RELEASE 3.3 ENHANCEMENTS

INTRODUCTION

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

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

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

NOTES FOR MINERVA USERS

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

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

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

SUPERBASIC CHANGES WITH MINERVA

With the minor exceptions detailed below, all of the enhancements to SuperBASIC described in the Minerva documentation are supported by QLiberator. In some cases where there are bug fixes or obvious enhancements, Q_Liberator was already capable of handling things correctly (e.g. String SELect, FILL$, and RESPR). The TRACE routines TRON, TROFF and SSTEP cannot be compiled — this should not come as a surprise. We have also chosen not to support FOR loops with string variables. If you really think that we should, write to us and let us know. QLiberator will attempt to coerce a string FOR variable to a number. If this is not possible then the runtime system will issue QLIB error 29 — string is not numeric.

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

WHEN HANDLING

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

WHEN ERROR

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

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

For example:

```
  WHEN ERRor
        PRINT ‘something went wrong’: STOP
  END WHEN
```

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

```
  WHEN ERRor: PRINT “Oops!‟
```

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

ENTERING WHEN ERROR

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

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

To be useful, a WHEN ERRor routine needs to be able to determine where the error occurred and what the error was. Then it may be possible to take corrective action or at least print a meaningful message. The ROM contains functions and procedures to support you. ERLIN is a function which returns the line numheratat which an error occurred. ERNUM is *a* function which returns the error number as the usual small negative integer. As an alternative to testing ERNUM, there is a set of functions with names corresponding to the system error codes which return true (=1) if that error occurred. ERR_NF for example, returns true IF *a* 'not found' error haa occurred. The complete list of functions is listed below in the same order as the error codes in the function QERR$ from chapter 11.

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

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

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

For example:

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

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

EXITING WHEN ERROR

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

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

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

```
  e.g. CONTINUE 200      continue from line 200
```

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

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

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

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

Be careful with expressions using ERLIN because explicit line numbers are not automatically adjusted if you RENUMber the program.

TURNING OFF WHEN ERROR

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

WHEN ERROR and Q_ERR

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

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

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

WHEN ERROR IN COMPILED PROGRAMS

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

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

WHEN ERROR AND EXTERNALS

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

WHEN VARIABLE

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

The syntax looks as follows:

```
  WHEN expression
    statements
  END WHEN
```

where expression is usually of the form:

```
  Variable relational operator expression
```

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

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

You can have as many WHEN clauses in a program as you choose, each related to the same or different variables. Changing a watched variable will result in at most one WHEN clause being executed. Thus the order in which WHEN clauses are tested can be significant and depends upon the order in which they are encountered at runtime. Unlike WHEN ERRor which is static and operates on one level only, statements inside one WHEN clause may trigger entry to another WHEN clause. The only restriction is that **it** is NOT possible to re—enter the WHEN clause which is currently being processed. The example overleaf should help to clarify the behaviour of WHEN. It’s worth trying it on your own system.

```
100 WHEN a=1
110   PRINT ‘a=1’,
120   a=0: b=1
130 END WHEN
200 WHEN b=1
210   PRINT ‘b=1’,
220   b=0: a=1
```

```
230 END WHEN
300 WHEN a>0
310   PRINT ‘a>0’,
320 END WHEN
500 a=1
510 PRINT ‘end’
```

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

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

STOPPING WHEN PROCESSING

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

```
  WHEN variable
```

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

WHEN VARIABLE IN COMPILED PROGRAMS

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

MISCELLANEOUS IMPROVEMENTS

TRACE OPTION

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

ERROR CONSOLE

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

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

FREE RUNNING PROCEDURES

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

QLIB_SYS

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

NEW ERROR MESSAGES

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

```
  Error….END WHEN without matching WHEN

  Error….Nested WHEN not allowed
```

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