGRAM IS COMPILED. If this is not observed, the compiler will find ambiguous names or unpredictable runtime behaviour will result. will result.

This condition is relaxed with release 3.2 which allows missing procedures to be declared with the keywords EXT_PROC and EXT_FN (see chapter 14). Externals compiled with release 3.2 can also be linked using this directive. In this case the INIT and TABLE parameters should be omitted.

The extensions in QLIB_EXT can be linked to your programs with the following directive:

```
REMark $$asmb=mdv1_qlib_ext,0,10
```

The following page contains an example of the use of $$asmb.

As an example of procedure linkage TO Q_Liberator, here is a shortened form of the file QLIB_EXT. The directive in the source program would be:

```
      REMark $$asmb mdv1_qlib_ext,0,12


000   43FA000A      start    lea.l  table,a1          standard procedure
004   34780110               move.w bp_init,a2        linkage (not used
008   4E92                   jsr    (a2)              by Q_Liberator)

                     **** Possible additional initialisation routine
                     **** 2nd parameter for $$ASMB ***

00A   4E75          INIT
                             rts                      must end with rts

                     **** Procedure and function table
                     **** 3rd parameter for $$ASMB

00C   0002          TABLE    dc.w   2                 2 procedures
00E   0028                   dc.w   curson-*
010   08515F435552            dc.b   8,'Q_CURSON',0
01A   0014                   dc.w   cursoff-*
01C   09515F435552            dc.b   9,'Q_CURSOFF'
028   000000000000            dc.w   0,0,0

02E   6112          CURSOFF  bsr.s  channel           Q_CURSOFF
030   660E                   bne.s  curs2             procedure
032   700F                   moveq  #sd_curs,d0
034   6006                   bra.s  curs1

036   610A          CURSON   bsr.s  channel           Q_CURSON
038   6606                   bne.s  curs2             procedure
03A   700E                   moveq  #sd_cure,d0
03C   76FF          curs1    moveq  #-1,d3
03E   4E43                   trap   #3
040   4E75          curs2    rts

042   34780112      CHANNEL  move.w ca_gtint,a2       subroutine to
046   4E92                   jsr    (a2)              return channel id
048   6618                   bne.s  chanl             in a0
04A   70F1                   moveq  #-15,d0
04C   5303                   subq.b #1,d3
04E   6612                   bne.s  chanl
050   7028                   moveq  #40,d0
052   C0F69800               mulu.w 0(a6,a1.l),d0
056   206E0030               move.l bv_chbas(a6),a0
05A   D0C0                   add.w  d0,a0
05C   20768800               move.l 0(a6,a0.l),a0
060   7000                   moveq  #0,d0
062   4E75          chanl    rts
```

Chapter 10 Inter-Job Communication

Q_Liberator object programs, like other independent programs can be loaded and started using the procedure EXEC. You have to specify the full name of the object program.

eg EXEC MDV1_DEMO_SORT_OBJ

When you type this as a direct command the sort program starts to run, but you will still be able to use SuperBASIC, ie they run concurrently. Sometimes it is more useful to suspend SuperBASIC when the object program is running, particularly to avoid conflicts over the use of the keyboard. EXEC_W will do this automatically.

eg EXEC_W MDV1_DEMO_SORT_OBJ

Now while the sort program is running, it is not possible to use SuperBASIC. Be careful to provide a "way out" of programs started using EXEC_W, or you will have to reset the machine to stop them.

EXEC and EXEC_W can also be used within compiled programs to start other jobs running. For the following discussion we shall refer to the job which contains the EXEC as the *parent* job and the job which it starts as its *daughter*. Within the limits of QDOS, any job can spawn as many daughters as it pleases. A job also has an *owner* associated with it, which may or may not be the same job as the parent.

Jobs only survive for as long as their owner exists. If the owner is removed or comes to a natural end, all jobs which it owns are automatically removed.

EXEC makes job 0 the owner of the daughter job.

EXEC_W makes the parent the owner of the daughter job. The parent is suspended whilst the daughter job is running. SuperBASIC and any other jobs continue to run.

PASSING INFORMATION TO JOBS

QDOS defines mechanisms for passing useful information to jobs upon their creation but EXEC and EXEC_W in their standard form provide no support for this facility.

Q_Liberator has been designed to exploit QDOS to the full and so three closely related procedures are supplied to complement EXEC and EXEC_W. They are QX, QW, and QX_JOB0. They share a common syntax which is described below, but first let us make plain the differences between them.

QX loads and starts an object program making the parent the owner. The parent continues to run.

QW loads and starts an object program making the parent the owner. The parent is suspended until the daughter is complete. (cf EXEC_W)

QX_JOB0 loads and starts an object program, but makes the owner Job 0. (cf EXEC) Since Job 0 cannot be removed, using QX_JOB0 will spare the new job from a premature death if its parent is removed. It is only useful within programs.

THE PROCEDURE QX

The simplest form of QX is:

QX objectname

In this form the procedure behaves identically to the EXEC procedure except that:

a) There is no need to supply the extension _OBJ, since QX assumes that you are running a Q_Liberator object program.

b) The job is given a priority of 8 whereas EXEC gives it 32 (except when using the Toolkit).

c) The owner is the parent job.

Like EXEC, QX can be typed directly at the keyboard or used within a compiled program. When used as a direct command the parent is job 0.

For example QX MDV1_DEMO_SORT

This has the same effect as EXEC MDV1_DEMO_SORT_OBJ.

PASSING A COMMAND STRING

It is very useful to pass information to a program when it is started. For example a program which prints a file could be passed the file name or the heading for the top of each page. QDOS provides facilities to pass a command string to a job via its stack when it is created, but few programs exploit this feature. QX makes this possible for Q_Liberator programs.

Using Q_Liberator, this command string can be any string literal or string variable up to a length of 127 characters. If you wish to pass numeric data to a job then it must first be moved to a string. The command string can also be a SuperBASIC name, but then the range of characters available is restricted.

To pass such a string it must be given as the first parameter after the object name in a QX procedure call.

```
eg QX MDV1_PRINTFILE,"accounts_data July 1986"
   QX MDV2_spooler,contents_doc
```

In your program the command string appears automatically in a reserved string variable called CMD$. This will contain an empty string, length 0 if no command has been passed. This is the only special characteristic of CMD$; it can be used as a normal string variable throughout the rest of the program.

When developing programs with the interpreter to work with a command string, you will need to set up CMD$ manually to test the program.

PASSING CHANNELS TO JOBS

Finally QX can be used to pass a list of channels to a daughter job. Such channels must already have been opened by the parent job or they will cause a runtime error. They are entered into the daughter job's channel table as being already OPENed . They must not be reOPENed or CLOSEd by the daughter job or behaviour will be unpredictable. In general you need not worry about closing channels because QDOS tidies up for you when the job is removed.

Channels passed to a job in this way can be accessed by both parent and daughter job. This means that 2 or more jobs could all write to the same file without any 'in use' errors occurring.

The first channel in the parameter list is passed to the new job to replace its own channel 0. The second replaces channel 1. Thereafter the channels which are replaced number sequentially from 3. Channel 2 ought to be reserved for LISTing and so cannot be passed.

```
eg QX mdv1_testprog,#3,#3
```

Start testprog using the parent's channel 3 as testprogs's channel 0 and also as its channel 1.

If you want to leave a channel as it is, then a gap can be left in the parameter list by typing a comma.

```
QX mdv1_demo,"TITLE",,#1,#4
```

Start testprog, passing "TITLE" as the command string. It will use its own channel 0, the parent's channel 1 as its channel 1, and the parent's channel 4 as its channel 3 !

Remember, the channel *numbers* relate to the parent job; the *position* of the parameter determines the channel which is replaced in the daughter job.

A example which can easily be tried should help to clarify the above. Enter the following 1 line program and compile it with the name MDV1_COMMAND.

```
10 PRINT cmd$
```

All it does is print the command string which it is passed. Now type

```
QX MDV1_COMMAND,"Where am I now?"
```

This will result in the program printing the command string "Where am I now?" on its own channel 1. Now the fun starts. Try

```
OPEN #3,scr_50x50a50x50
QX MDV1_COMMAND;"Inside your window",,#3
```

The message appears inside the window which you just OPENed because COMMAND is using SuperBASIC's channel 3.

WORKING WITH PIPES

A pipe is a one-way connection between two channels. Pipes are a very useful means of passing messages or data between jobs. Messages are PRINTed into one end of a pipe and retrieved from the other end using INPUT. In the following description we shall refer to the end which PRINTS as the *active* end and the other as the *passive* end.

A pipe has a fixed length, determined when the active end is opened and behaves as a 'first in first out' buffer.

The active end of a pipe can be opened with a normal SuperBASIC OPEN. The length is appended to the device name PIPE.

```
eg   OPEN #4,PIPE_1024
```

The passive end of a pipe can only be opened with the Q_Liberator extension Q_PIPE. There are two forms:

```
Q_PIPE #pipe_chan
```

This takes channel pipe_chan, as passed by another job using QX, assumes its a pipe already opened, and opens the passive end. The passive channel id replaces the active one in the job's channel table. Either the parent or the daughter can elect to open the passive end, permitting pipes to be set up in both directions.

There is also a form of Q_PIPE for creating a pipe between two channels owned by the same job.

```
Q_PIPE #active to #passive
```

Here #active is a pipe already actively opened and #passive is an unused channel number less than #active. You will get a 'bad parameter' if this is not the case. It is a useful convention to make the active end an even channel number and the passive end odd.

Such a pipe can serve as a useful temporary memory buffer.

```
eg  10 OPEN #4,PIPE_256
    20 Q_PIPE #4 to #3
    30 PRINT #4,"plumbing"
    40 INPUT #3,a$
    50 PRINT a$
```

If you completely fill a pipe with data, the active end will wait until the pipe is emptied. End of file (EOF) is signalled at the passive end when the active end is closed.

There is a demonstration showing this technique being used to create a sorted microdrive directory in DEMO_PIPEDIR.

USE WITH QJUMP TOOLKIT II

This Toolkit also contains procedures which support the creation of pipes between jobs and a host of other useful functions. Q_Liberator was designed to be compatible with, and to complement this product.

The Toolkit procedures and functions have been extensively tested with Q_Liberator. Almost all will work correctly in compiled programs. There are a few functions and procedures which are not useable (eg ED), and some which should be used with care (wildcard commands). PARNAM$ and PARSTR$ cannot be used because they require interpreter data structures which are not emulated. EW and EX had problems in Toolkit version 2.05.

Default directories are supported throughout Q_Liberator and the object programs which it produces.

The extended EXEC command, EX, contained in the Toolkit can pass a command line to a Q_Liberator program in the same way as QX. Pipes can also be created between a chain of jobs. Q_Liberator is the ideal tool for writing short filter programs to exploit this.

The convention adopted for channel numbering when writing filters is

#0 is the input channel
#1 is the output channel

Other channel numbers passed to the job start from #3 as with QX.

The following is an example of a short filter program, DEMO_PAGER which splits a document into numbered pages, putting a title at the top of each. End of page can be forced by placing '.pa' at the start of a line. An example of its use to print a text file on a printer might be:

```
EX demo_pager_obj,flp2_textfile,ser1;"AGENDA"
```

```
10  REMark DEMO_PAGER
20  REMark page size is 72
30  REMark
100 L=0: P=1
105 title
110 REPeat page
115   IF EOF(#0) THEN formfeed: STOP
120   INPUT #0,a$
130   IF a$="."pa" THEN
140     formfeed: title
150   ELSE
160     PRINT #1,a$: L=L+1
170     IF L>64 THEN formfeed:TITLE
180   END IF
190 END REPeat page
200 :
210 DEFine PROCedure formfeed
220   PRINT CHR$(12);:P=P+1: L=0
230 END DEFine
240 :
250 DEFine PROCedure title
260   PRINT cmd$,"Page : ";P\\\
270 END DEFine
```

SETTING THE PRIORITY WITH QX

QX, QW and QX_JOB0 can also set the priority of the job which they load. The priority is simply specified as a number or variable anywhere in the parameters for QX.

```
eg QX mdv1_testprog,"command string",100
```

Start testprog with a priority of 100 and pass a command string.

A string variable or unset name is treated as a command string, a numeric variable or literal is treated as the priority and a variable or number preseeded with # is treated as a channel to be passed.

Chapter 11 Error Trapping

Writing and running computer programs is an activity fraught with errors. How many times have you seen 'not found', 'bad or changed medium', 'error in expression' etc, at a critical point in operations?

In professional programs, considerable attention has to be given to trapping errors so that recovery where possible takes place automatically. If the user must be troubled with an error message then the program can present it in a meaningful way.

When working with the SuperBASIC interpreter, you can often recover manually from errors by, for example, listing the program to see what was expected and restarting at a specific point.

When a program has been compiled however, this is not possible because the source form is no longer present. It becomes essential to include some error trapping routines in the program.

EXISTING ERROR TRAPPING FACILITIES

Most QL systems are equipped with either a JM or AH ROM. You can check which yours has by typing PRINT VER$. The versions of SuperBASIC in these QLs provide no support for programmed error trapping whatsoever. Manual error recovery is possible with RETRY and CONTINUE.

A few of the later QLs have JS or MG ROMS. These implemented a form of error trapping based on the WHEN ERROR keyword, but unfortunately the implementation itself contained errors and was never formally documented. Consequentially few programs are written to use this error trapping. For these reasons this form of error trapping is not supported by Q_Liberator.

Another approach to error trapping is to turn the procedures which commonly generate errors eg OPEN, into functions such as FOPEN. These return an error code to the program as the value of the function. A considerable number of such functions is contained within the Toolkit, and in many disk system ROMS. Their use is fully supported by Q_Liberator.

Q_Liberator has an alternative way of handling errors, suitable for any QL ROM.

Q_LIBERATOR ERROR TRAPPING

Every Q_Liberator program automatically contains a rudimentary form of error trapping which can help to avoid disastrous failures. This is the 'Retry' mechanism described in chapter 6. Whenever a call to a ROM routine returns an error code, you are invited to intervene manually and repeat the operation.

Secondly Q_Liberator provides a suite of SuperBASIC extensions which let you selectively trap errors reported from any ROM procedure. These can be used in both compiled and interpreted programs on any version of the QL.

TURNING ON ERROR TRAPPING

Before you can trap errors from a procedure its name must be added to an internal list using the procedure Q_ERR_ON. We shall refer to this as the *error trap list*. The parameters for Q_ERR_ON are one or more strings containing the procedure names to be trapped. Q_ERR_ON will give a 'bad parameter' error if any name is not a machine code procedure. Note that functions and user written procedures cannot be error trapped in this way.

```
eg   Q_ERR_ON "OPEN"
     Q_ERR_ON "OPEN","OPEN_IN","INPUT","COPY"
```

You can print the complete error trap list on channel 0 using the procedure Q_ERR_LIST, which takes no parameters.

Q_ERR and Q_ERR$

When an error is detected by a procedure on the error trap list, your program will not stop with a message. Instead the error number is stored internally and the procedure returns normally. A program can check if an error occurred by using the function Q_ERR, which returns the last error number, or 0 if no error occurred. Q_ERR ought to be tested every time a procedure on the error trap list is called, but this need not be in the next statement since its value is only overwritten on the next call to a trapped procedure.

```
eg      10 Q_ERR_ON "INPUT"
        20 INPUT x
        30 IF Q_ERR<>0 THEN PRINT "Error ";Q_ERR;" detected"
```

The error numbers returned by Q_ERR are the standard QDOS error keys and will normally be negative. To assist in producing error messages a function Q_ERR$ is included in the demonstration library. This will return a string containing the QDOS error text for any error number. It is reproduced at the end of this chapter to serve as a list of error numbers.

TURNING OFF ERROR TRAPPING

Once a procedure has been placed on the error trap list it stays there even if you type NEW, CLEAR or LOAD another program. The only way to clear the error trap list is to use the procedure Q_ERR_OFF.

Q_ERR_OFF will remove one or more procedures from the error trap list. It takes one or more strings as its parameters in the same way as Q_ERR_ON. However if no parameters are supplied then Q_ERR_OFF will remove all procedures from the error trap list.

```
eg  Q_ERR_OFF "INPUT","COPY"
    Q_ERR_OFF
```

Compiled programs which use error trapping each have their own error trap list, which does not interfere with the interpreter's error trap list.

A WORD OF CAUTION

The error trapping facilities presented here require care in their use. If you turn on error trapping and omit to test Q_ERR, you can have the illusion that your program is operating correctly when it is in fact generating errors.

If you are getting strange results, check what is on the error trap list.

ERROR TRAPPING EXAMPLE

As a simple example of Q_ERR here is a robust numeric INPUT procedure which won't stop with 'error in expression' if alpha characters are typed and which will give a meaningful error if 'buffer overflow' occurs.

```
100 REMark DEMONSTRATION OF ERROR HANDLING
110 REMark demo_qerr
120 :
130 REPeat demo
140   numinput x
150   PRINT x
160 END REPeat demo
170 :
180 DEFine PROCedure numinput(n)
190   Q_ERR_ON "INPUT"
200   REPeat getnum
210     INPUT "Number >";n
220     IF Q_ERR=0 THEN EXIT getnum
230     BEEP 200,10
240     PRINT
250     IF Q_ERR=-17 THEN PRINT "Only numbers please"
260     IF Q_ERR=-5 THEN PRINT "Too many characters"
270   END REPeat getnum
280   Q_ERR_OFF "INPUT"
290 END DEFine numinput
```

Finally, here is the listing of the function Q_ERR$ which returns the last QDOS error as a string.

```
1000 DEFine FuNction Q_ERR$
1010 REMark demo_qerr
1020   LOCal e
1030   e=Q_ERR
1040   SELect ON e
1050     =0  : RETurn ""
1060     =-1 : RETurn "not complete"
1070     =-2 : RETurn "invalid job"
1080     =-3 : RETurn "out of memory"
1090     =-4 : RETurn "out of range"
1100     =-5 : RETurn "buffer overflow"
1110     =-6 : RETurn "channel not open"
1120     =-7 : RETurn "not found"
1130     =-8 : RETurn "already exists"
1140     =-9 : RETurn "in use"
1150     =-10: RETurn "end of file"
1160     =-11: RETurn "drive full"
1170     =-12: RETurn "bad name"
1180     =-13: RETurn "transmission error"
1190     =-14: RETurn "format failed"
1200     =-15: RETurn "bad parameter"
1210     =-16: RETurn "file error"
1220     =-17: RETurn "error in expression"
1230     =-18: RETurn "arithmetic overflow"
1240     =-19: RETurn "not implemented"
1250     =-20: RETurn "read only"
1260     =-32: RETurn "bad line"
1270     =REMAINDER : RETurn "error "&e
1280   END SELect
1290 END DEFine Q_ERR$
```

Chapter 12 Job Control

When working with multitasking programs, it is useful to have procedures to list which jobs are currently running, to remove jobs which are no longer needed, and to set the relative priority of jobs.

Such procedures are available from many sources. They are included in the Toolkit, on most disk system ROMs, have been published in magazines and books, and are available from the QUANTA (QL user group) library.

For those who have no access to these routines, we have included a suite of procedures to control jobs in the file QJOB_BIN. This file is an optional extra which can be omitted from the BOOT program if required. Whilst these procedures perform in roughly the same manner as other job control procedures, they have some advantages and are generally useful. They have been given short names because they are often typed.

LISTING JOBS

QJ [#channel][,owner_job]

This procedure lists the tree of jobs starting from the specified owner job to a given channel. If no channel is specified then channel 2 is used. If no owner job is specified then job 0, SuperBASIC is assumed and all jobs in the system will be listed. The format of the listing is best shown by example.

Typing QJ might produce the following:

```
Job Owner Size Priority    Name
  0     0  20k  S 32       BASIC
  1     0  10k     8     Q demo_1
  2     1  15k  S  8     Q demo_2
```

where Job is the job number,
Owner is the job number of the owner,
Size is the memory area occupied by the job,
Priority is the priority on a scale from 0 (inactive) to 255,
Name is the job's name (if it has one).

The 'S' before the priority indicates that a job is suspended, eg waiting for the keyboard or another job.

The 'Q' before the name indicates a Q_Liberator job.

Note that in the example, job 2 is owned by job 1. If you wanted to see only the tree owned by job 1 then

```
QJ 1
```

would display the following:

```
Job Owner Size Priority   Name
  1     0   10k      8    Q demo_1
  2     1   15k  S   8    Q demo_2
```

If you want to process this list with a program then you can divert the listing to a channel other than the screen. A useful technique is to list the jobs into a PIPE, both ends of which are available to the same program. The records can then be read back from the PIPE into an array and processed as required. Note that the layout of the fields is fixed to make this easy.

REMOVING A JOB

The procedure QR will remove, ie terminate, a given job. If a job owns other jobs, then they will be removed also. It is not possible to remove job 0. The format is:

```
      QR jobname [,error_code]
or    QR jobnumber [,error_code]
```

As you can see, the job can be specified by name or number. The optional error code, if present, is passed back to the program which started the job, eg as the result of EXEC_W or QW. It can be trapped using Q_ERR error trapping. If no error code is specified, 0 is returned.

CHANGING THE PRIORITY OF A JOB

The procedure QP will set the priority of a job to a given value in the range 0 to 255. A priority of 0 means that a job is inactive and uses no CPU time.

```
      QP jobname,priority
or    QP jobnumber,priority
```

FINDING THE CURRENT JOB NUMBER

It can be useful for a job to know its own job number. The function Q_MYJOB will return this as an integer.

```
eg    PRINT Q_MYJOB
```

CURSOR CONTROL

Each console device has a cursor associated with it. It is normally only turned on during an INPUT statement. It is useful to be able to enable the cursor at other times, in particular to allow Control-C to switch the keyboard to that device. The cursor will flash when the keyboard is attached to it.

```
Q_CURSON [ #channel ]
```

will turn on the cursor for a given channel. The default is channel is 1.

```
Q_CURSOFF [ #channel ]
```

turns it off again.

Chapter 13 Solving Problems

This chapter is designed to help you if you experience problems with Q_Liberator.

PROBLEMS WITH MICRODRIVES

If you find that you cannot read either the Master or your Working copy, and you normally do not experience loading problems, then it is possible that the microdrive is defective. In such circumstances we will replace it free of charge if it is returned to us.

If both microdrive cartridges will not read then there is a fair probability that your machine is misaligned. We will replace the microdrives if you return them, but if the problem persists, your machine should be serviced.

Note that we can tell how many copies have been made from a Master. Claims that a Master does not read when it has in fact expired will be viewed with suspicion.

PROBLEMS WITH COMPILED PROGRAMS

At some time you may come across a program which does not function correctly when compiled or worse still, which crashes the machine. Before assuming that there is an error in Q_Liberator, please check the following:

Does the program run correctly under the interpreter every time?

Did you ignore warnings at compile time? If so, go back and check them.

Try running the program with QW in place of QX. If it now runs correctly the problem is likely to be keyboard handling. Try enabling the cursor.

If the program uses assembler extensions,

Are you sure that the correct versions are loaded?

Do they make assumptions which are invalid when run from other than job 0 ? For example we have seen routines to set up user defined graphics which have a hard coded reference to one of the SuperBASIC channels.

The same applies to machine code routines which are CALLed.

If the whole system crashes,

It is possible that your program is running out of heap or stack at a critical point. Try increasing these parameters using QLIB_PATCH and see if it makes any difference. Use the statistics option.

Are you accessing a channel number larger than the channel table allows? Again QLIB_PATCH can help.

If all else fails, please try to isolate the error down to a small program which demonstrates it consistently. Please send the program, a description of the error and as much supporting documentation as possible, to the address below. Include the serial number of Q_Liberator and don't forget your telephone number and address.

Please do not telephone with such problems; it is not realistic to solve them in this way.

Remember that Q_Liberator has been extensively tested before release. The solution to most problems is contained within this manual. Please read it carefully and persevere. Check too for any additional INFO files which may have been supplied.

Address for all correspondence:

Liberation Software
43 Clifton Road
Kingston upon Thames
Surrey
KT2 6PJ

Chapter 14 Release 3 Extensions

In general release 3.2 compiles programs faster than the budget compiler. The generated code is more concise and also runs faster. In addition to these advantages, release 3 has an exciting range of features, described in this chapter.

INTEGER FOR VARIABLES

The SuperBASIC interpreter insists that the control variable of a FOR..ENDFOR loop be floating point. Release 3.2 supports 2 methods for speeding up compiled programs by allowing integer FOR variables.

The first method is suitable for most programs and is achieved via a new option, AUTO (-AUTOF if using in the command line interface). This instructs the runtime system to treat any FOR variable as an integer if the start, end and step are all integer quantities. If the same variable is later used in a floating point FOR loop, then the runtime system alters its type to a float automatically. Q_Liberator can do this because it knows a variable's type at runtime, a prerequisite for a full implementation of the language.

Selecting the AUTO option can greatly speed up many programs, particularly if the FOR variable is used as an array index, but there is one drawback which may be encountered. If such a FOR variable is currently an integer and used as a parameter to an INPUT statement, then INPUT won't let you enter a floating point number into the variable. INPUT does not know that the variable's type is changeable. You can get round this easily by INPUTing to another float variable, then assigning it to the FOR variable. This will change the type to a float.

AUTO can also be turned on or off for any program after compilation using the QLIB_PATCH utility.

DEF_INTEGER

The second method for supporting integer FOR variables is through the pseudo keyword, DEF_INTEGER. DEF_INTEGER should be followed by a list of floating point variables which are to be treated as integers by the compiler, even though their name does not end in %. If a variable named in a DEF_INTEGER statement is used as a FOR variable, then the compiler generates a fast integer loop. Unlike AUTO variables, the type of such variables is fixed, and the compiler can generate slightly faster code.

Any DEF_INTEGER statements must be placed right at the start of a program. Only REMarks, other DEF_INTEGER statements and the special keywords EXT_FN, EXT_PROC and GLOBAL (described later) can precede DEF_INTEGER. DEF_INTEGER is totally ignored by the interpreter.

Example 10 DEF_INTEGER i,j,k : REMark Treat i,j and k as integer

DEF_INTEGER is of course also useful outside of FOR loops to get the benefit of integer arithmetic. For maximum performance (and smaller code), the $$i directive which makes the compiler treat constants as integers should be used either globally or locally when working predominantly with integer variables.

Be aware when testing programs which use DEF_INTEGER, that the interpreter will regard such variables as floats and that / (divide) always produces a floating point result. DIV can be substituted if appropriate.

COMPATIBILITY WITH REL 2 OBJECTS

The release 3.2 runtime system will support old programs compiled under release 2 or the budget compiler. However it is generally worth recompiling with release 3 to make them both smaller and faster. Note however that rel 2 objects cannot be used as resident procedures, or processed with Qlib_Patch .

INTEGRATION WITH QLOAD

QLOAD is our indispensible utility for loading interpreter programs in much the same time as compiled programs. Because the files used by QLOAD are identical in format to rel 3 workfiles, Q_Liberator can compile from such '_sav' files directly. Using QSAVE followed by LIBERATE is similar to the two stage method of compiling large programs on unexpanded systems. It has the advantage of ensuring that the latest interpreted version of a program is always saved and consistent with the compiled program.

ROMABLE CODE

All Q_Liberator 3.2 object programs are ROMable provided that any extensions linked to them using $$asmb are also ROMable, ie they should not have any storage areas within their code.

The utility program RPM (Romable Program Manager) is available to support creation of ROMs containing any desired mixture of compiled programs, subroutine libraries and assembler extensions.

EXTERNAL PROCEDURES

One of the great strengths of SuperBASIC is its provision for extending the language with new resident procedures. (Throughout this section, the term 'procedure' includes functions too.) Such extensions are normally loaded into the resident procedure area using RESPR and LBYTES and remain available until the machine is reset. To date it has not been possible to create such extensions without recourse to assembly language programming.

Q_Liberator Release 3.2 changes this. Now the compiled procedures and functions within any Rel 3.2 object file can be called by other programs, just as if they were built in procedures! We will term such compiled procedures 'external procedures', or just 'externals', because they have been separately compiled from any program which uses them. External procedures can be linked to a program in a variety of ways as described below.

An object file can contain 0,1 or more resident procedures (or functions). There is no limit to the complexity of an external procedure - it could if required be an entire program. They can be called interactively from the interpreter, by interpreted or compiled programs with parameters passed back and forth. Externals can even call other external procedures.

There is also no limit to the number of external procedures which an object file can contain, but since it is not always desirable to have all the procedures in a file available externally, Q_Liberator requires that you indicate the procedures which are external by placing the compiler directive 'REMark $$external' on the line preceding the DEFine statement. Alternatively you can use 'REMark $$ext_all' which indicates that all procedures and functions are external. As an example, consider the following short program:

```
10 REMark $$external
20 DEFine PROCedure SQUARE(x)
30 PRINT x*x
40 END DEFine 45 :
50 REMark $$external
60 DEFine FuNction FRA(x)
70 RETurn x-INT(x)
80 END DEFine 85 :
90 SQUARE 10: PRINT FRA(1.5)
```

If this is compiled, the object program (assume its called demo_obj) would contain SQUARE and FRA as externals. To indicate this, Q_Liberator prints their names at the end of compilation. Note that this program can still be executed as a job in which case line 90 would be executed. When used as an external procedure file however, line 90 would never be executed as it is not within any procedure.

Candidates for turning into external procedures might be that set of useful routines which you always include in your programs, or a graphics routine library, or perhaps a utility which you want to make resident.

In the following sections we will see three different ways in which these external procedures can be used.

USING EXTERNALS AS RESIDENT PROCEDURES

You are probably already familiar with the way in which additional procedures can be added to SuperBASIC using RESPR and LBYTES. Externals can be used in the same way. If you have Toolkit 2, simply LRESPR the object file. Otherwise you need to note the size of the code area of the object program as displayed at the end of compilation. (You can also get it from a WSTAT type of command). Then type something like:

```
base=RESPR(600): LBYTES mdv1_demo_obj,base: call base
```

assuming the code size in the example is 600 bytes.

After loading the external procedures can be used by any job in the system, indeed by several jobs simultaneously if required. Try it with the demo program.

It is NOT possible to load externals into the resident procedure area in a compiled program - they must be always be loaded by an interpreted program prior to execution.

USING EXTERNALS AS OVERLAYS

Resident procedures are very useful, but they have a big drawback - once loaded they can never be removed. If you continually load more and more of them, sooner or later you will run out of memory.

But of course, Liberation Software has the answer - overlays. An overlay is simply a file containing external procedures, but it can be loaded and unloaded as and when required. When they are loaded, the procedures behave like resident procedures. When they are unloaded, all space which they formerly occupied is released. Overlays can be loaded and unloaded by both the interpreter and compiled jobs, but there is an important difference in their behaviour. We term a program which calls overlays a ROOT program.

When an overlay is loaded by a compiled program, ONLY that compiled program can access the externals in the overlay. This is not true of overlays loaded by SuperBASIC. In this case the externals behave just like resident procedures and other jobs may use them. You must ensure however that you don't unload an overlay while a job is using one of its procedures - a system crash would be inevitable.

Overlays find application as a means of supporting large programs in minimum memory. The major subfunctions of a program suite should be made into overlays which will individually fit into the memory available. Then a small root program can load and unload the appropriate overlays as different functions are called.

If you want to work with overlays, then you should ensure that the file QLIB_OVL has been loaded as a resident procedure. (BOOT normally does this). This file contains the extensions OVERLAY and UNLOAD described below. If you use overlays in a compiled program you can link these extensions to your program with the following directive:

```
REMark $$asmb=flp1_qlib_ovl,0,10
```

THE PROCEDURE OVERLAY

Syntax: OVERLAY overlay_number,exteral_file

Errors:

-3	Out of memory
-7	File not found
-8	Already exists - Overlay number occupied
-12	Bad name - not a valid file or no external procedures in file

This procedure loads an object file into memory (common heap) and links the external procedures which it contains to the current job. If the job is the interpreter, they are marked as machine code functions and procedures in the name table.

OVERLAY requires two parameters. The first is an OVERLAY NUMBER (1 to 15) which is used to identify the overlay in a job specific overlay table. The second is the name of the object file containing them. It is the responsibility of the user program to keep track of which overlay numbers are occupied.

For example, the procedure and function in the file DEMO_OBJ could be made available by typing

```
OVERLAY 1,MDV1_DEMO_OBJ
```

Up to 15 different files can be simultaneously used as overlays. An overlay can itself call other overlays subject to this maximum. Note that OVERLAY only works with rel 3.2 object files. Trying to OVERLAY other files of machine code procedures will not work.

THE PROCEDURE UNLOAD

Syntax: UNLOAD [overlay_number]

This procedure removes one or all overlays from memory. All the procedures within the overlay disappear and the space they occupied is reclaimed. A subsequent attempt to use a now missing external procedure will result in a QDOS 'Bad name' (-12) error. This is true for both compiled and interpreted programs.

The optional parameter specifies which overlay is to be unloaded. If none is specified then all overlays are unloaded. When overlays are used with the interpreter, they remain even after NEW has been typed. UNLOAD is the only way to clear them.

EXTERNALS AS COMPILED SUBROUTINE LIBRARIES

The third method of using externals is only available to compiled programs. With these, it is possible to LINK an object file containing external procedures to a program at compile time. The technique is similar to that described for assembler extensions on page 9.3 of the user manual. The directive $$asmb is used, specifying the object file name but with no initialisation address and no table address. For example,

```
10 REMark $$asmb=mdv1_demo_obj
```

when placed at the start of a program would allow that compiled program

to use the externals SQUARE and FRA. For each compiled subroutine library used, an entry in the overlay table is created. The runtime system allocates these from 15 downwards, leaving the lower range for externals.

This feature is useful to speed development of large programs. As a section of code is completed, it can be compiled and treated as a library. The source code for that section can be deleted or hidden from the compiler with the $$off directive.

COMPILING PROGRAMS WHICH USE EXTERNALS

At compile time Q_Liberator needs to know for certain the precise type associated with each name in a program, otherwise an 'Ambiguous name' error will be reported by the compiler. In general, if you can RUN a program then all names will be unambiguous and compilation will be successful. However when a program uses externals, the compiler must be explicitly informed about all references to external procedures and functions by pseudo keywords placed within the program IF those externals are not currently loaded into the system.

In general you should write your external routines first, test them and then compile them. To avoid ambiguities, ensure that the interpreted source form of the externals has been deleted from memory. Now load the external file as an overlay and code and test the root program. When you are ready to compile the root no changes have to be made if you have sufficient memory to leave the overlay loaded. If you have not, you must UNLOAD the overlay and inform QLIB of the procedures which it contained by using the pseudo keywords EXT_PROC or EXT_FN.

EXT_PROC and EXT_FN

Syntax: EXT_PROCstring [, string]...
EXT_FN string [, string]...

These procedures inform Q_Liberator of external procedures or functions which are not currently loaded. Each parameter MUST be a STRING containing a procedure or function name. These keywords have no effect on interpretation, but a syntax check to ensure that the parameters are strings is carried out when the program is run under the interpreter. EXT_PROC and EXT_FN must be placed at the beginning of the program. They can also be used to define the types of assembler extensions which are linked using $$asmb directive for occasions when they are not resident at compile time.

For example, if a program makes use of the FRA and SQUARE defined earlier, and the file demo_obj has not been loaded as an overlay or in the resident procedure area, then the following statements should be placed at the start of the program:

```
EXT_FN "FRA" EXT_PROC "SQUARE"
```

If this is not done then Q_Liberator will think that FRA is a variable and SQUARE will give an 'ambiguous name' error.

VARIABLES AND EXTERNAL PROCEDURES

The treatment of variables when a program calls an external procedure is a logical extension of the SuperBASIC concept of LOCAL variables. All variables used within an external file are considered LOCAL to that external ie they are only accessible by the procedures within the external file. Thus the value of a variable in a root program is not influenced if a similarly named variable is altered within an external. Externals behave as program modules, insulated from the programs which call them. There are however two very important exceptions.

The first exception concerns any parameters passed to an external. If the value of a parameter is altered by an external procedure then the corresponding variable in the main program is also altered as you might expect. Compiled programs can pass any type of parameter back and forth between externals, including arrays. Furthermore, since Q_Liberator is a true SuperBASIC compiler, the types of the actual parameters are used within the external.

When calling externals from an interpreted program, only scalar parameters ie floats, integers and strings can be passed. See the implementation notes later in this chapter for an explanation.

The second exception concerns GLOBAL variables. A global variable is a variable defined in a root program which is to be accessible by one or more external routines. Global variables must be indicated to the compiler by means of a GLOBAL statement. They cannot be used with the interpreter.

THE PSEUDO KEYWORD GLOBAL

Syntax: GLOBAL variable [, variable] ...

This keyword when placed at the start of a program indicates that all variables in the list of parameters are to be treated as GLOBAL. Any externals which this program calls should also contain a GLOBAL statement if the global variables are to be accessed. One GLOBAL statement serves for all the procedures in a file and only those globals which are actually referenced need be specified. ie The external's global variable list may be a subset of the root program's global variable list. Only compiled programs can share global variables. Global statements are simply ignored by the interpreter. This may change in a subsequent release.

IMPLEMENTATION NOTES

There are important differences in the behaviour of externals when called from the interpreter and when called by a compiled job. Before it can be executed, an external job must allocate it own variables area and have available a Q_Liberator runtime data area containing amongst other things a heap area, a stack area and a channel table.

EXTERNALS CALLED BY COMPILED JOBS

When called by a compiled job, an external uses its host job's data area and so can use any channels already opened or can open its own. You must be careful to allocate sufficient stack in the root program to accomodate any externals which are called.

The external's variables area is created on the first call to ANY external procedure within the external file. It remains until the job ends or, if the external is being used as an overlay, until the overlay is removed. On subsequent calls a change of pointers is all that is necessary to provide the correct environment.

Note that the values of all variables within an external remain valid between procedure calls, ie they are static.

EXTERNALS CALLED BY THE INTERPRETER

When an external procedure is called by the interpreter, a new job has to be created to provide the runtime data area needed to execute the external. This is done on each and every call to an external routine.

After creation, any parameters to be passed are copied to the new job along with a list of opened channels. The external can freely use any channels which have already been opened, but if it opens further channels, they will not be present when outside of the overlay. You must be very careful not to allow the external to close channel 0, or you will be unable to enter SuperBASIC lines when you return from the external.

The job is now activated and the interpreter is suspended. When the new job ends, the interpreter is activated, the parameters are copied back and the external job is removed.

Parameters have to be copied because SuperBASIC has a tendancy to move around memory. It is not possible to pass array parameters because the overheads would be too great. If you try it you will get a 'Not implemented' error.

Because a new job is created each time a procedure is called, variable values are lost between procedure calls. This mechanism is considerably slower than that used by compiled jobs, but is fine for development purposes and for calling compiled toolkits or utility routines.

Note that whilst the interpreter can call compiled external routines, it is NOT possible for external routines to call interpreted procedures.

SEARCHING FOR NAMES

At compile time, Q_Liberator only needs to know the name and type of each procedure and function; the address is resolved at runtime. When a root program or external is first initialised it searches in various places for the procedures it requires to run, using the procedure's name as its search key. The order of searching is as follows:

1. The linked resident procedures table of the current job. This contains the names of any external procedures and assembler routines linked at compile time using $$asmb.

2. (Externals only) The linked resident procedure table of the root program.

3. (Externals only) Procedures in the root program which are marked as external.

4. The SuperBASIC name table which contains the names of all built in and resident procedures.

If all searches fail then the procedure is marked as undefined. Calling it will result in a 'bad name' error. It may well become defined later if it is loaded as an overlay.

It should be clear from the above that it is possible for a compiled program to effectively override resident procedures if its own external procedures have the same name.

FREE RUNNING PROCEDURES

To further add to the rich range of facilities available through externals, we have provided a means of calling external procedures (NOT functions) such that they execute as independant jobs while the host program continues to run. Such procedures can have parameters passed to them in the normal way, but no parameters are copied back when the external job ends.

To start such a procedure simply put an exclamation mark after the last parameter. For example, if the compiler QLIB_OBJ is resident then typing

```
QLIB!
```

will start the compiler and still let you use BASIC.

COMPATIBILITY WITH QRAM

Q_Liberator and the programs it compiles thrive in the QRAM environment. They can also be processed by the QRAM utility routine to make them available on a hotkey.

COMPATIBILITY WITH QPTR INTERFACE

As supplied, Release 3.2 can be used to compile programs which use the SuperBASIC interface to the QJUMP QPTR package. The instructions in the QPTR manual describing how to modify release 3.1 to be compatible should not be followed.

CONFIGURING THE COMPILER

You can customise the compiler to suit your particular hardware configuration and define your own set of default options with the procedure QLIB_USE. This would normally be done inside a BOOT program. The syntax is

```
QLIB_USE load_device, help_device, x_pos, y_pos, "options bits"
```

All parameters are optional. You need only specify the parameters you wish to change - the others can be defaulted by using a comma.

Load_device	The device where the file QLIB_OBJ is located.
Help_device	The device where the file QLIB_HELP can be found.
X_pos	The X coordinate of the top left hand corner of the window.
Y_pos	The Y coordinate of the top left hand corner.
"option bits"	This is a 10 character string in which each position refers to an option. A '1' in a given position enables the corresponding option, a '0' disables it.

Device parameters can be up to 10 characters long, giving the possibility of specifying not only a device, but a directory.

For example, QLIB_USE flp1_abc_,flp1_abc_

The compiler would be found under flp1_abc_qlib_obj and the HELP file as flp1_abc_qlib_help.

X_pos, X_pos and the option bits string are updated in memory each time the compiler terminates and are remembered for the next compiler call.

The option bits are in the following order:

1	-STAT	default 0
2	-DEBUG	0
3	-LINES	1
4	-NAMES	1
5	-RUN	0
6	-AUTOF	1
7	BEEP	1
8	-WINDS	1
9	reserved	
10	reserved	

There is however little need to refer to this table because an easy method of deriving the parameters to QLIB_USE is available.

THE FUNCTION QLIB_LIST$

This function returns a string containing the current default value of each of the parameters maintained through QLIB_USE. Each parameter is separated from the next by a new line character. Therefore the line

```
PRINT QLIB_LIST$
```

will display all the defaults on the screen. The simplest way to configure the compiler is to load it and select the options and positions which you require as your defaults then exit. Then use QLIB_LIST$ to display the values and construct a QLIB_USE line to put into your BOOT program.

The default parameters for a microdrive based QL as set in the standard BOOT program is

```
QLIB_USE mdv1_,mdv1_,74,50,"0011010100"
```

This combination will compile most programs without any changes and debugging is simplified by the inclusion of line numbers. AUTOmatic integer FOR is turned on for speed. If you want to obtain the smallest possible object size at the possible expense of some minor changes, turn off the NAMES and LINES options, and include $$i at the start of the program to force integer mode.

MAKING THE COMPILER RESIDENT

If you have sufficient memory, you can make Q_Liberator resident and instantly available. This can be done in several different ways as summarised below. Details of the techniques behind these facilities are explained in the section on external procedures.

1) To make the compiler a permanent resident procedure

```
b=RESPR(size): LBYTES MDV_QLIB_OBJ,b : CALL b
```

where size is the length of the file QLIB_OBJ. This is given in Appendix B. It is likely to change in future releases.

2) To load the compiler as an overlay (temporary resident procedure)

```
OVERLAY 1,MDV1_QLIB_OBJ
```

3) A QLIB system file can be created with the RPM utility containing the compiler, runtime system, toolkit routines and anything else you choose to include. This file could be loaded as a resident procedure or put into a ROM. This is already done for you if you have the 64K ROM version.

The Q_Liberator menu can then be called up by typing the procedure name QLIB. If you wish you can append an option string in a string variable, in which case Q_Liberator will run automatically. If you want to continue working with SuperBASIC while the compiler is running, append an exclamation mark after the last parameter. For example,

```
QLIB "MDV1_DEMO"!
```

would automatically compile the file MDV1_DEMO_WRK or MDV1_DEMO_SAV.


Liberator

INTRODUCTION

When Q_Liberator was originally conceived, the majority of QLs were fitted with AH and JM ROMS. The later ROMS, JS and MG introduced the WHEN ERROR and WHEN variable constructs, but deficiencies in the implementation meant that they could not be used reliably although Toolkit 2 from QJUMP went some way towards correcting them. By that time we were concentrating on enhancing Q_Liberator to provide full compatibility with QJUMP products such as QRAM and HOTKEY 2 and to provide the valuable facility of external procedures and functions.

The emergence of MINERVA prompted us to revisit Q_Liberator to provide support for its dual screen mode feature and to add some enhancements we had long planned. At the same time we have implemented WHEN error and WHEN variable as they work consistently with that ROM. The result is Q_Liberator Release 3.3.

This release will run object code programs compiled by all previous versions of Q_Liberator. Note however that the 3.3 runtime system must be used with the 3.3 compiler. Use of an earlier runtimes will give QLIB error 5 - Internal error.

NOTES FOR MINERVA USERS

This is the first release which we claim to be truly Minerva compatible. For the record, all release 3 versions of Q_Liberator will run with Minerva in the single screen mode. Release 3.24, which was issued on a restricted basis to QUANTA first supported dual screen mode.

Please read the documentation supplied with Minerva as it makes several references to Q_Liberator. Compiled programs with machine code extensions which require more space on the RI stack than is available can with crash the system. Minerva prevents this by the rather dramatic action of removing the offending job. Thus if you find your program suddenly aborts without reason then try increasing the stack size with QLIB_PATCH.

Whilst the improvements to the speed of the graphics routines and floating point routines are exploited to the full by Q_Liberator, the improvements to the speed of the SuperBASIC interpreter will diminish the perceived speed up factor of the compiler when compared to the interpreter.

SUPERBASIC CHANGES WITH MINERVA

With the minor exceptions detailed below, all of the enhancements to SuperBASIC described in the Minerva documentation are supported by Q_Liberator. In some cases where there are bug fixes or obvious enhancements, Q_Liberator was already capable of handling things correctly (e.g. String SELect, FILL$, and RESPR). The TRACE routines

TRON, TROFF and SSTEP cannot be compiled - this should not come as a surprise. We have also chosen not to support FOR loops with string variables. If you really think that we should, write to us and let us know. Q_Liberator will attempt to coerce a string FOR variable to a number. If this is not possible then the runtime system will issue QLIB error 29 - string is not numeric.

Be careful in the use of Minerva's enhancements if you want your software to be portable to other ROMS.

WHEN HANDLING

The major enhancement in Release 3.3 concerns WHEN handling. This feature can only be used with the following ROMS: JS, MG variations and Minerva. To date there has been no full description of the WHEN ERRor and WHEN variable constructs which we found to contain complexities when researching their behaviour prior to implementation in Q_Liberator. The sections below are an attempt to rectify this lack of documentation.

WHEN ERROR

In chapter 11 we explained the need for error trapping in a program and described the Q_ERR facilities supplied with Q_Liberator. From Release 3.3 we have implemented error trapping which is completely compatible with the facilities originally implemented in the JS ROM and corrected in Minerva. In contrast to the Q_ERR error trapping which provides keyword specific error handling, WHEN ERRor trapping applies to all keywords.

WHEN ERRor is invoked by including a WHEN ERROr routine somewhere in the program. A WHEN ERRor routine starts with a WHEN ERRor statement and ends with an END WHEN statement. When such a routine is executed the statements between WHEN and END WHEN are ignored, but the address of the first statement is recorded. After this, whenever an error is encountered the statements between WHEN and END WHEN are executed.

For example:

```
WHEN ERRor
     PRINT 'Something went wrong': STOP
END WHEN
```

A single line version of WHEN ERROR is also possible along the lines of single line REPeats and FOR statements. No END WHEN is necessary:

```
WHEN ERRor: PRINT "Oops!"
```

WHEN ERRor routines cannot be nested inside each other in your source program. At runtime they are static. Whilst it is allowable and is often useful to have more than one WHEN ERRor within a program, only the last one encountered will be active.

ENTERING WHEN ERROR

Once it is active, the WHEN ERRor routine will be invoked whenever an error occurs within a program. With the interpreter this includes errors which occur when entering direct commands.

Once inside a WHEN ERRor, there are few restrictions on the sort of processing you can do. The environment is that of the routine in which the error occurred. In particular, local variables which existed at the time of the error are still accessible and functions and procedures can be called at will. Note however that within the error routine further error trapping is effectively turned off. If an error occurs within an error routine then it will cause the program to stop. The interpreter prints a message in the normal way except that 'in WHEN processing' is to let you know what has happened.

With compiled programs if an error occurs during WHEN ERRor processing then it is displayed on the pop up error console in the normal way with the error message preceeded by 'During WHEN,'. You then have the opportunity to Retry, Continue or Abort.

To be useful, a WHEN ERRor routine needs to be able to determine where the error occurred and what the error was. Then it may be possible to take corrective action or at least print a meaningful message. The ROM contains functions and procedures to support you.

ERLIN is a function which returns the line number at which an error occurred. ERNUM is a function which returns the error number as the usual small negative integer. As an alternative to testing ERNUM, there is a set of functions with names corresponding to the system error codes which return true (-1) if that error occurred. ERR_NF for example, returns true if a 'not found' error has occurred. The complete list of functions is listed below in the same order as the error codes in the function Q_ERR$ from chapter 11.

ERR_NC, ERR_NJ, ERR_OM, ERR_OR, ERR_BO, ERR_NO, ERR_NF, ERR_EX, ERR_IU, ERR_EF, ERR_DF, ERR_BN, ERR_TE, ERR_FF, ERR_BP, ERR_FE, ERR_XP, ERR_OV, ERR_NI, ERR_RO, ERR_BN

The procedure REPORT is useful for printing the message associated with the last error which occurred or with a given error number. Note that the default channel for REPORT is channel 0. The syntax is as follows:

```
REPORT [ #channel, ] [ error ]
```

For example:

```
REPORT              Print last error message on #0
REPORT -5           Prints 'already exists' on #0
REPORT #1,ERR_NF    Prints 'not found' to #1
```

REPORT unfortunately insists on printing a line feed after the error message.

EXITING WHEN ERROR

There are three legal ways by which you can leave a WHEN ERRor clause.

The keyword END WHEN, which must always be present at the end of an error routine, will return control to the statement after the statement which caused the error ("the error statement").

The procedure CONTINUE can be used at any point in an error routine to cause a return to the main program. If no parameter is present then CONTINUE works just like END WHEN and returns to the next statement. If you have Toolkit 2 then the functionality of CONTINUE is enhanced to allow continuation from an arbitrary line number within the program. Of course this line MUST be within the same procedure as the error statement and will typically be very close to it.

e.g. CONTINUE 200 Continue from line 200

The procedure RETRY can be used without a parameter to restart execution at the start of the statement which caused the error. As with CONTINUE, RETRY can be given a line number if Toolkit 2 is present, in which case it behaves identically to CONTINUE with a line number as described above.

Use of CONTINUE and RETRY is only possible inside WHEN ERRor. Note that although Toolkit 2 is necessary for the interpreter to run programs which use "RETRY line_number" or "CONTINUE line_number", Q_Liberator will correctly compile and execute these statements without the presence of Toolkit 2. Fortunately the syntax is accepted on any ROM supporting WHEN, so such programs can be entered and compiled, even though the interpreter would not run them correctly.

RETRY is most useful when used with the ERLIN function. Note the difference between RETRY which retries the error statement and RETRY ERLIN which will restart at the beginning of the line which includes the error statement. This gives you the opportunity to keep things tidy before the statement is retried. The example below shows how this technique can be used to catch the error in expression which occurs if text is entered into a numeric variable. Try it.

```
100 WHEN ERRor
110     IF ERR_XP THEN
120         AT 10,10: PRINT "Numbers only!"
130         RETRY ERLIN
140     END IF
150     PRINT "At line ";ERLIN,": REPORT #1: STOP
160 END WHEN
170 :
500 AT 8,7 : PRINT "      ": at 8,0:INPUT "Number ";n
510 AT 10,0: PRINT "Thank you      "
```

Be careful with expressions using ERLIN because explicit line numbers

are not automatically adjusted if you RENUMber the program.

TURNING OFF WHEN ERROR

When working interactively with the interpreter, any error routine active within your program will still be active if you interrupt execution. This can lead to confusion, particularly if the error routine ignores some classes of error. You might type a command and assume it has worked correctly because no error is reported. In reality the command has failed but there is no routine with the responsibility of informing you. To avoid this, WHEN ERRor handling can be turned off and the system returned to normal by typing 'WHEN ERRor' as a direct command.

WHEN ERROR and Q_ERR

These two different forms of error trapping do not compete in any way; in fact they compliment each other. Both forms of error trapping store the error number in the same location so the functions Q_ERR and ERNUM are in fact interchangeable.

WHEN ERRor is a global form of error trapping. Any error in a program invokes it without any other special coding being necessary. In contrast Q_ERR is specific. It only operates on procedures which have been put on its error trap list by Q_ERR_ON. However there is the disadvantage that Q_ERR must be tested after each statement which could potentially lead to an error.

When both forms of error trapping are used within the same program, putting a procedure on the error trap list with Q_ERR_ON effectively redirects all errors associated with that procedure to the Q_ERR routines. The WHEN ERRor routine will never be called for errors in that procedure. Thus one might use WHEN ERROR for general error handling and Q_ERR for specific exceptions.

WHEN ERROR IN COMPILED PROGRAMS

We have made every effort to ensure that WHEN ERRor is implemented within Q_Liberator in a manner completely compatible with the interpreter. This we have achieved for all the errors which are returned by procedure calls. However those errors listed as QLIB errors which are mainly programming errors, cannot be trapped. This is no great restriction because such errors are usually non recoverable. One consequence is that 'division by zero' cannot be trapped and will lead to an abort.

A program which uses WHEN ERRor can only be entered and compiled on a system with JS, MG or Minerva ROMS. However the object programs will run on any QL provided that the procedure REPORT is avoided. Q_Liberator will produce compatible code to support use of ERLIN, ERNUM and all the functions which test for specific errors such as ERR_NF even though those functions are not present in the AH and JM ROMs.

WHEN ERROR AND EXTERNALS

The scope of a WHEN ERRor routine does not extend to trapping errors within compiled external procedures called by a program. If error trapping is required within an external then a separate WHEN ERRor should be included within the external itself.

WHEN VARIABLE

WHEN ERRor causes a routine to be automatically called whenever an error occurs. In a broadly similar fashion, WHEN VARIABLE causes a routine to be called whenever a specified variable changes. It can be used to create event driven programs.
The syntax looks as follows:

```
WHEN expression
    statements
END WHEN
```

where expression is usually of the form:

```
Variable  relational_operator expression
```

When a WHEN clause is executed, the statements within it are ignored but the first variable in the expression is entered in a table of 'watched WHEN variables'. Thereafter, every time a value is stored in this variable the WHEN clause is invoked. If the condition following the WHEN evaluates to true then the statements which follow will be executed. More than one variable can precede the relational operator but it is important to realise that only the FIRST variable after the WHEN is 'watched'. Some examples should clarify this:

`WHEN x=100`	invoked when 100 stored in x
`WHEN x>50`	invoked when something greater than 50 stored in x
`WHEN x=y`	invoked when x is changed to equal y. Changing y to equal x does NOT invoke the routine
`WHEN x+y=0`	invoked when x is changed such that x+y=0. Changing y so that x+y=0 will NOT invoke routine

You can have as many WHEN clauses in a program as you choose, each related to the same or different variables. Changing a watched variable will result in at most one WHEN clause being executed. Thus the order in which WHEN clauses are tested can be significant and depends upon the order in which they are encountered at runtime. Unlike WHEN ERRor which is static and operates on one level only, statements inside one WHEN clause may trigger entry to another WHEN clause. The only restriction is that it is NOT possible to re-enter the WHEN clause

which is currently being processed. The example overleaf should help to clarify the behaviour of WHEN. It's worth trying it on your own system.

```
100 WHEN a=1
110   PRINT "a=1",
120   a=0: b=1
130 END WHEN
200 WHEN b=1
210   PRINT "b=1",
220   b=0: a=1
230 END WHEN
300 WHEN a>0
310   PRINT "a>0",
320 END WHEN
500 a=1
510 PRINT "end"
```

When this is executed the sequence is as follows. At 500, setting a to 1 triggers the WHEN at line 100 which is first in the list. The WHEN at 300, is not activated even though its condition is true. At 120, whilst still inside the first WHEN, b is set to 1 triggering the WHEN at 200. At 220, a is again set to 1. The WHEN at 100 is already activated and so is ignored, but the condition for the WHEN at 300 is met and is therefore triggered. Then we return from the three nested WHENs via lines 320, 230, 130 and finally back to the main program at line 510. Thus the output from the program is:

```
a=1     b=1     a>0     end
```

STOPPING WHEN PROCESSING

A variable can be removed from the watched list by a statement of the form:

```
WHEN variable
```

The first WHEN clause for the specified variable is removed. Others for the same variable may still remain in force.

WHEN VARIABLE IN COMPILED PROGRAMS

Nothing much to say here. Q_Liberator WHEN handling is precisely compatible with the behaviour of the interpreter described above. As with WHEN ERRor, WHEN handling does not extend into externals called by a program, but externals can have their own WHENs if required.

MISCELLANEOUS IMPROVEMENTS

TRACE OPTION

A TRACE option has been added to the compiler. When it is turned ON statement separators are inserted in the object code. This only

marginally increases the code size as they usually replace redundant filler bytes. The only advantage currently is that a statement number is printed on the error console after the error line number. In future we may develop a debugger for Q_Liberator code in which case the TRACE option will allow code to be single stepped. Please write to us if you are interested in such a tool. TRACE occupies the first reserved entry in the QLIB_USE parameter list.

ERROR CONSOLE

When a Q_Lib error is reported on the pop up error console in place of the RETRY Y/N prompt you can now opt to Retry, Continue or Abort by typing the appropriate character. Retry repeats the offending statement, continue ignores it and abort terminates the job. You might also spot that the border of the error console has been changed to a chequered pattern.

With Minerva in two screen mode, the error console pops up on the current default screen for that job.

FREE RUNNING PROCEDURES

The concept of free running procedures was introduced on page 14.9 of the user manual. In releases prior to 3.3, such procedures could only be started from the interpreter. Release 3.3 removes this restriction and allows compiled programs to spawn new independent jobs by a simple procedure call.

QLIB_SYS

Over the years the Q_Liberator system has grown in size and has become spread over several files. As an alternative to individually loading each file of extensions we have linked all those commonly required in one file named QLIB_SYS. QLIB_SYS was produced using RPM (of course!). The RPM control file is also supplied as QLIB_RPM for those who might want to change it to include say QLOAD/QREF or the compiler itself, QLIB_OBJ. QLIB_SYS is now part of the standard BOOT routine. QLIB_BOOT still contains the instructions to load files individually.

NEW ERROR MESSAGES

The compiler has two new error messages associated with WHEN constructs. Their meaning should be obvious.

```
Error....END WHEN without matching WHEN

Error....Nested WHEN not allowed
```

The runtime error message, 'Can't retry' is now issued if RETRY or CONTINUE are used outside of a WHEN ERRor clause.

QLIB_BIN

This contains phase 1 of the compiler, LIBERATE, and the extensions for loading object programs, QX, QW and QX_JOB0. It must be loaded by a BOOT program if you intend to compile programs. QLIB_BIN is configured during the CLONE procedure.

QLIB_RUN

This is the run time system. It must be present to run object programs except for those programs which have had the run time system linked at compile time. The second phase of the compiler itself requires this file.

QLIB_OBJ

This is the second phase of the compiler. It is loaded by the LIBERATE command and requires that QLIB_BIN and QLIB_RUN are present.

QLIB_EXT

This file is optional; it is not required by the compiler. You may choose not to load it by amending the BOOT program. It contains the following SuperBASIC extensions:

QJ, QP, QR, Q_MYJOB, Q_CURSON, Q_CURSOFF, Q_PIPE, Q_ERR_ON, Q_ERR_OFF, Q_ERR_LIST and Q_ERR.

QLIB_BOOT

This file is the source of the BOOT program created by CLONE. In its standard form it loads QLIB_BIN, QLIB_RUN and QLIB_EXT. You can create other BOOT programs (eg to load only phase 1) by editing this one.

QLIB_PATCH_OBJ

This is a utility in object form for changing certain runtime parameters without having to recompile. It requires QLIB_RUN to be resident.

CLONE

This is a BASIC program supplied in source form for making copies of the Q_Liberator system.

In addition to the above, various demos are supplied in source form with the master.

QLIB_BIN

This contains phase 1 of the compiler and the following extensions :

LIBERATE, GLOBAL, EXT_FN, EXT_PROC, DEF_INTEGER, QLIB_USE, QLIB_LIST$,-QX, QW and QX_JOB0. It must be loaded by a BOOT program if you intend to compile programs.

QLIB_RUN

This is the run time system. It must be present to run object programs except for those programs which have had the run time system linked at compile time. The second phase of the compiler itself requires this file.

QLIB_OBJ

This is the second phase of the compiler. It is loaded by the LIBERATE command and requires that QLIB_BIN and QLIB_RUN are present. The size of this file as required when making the compiler resident can be found in QLIB_BOOT

QLIB_EXT

This file is not necessary for the compiler to operate. It contains the following extensions:

QJ, QP, QR, Q_MYJOB, Q_CURSON, Q_CURSOFF, Q_PIPE, Q_ERR_ON, Q_ERR_OFF, Q_ERR_LIST and Q_ERR.

QLIB_OVL

This contains the procedures OVERLAY and UNLOAD.

QLIB_HELP

This is the HELP text file for the compiler. It need only be present on a working copy if you think you need it.

QLIB_BOOT

This file is the source of the BOOT program created by CLONE. In its standard form it loads QLIB_BIN, QLIB_RUN and QLIB_EXT. You can create other BOOT programs (eg to load only phase 1) by editing this one.

QLIB_PATCH_OBJ

This is a utility in object form for changing certain runtime parameters without having to recompile. It requires QLIB_RUN to be resident.

QLIB_SYS

This contains QLIB_BIN, QLIB_RUN, QLIB_EXT, QLIB_OVL as a single loadable file.

QLIB_RPM

The Resident Program Manager source file to used to create QLIB_SYS

CLONE

This is a BASIC program supplied in source form for making copies of the Q_Liberator system.

In addition to the above, various demos are supplied in source form with the master.