Liberator

The Definitive SuperBASIC Compiler

USER MANUAL

Q_LIBERATOR USER MANUAL

First edition 1986

Published by Liberation Software

43 Clifton Road
Kingston upon Thames
Surrey
KT2 8PJ

Contents

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 SuperBASIC's more exotic features and so a major design aim was to adhere rigidly to the SuperBASIC syntax. There seemed little point in supporting only a subset.

The result was Q_Liberator, a sophisticated tool which produces compiled programs with the following benefits:

They load in a fraction of the normal time.

They use less file space and less memory space.

They execute faster than the interpreted version.

They are secure. Q_Liberator programs are indecipherable when examined.

With a few well documented (and obvious) exceptions, virtually any SuperBASIC program can be compiled successfully. 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 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 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 hardware. Special provision has been made to ensure that large programs can be compiled and run on an unexpanded QL.

If extended memory is available then Q_Liberator will exploit it.

Q_Liberator will work with all extension disk systems which adhere to the standard QL format.

PACKAGE CONTENTS

The Q_Liberator library case contains this manual, a registration card, a *Master* microdrive containing all the Q_Liberator software and a *Working Copy* on microdrive or other QL media. The Master microdrive can be identified by its red label.

The Master copy can only be used for creating further Working Copies; it is not possible to use it to compile programs. The Master should be kept in a safe place; it is your ultimate security copy. Replacement, except in cases of faulty materials, is chargeable.

The Working copy of Q_Liberator contains a BOOT program and several files, described in detail in Appendix B. They are grouped in 3 directories:

Files beginning with QLIB_ contain the Q_Liberator system.

QLIB_BIN is the first phase of the compiler.

QLIB_OBJ contains the second phase. This file is protected against unauthorised copying.

QLIB_RUN contains the runtime system.

QLIB_EXT contains SuperBASIC extensions.

Files beginning with DEMO_ are miscellaneous demonstration programs, some of which are described in this text.

Files beginning with INFO_ (if any) contain additional documentation.

MAKING COPIES

You can freely copy all these files for your own use, with the exception of QLIB_OBJ, which can only be duplicated by creating a new working copy from the Master. Copying QLIB_OBJ by other means will stop the compiler from running. QLIB_OBJ is not needed to run a compiled program.

Appendix A describes how to make a working copy for any media from the Master microdrive.

To discourage the unlawful distribution of Q_Liberator, the number of additional copies which you can create from a Master is limited to five. Each Master and each working copy have unique serial numbers to aid in the identification of unauthorised copies.

It is Liberation Software's belief that the measures described above will not inconvenience the legitimate user.

HOW TO USE THIS MANUAL

This manual is designed 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.

Chapter 2 is a rapid introduction to compiling and running programs with Q_Liberator. All users are encouraged to work through this chapter to gain some immediate experience before tackling subsequent chapters.

Chapter 3 is primarily intended for those who have never encountered multitasking before. Advanced users may wish to skip some sections.

Chapter 4 describes how to use the compiler in detail, including the command line syntax and various directives.

Chapter 5 is a reference list of all messages which can result when a program is compiled.

Chapter 6 lists the error messages which can occur when compiled programs are running.

Chapter 7 describes how to optimise the memory usage of Q_Liberator programs. It can be ignored initially, but all programs can benefit from the sort of tuning described, and in some cases it is essential.

Chapter 8 is a detailed comparison between Q_Liberator and the interpreter. It is important to be aware of the differences and enhancements to get the best from the system.

Chapter 9 covers the use of SuperBASIC extensions written in assembly language within compiled programs.

Chapter 10 describes a suite of procedures for passing command lines and channels between jobs. Pipes are also discussed.

Chapter 11 describes how to add some error trapping to a program.

Chapter 12 describes a suite of procedures for controlling jobs. Many users will already have similar procedures, but the Q_Liberator versions offer some advantages.

Chapter 13 is for those who encounter problems with the compiler. It contains a checklist of common causes of problems and possible solutions.

COMMERCIAL USE

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

The runtime system QLIB_RUN is linked to the object program and not supplied as a separate file. (see chapter 4)

The procedures in QLIB_EXT, if used, are linked to the object program. (see chapter 9)

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

Liberation Software is notified of all such programs.

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 indispensable during the creation of Q_Liberator. The language described therein is followed precisely except where documented in this manual.

CREDITS

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 an average SuperBASIC program and to run the compiled version.

Reset your QL, place your Working Copy of Q_Liberator in MDV1_ (or FLP1_) then press F1 or F2 as you see fit. There is a BOOT program on the microdrive which 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.

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 which sorts integers, strings or floats, present on the Working Copy.

Type `LOAD MDV1_DEMO_SORT`

and wait for the cursor to reappear. Now type RUN and watch the screen. All being well you should see the 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* i.e. run concurrently with other programs.

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

Type `LIBERATE MDV1_DEMO_SORT,`

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.

Q_LIBERATOR WINDOWS

When loading is complete you will see the main Q_Liberator screen containing 3 windows.

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. When this number reaches the maximum line number, 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 number 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.

The middle window is used to enter a *command line* for the compiler. In our example no command was necessary because phase 2 of the compiler was automatically started after phase 1. We shall return to the command line later.

RUNNING THE COMPILED PROGRAM

When you see the message "Q_Liberator finished" in the command window, 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_OBJECT`

to load and run the 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, in which case the times for the compiled program are slightly longer, because the SuperBASIC interpreter is still active.

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, you can still type SuperBASIC commands because the programs are running simultaneously. We shall explore this further in the next chapter.

QX and its companion QW have other uses, explained fully in chapter 10.

SEPARATING PHASE 1 AND PHASE 2

When there is insufficient memory to hold both the source program and phase 2 in memory simultaneously, the program can be compiled in

separate phases as follows:

```
LIBERATE DEMO_SORT
```

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

```
LIBERATE
```

Now when Q_Liberator is loaded it prompts for a command line. Enter MDV1_DEMO_SORT and you will see the compiler running as before.

When it finishes you have the opportunity to compile another program or Q_Liberator can be ended by entering an empty line.

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 sufficient information to begin compiling your own programs. The next chapter gives some guidelines to ensure they will run successfully in a multitasking environment.

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 the 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, i.e. 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 *run time* 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 advantage 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 CONTROLLING

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 multitasking 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 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 the Q_Liberator compiler. You will already have seen how easy this can be if you have worked through the demonstration run.

CALLING PHASE 1

Before calling phase 1 you must LOAD the program which is to be compiled and *ensure that it runs correctly*. Q_Liberator cannot be expected to fix programming errors for you. If you have an unexpanded machine, type CLEAR to free any available memory before attempting compilation.

Phase 1 of Q_Liberator can now be started by using the LIBERATE command in one of the following forms:

a) `LIBERATE filename`
b) `LIBERATE filename,`
c) `LIBERATE filename,option_list`

Form a) calls phase 1 alone, whilst b) and c) automatically cause phase 2 to be loaded when phase 1 is complete. The option_list is described under phase 2.

'Filename' specifies the name Q_Liberator will us when forming the workfile name and later 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, but if there are extensions in your system to support default directories, then Q_Liberator will use them.

e.g.

```
LIBERATE mdv1_test
LIBERATE mdv1_demo,
```

It is not meaningful to use LIBERATE within a program. If you attempt this you will get the error 'bad name'.

If you want to compile a really large program on an unexpanded system, then you need only load QLIB_BIN when you boot the system. This occupies only 2K of memory. You can then use phase 1 to create the workfile. Now clear the program using NEW, load the runtime system QLIB_RUN and start phase 2 to complete the compilation.

CALLING PHASE 2

Phase 2 of the compiler can be started automatically from phase 1 as described above, or can be started independently by typing

```
LIBERATE
```

without any parameters.

Phase 2 gets its instructions via a *command line*. This is a string which identifies all the filenames to be used and specifies any special actions which should be taken during compilation. The compiler can receive its command line in several different ways.

When phase 2 is started independently then it is simply typed in response to the prompt in the command window.

The other ways in which a command line can be passed are described later in this chapter.

THE COMMAND LINE

A command line has the following format:

filename [option_list]

At a minimum it is just the name of a work file which was produced by phase 1. This name should be specified without the extension '_wrk'. If nothing follows the name then Q_Liberator will compile your program using standard default values throughout. The option list is described overleaf.

e.g. Command : flp1_demo

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.

COMPILER OPTIONS

One or more options, separated by spaces, can be placed after the filename to form an option list. Each consists of a short mnemonic name preceded by a minus sign. Options can be specified in upper or lower case in any order. Some options require a parameter to give further information to the compiler. Such parameters 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 details elsewhere.

-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 (e.g. 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 be run in standalone mode, i.e. without the runtime system loaded. You MUST use this option if you wish to sell your compiled program. Such programs are approximately 8600 bytes longer than programs without this option.

The device parameter specifies where the runtime system QLIB_RUN is to be copied from. e.g. mdv1_

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

Some examples of command lines are shown on the next page.

EXAMPLES OF COMMAND LINES

Command: mdv2_demo_sort -list ser1

The workfile is mdv2_demo_sort_wrk. Compile in the standard fashion but send any error messages to the printer connected to ser1. The object file will be mdv2_demo_sort_obj.

Command: mdv1_testprog -stat -noline -obj mdv2_test

The workfile is mdv1_testprog_wrk. Don't generate a line number table but include statistics. The object file will be mdv2_test_obj.

Command: flp1_graph -run flp1_ -name Demo

The workfile is flp1_graph_wrk. Link the runtime system flp1_qlib_run to the object program flp1_graph_obj. The job name is 'Demo'.

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

You now have the chance to try again or abort the compiler.

PASSING COMMAND LINES FROM PHASE 1

If you have sufficient memory to run phase 2 automatically after phase 1 then any options which are required can be specified as a second parameter to the LIBERATE command. Before calling phase 2, phase 1 combines the file name with the option list to produce a command line.

When phase 2 is started in this way it prints the command line which it recevied in the command window then runs automatically without any keyboard intervention. It will remove itself at the end of the compilation.

e.g. `LIBERATE mdv1_testprog,"-stat"`

Phase 1 produces the workfile mdv1_testprog_wrk. It then combines the two parameters into the command line 'mdv1_testprog -stat' and passes this to phase 2 which completes the compilation and terminates automatically.

PASSING COMMAND LINES USING QX

Phase 2 can also be started using the procedure QX, possibly under the control of a program. QX is described in chapter 10. The complete command line should be passed as the command string. This gives the possibility of batch operation of the compiler, i.e. many separate workfiles can be automatically compiled under the control of a program.

e.g.

```
QX flp1_qlib,"flp1_testprog -stat"
QX mdv1_qlib,"mdv1_demo -stat -list ser1 -name freddy"
```

Remember that phase 1 can only be executed as a direct command.

COMPILER DIRECTIVES

These are special REMark statements inserted into a SuperBASIC program to instruct the compiler of special storage requirements or assembler routines required at runtime. 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.

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.

The complete list of directives is given below. 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

Default maximum channel number to be used. This reserves space for the channel table. See chapter 8 under CHANNELS for more information.

REMark $$asmb=FILENAME,INIT,TABLE

This directive causes SuperBASIC extensions written in assembler to be linked into the object program during compilation. It may be specified up to 8 times. Each module can contain any number of procedures or functions.

See chapter 9 for details of how to use this directive.

EXAMPLES OF DIRECTIVES

```
REMark $$stak=1024
REMark $$heap=10000,chan=10
REMark $$asmb=mdv1_extensions_code,0,12
```

OPTIMISATION OF ARITHMETIC

There are two compiler directives which turn on or off a space/speed optimisation. They need no parameters.

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 arithmetic is used. Floating point operations on such constants are slightly slower, due to the coercion from integer to float which has to be performed.

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.

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 the size specified.

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

Chapter 5 Compiler Messages

This chapter explains all of the messages which can occur when you are compiling a program with 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, e.g. '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.

Phase one of Q_Liberator produces only two error messages relating to the format of your source program. They are displayed on channel 0. Both are concerned with the structure of procedure or function definitions.

END DEFine error

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

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.

e.g.

```
DEFine PROCedure TEST(x)
IF x=1 THEN END DEFine
PRINT x
END DEFine
```

During phase one Q_Liberator checks that there is only one END DEFine for procedure or function. This is necessary for phase 2 to operate correctly. 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.

This change is made to your source program in memory. It is the only time that Q_Liberator is so bold as to actually change your program for you, because it breaks a fundamental rule. If you LIST such a program after running phase 1, you will see the inserted RETurns.

Thus the above example would become:

```
DEFine PROCedure TEST(x)
IF x=1 THEN RETurn
PRINT x
END DEFine
```

The message 'END DEFine altered' 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. e.g. 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.

The complete list of messages reported during phase 2 is given below:

Warning..END IF without IF

The computer 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.

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 undocumented constructs in JS and later ROMs for error trapping, i.e. 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 (e.g. 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. e.g. 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.

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, recovery from many 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, i.e. 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 on the category of runtime error.

Q_Liberator splits runtime errors into three categories:

- Initialisation errors
- QDOS errors
- Q_LIB 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 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 ought to be loaded by a boot program prior to running an object program, or the program should have its own copy linked to it.

The second thing that a Q_Liberator object program does is check that any extra procedures or functions which it needs are present. If it cannot find them, then it 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, you might see:

JOB : Spacegame ZAP EXPLODE missing!

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. 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 Q_LIB '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, i.e. 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 : 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.

In addition to this standard form of error recovery, QDOS errors can be trapped using Q_ERR error trapping, explained in chapter 11.

Q_LIB ERRORS

Q_LIB 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 messages.

Each Q_LIB 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. Q_LIB 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 of memory organisation fir 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 (e.g. 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 with an expression
24 Occurred when storing data into a variable, e.g. 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 it's 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.

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 the 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 accommodate 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).

Stack

The stack area is a general work area used to store return adresses, 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 (i.e. those that grow or shrink in size during runtime). All strings and arrays and any associated descriptors are stored here. When a 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 bytes at a minimum. To avoid any possibility of common heap fragmentation you should obtain 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, i.e. 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
```

You will be asked for the object name which you want to change (no need to specify _obj). The current parameters are displayed and you can overwrite them if necessary.

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.

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 rigorous 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 the QL User Guide. 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 during phase 1. This change is actually made to the interpreted program in memory; if you SAVE it, you will have a correctly structured version.

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 (e.g. 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.

i.e. `10 FOR x=1 TO 10: PRINT x`

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

are equivalent.

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
```

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 the 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.

e.g.
```
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 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 only of use 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 necessarily 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.

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. 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, i.e. 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 occurrences, 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. e.g. 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+) * (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 money by keeping your channel numbers low.

At compile time, Q_Liberator allocates a similar channel table large enough to accommodate 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 accommodates 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.

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 the 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 is 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 channle 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 permanently 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 the $$asmb directive is:

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

where

FILENAME

is the full name of the module e.g. MDV1_EXTENSIONS_CODE

INIT

is the address in the 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, unpredictable runtime behaviour will result.

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

```
REMark $$asmb=mdv1_qlb_ext_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  chan1         in a0
04A  70F1              moveq  #-15,d0
04C  5303              subq.b #1,d3
04E  6612              bne.s  chan1
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      chan1   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.

e.g. `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, i.e. 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.

e.g. `EXEC_W MDV1_DEMO_SORT_OBJ`

No 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 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.

e.g.

```
QX MDV1_PRINTFILE,"accounts_dataJuly 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.

e.g. `QX mdv1_testprog,#3,#3`

Start testprog using the parent's channel 3 as testprog'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 also 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 present job; the *position* of the parameter determines the channel which is replaced in the daughter job.

An 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 int 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 other end 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.

e.g. `OPEN 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 it's 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.

e.g.

```
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 (e.g. 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 forfeed: title
180   END IF
190 END REPeat page
200 :
210 DEFine PROCedure formfeed
220   PRINT #1,CHR$(12);: P=P+1: L=0
230 END DEFine
240 :
250 DEFine PROCedure title
260   PRINT #1,cmd$,"Page : ";P\\\
270 END DEFine
```

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 crucial 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 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. Consequently, 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 procedures which commonly generate errors, e.g. 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.

e.g.

```
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.

e.g.

```
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 messages.

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 it no parameters are supplied then Q_ERR_OFF will remove all procedures from the error trap list.

e.g.

```
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     =-21: RETurn "bad line"
1270     =REMAINDER : RETurn "error "&e
1280   END SELect
1290 END DEFine
```

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 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 the job is suspended, e.g. 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, i.e. 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, e.g. as a 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.

e.g. `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 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

Appendix A Making a Working Copy

All of the Q_Liberator files with the exception of QLIB_OBJ can be freely copied for your own use. The second phase of the compiler, QLIB_OBJ, can only be copied by creating a new Working copy from the Master. Attempts to copy it by normal means will render the compiler inoperable.

MAKING A WORKING COPY

This is only possible when there are no other jobs running in the system.

Reset the QL, place the Master microdrive in MDV1_ and press F1 or F2 to boot from it.

After a short loading time you should see the Q_Liberator windows containing a display of how many copies can still be made from this Master, and a prompt asking for the target device name. This can be any microdrive, floppy or winchester.

Place a formatted medium in the target drive of your choice. If this is a microdrive, we recommend that it is empty, but in any event there should be room for approximately 100 sectors.

The copy program copies all the QLIB files on to the target disk, including the file QLIB_BOOT. This will have been tailored to suite the target medium. The working copy will require a BOOT program if you intend to load from it. If you wish to use QLIB_BOOT you should copy it (or rename it) to become BOOT. This is not done automatically because there may already be a BOOT program which you want to keep on the target drive.

Remove the Master cartridge as soon as the copy is complete.

Note that Q_Liberator expects to run from the first device of a particular device type, regardless of which number it was copied on. For example, if you make a working copy on MDV2, you must run it in MDV1. The name of the load device is embedded within QLIB_BIN for the LIBERATE command.

We recommend that you use Q_Liberator with the working copy in drive 1 and the programs which you are compiling on drive 2. You may wish to write protect the working copy to prevent accidents.

Note that the Master will not operate correctly if it is write protected, or if ANY of the files on it are changed.

Appendix B File Contents

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_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 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.

This file is optional; it is not required by the compiler. You may choose not to load it by amending the BOOT program.

QLIB_BOOT

This file is already renamed as BOOT on the Working copy which you receive. In its standard form it loads QLIB_BIN, QLIB_RUN and QLIB_EXT. You can create other BOOT programs (e.g. 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.

Appendix C Summary of Syntax

The syntax of all SuperBASIC extensions supplied with Q_Liberator is summarised here. The convention for syntax description is:

[ ] indicates an optional parameter

[ ] ... indicates that the last parameter can be repeated as necessary

LIBERATE

LIBERATE *filename* [, *option_list*]

QX *objectname* [, *command_string*] [, *#channel*] ...

QW *objectname* [, *command_string*] [, *#channel*] ...

QX_JOB0 *objectname* [, *command_string*] [, *#channel*] ...

QJ [*#channel*] [, *jobnumber*]

QJ [*#channel*] [, *jobname*]

QP *jobname*, *priority*

QP *jobnumber*, *priority*

QR *jobname* [, *error_code*]

QR *jobnumber* [, *error_code*]

Q_MYJOB

Q_CURSON [*#channel*]

Q_CURSOFF [*#channel*]

Q_ERR_ON ["*procedure*"] ...

Q_ERR_OFF ["*procedure*"] ...

Q_ERR_LIST

Q_ERR

Q_PIPE *#channel* [TO *#channel*]