Perhaps the best understood of these utilities is the BASIC command line interpreter. This is a Job in its own right (a rather special one), but is not an essential part of the QL. Other Jobs can run quite happily in the absence of BASIC. Many of the routines which are used by BASIC are equally applicable to user code. The arithmetic package is a very good example of this. The example in section 8.5.5 illustrates the point very nicely by producing an extra BASIC function to evaluate SINH of a variable.

Other simple routines for serial input and output, number conversion and microdrive handling are just a few out of the wide selection which are documented in the latter parts of this book.

4 Experimenting with QDOS

4.1 Introduction

Most of the other chapters in this book are devoted to describing how QDOS operates. This chapter is different because it is devoted to experimentation both from BASIC and Assembler. If you haven't read chapters 2 and 3, now would be a good time to do so, since this chapter assumes a certain basic understanding of the system and concepts like 'traps'.

Both the QL experimentor program and the examples in this section have been produced to give users an *intuitive feel* for the QL system. The QL Experimentor allows many of the features of QDOS to be accessed. Initially, the Experimentor program will allow traps to be generated from BASIC. It is essential to grasp the concepts behind the use of traps on the QL if other exciting features of QDOS are to be exploited. As we delve deeper into the depths of the computer, it will become apparent that the more advanced features can only sensibly be used from Assembler.

Practical programs written in Assembler are introduced at this stage to illustrate the more important aspects of QDOS, as seen from a machine code program. As a simple introduction, a Job is set up to display the time on the screen. Such a simple program is very useful to analyse because it involves the initialising of a Job, setting up a window, and converting the time and displaying it on the screen. The *self-cloning* Job example then illustrates many of the more complicated aspects of Jobs, including memory allocation and Job control.

There are more specialised examples scattered throughout the book. Chapter 10 contains complete examples for producing a BASIC FUNCTION and a BASIC PROCEDURE in 68000 Assembler.

4.2 BASIC to Machine Code

Before starting on the Experimentor program, several points about using machine code should be noted.

The following SuperBASIC keywords are relevant to machine code programs, but are not properly covered in 'The Sinclair QL User Guide':

CALL

This command allows machine code to be accessed directly from Basic. Seven data registers and six address registers can be passed to a routine using:

```
CALL address, D1,D2,D3,D4,D5,D6,D7,A0,A1,A2,A3,A4,A5
```

This will cause a JSR to be made to 'address'. Data registers D1 to D7 and address registers A0 to A5 can also be supplied as parameters. They need not all be supplied, so

```
CALL 262000,23,4
```

will only set D1=23 and D2=4. Note that D0 is not passed by the CALL command because it is usually used as the error register. However, this does mean that it is not possible to set D0 to any particular trap routine from BASIC. The experimentor program resolves the problem by passing D0 in D4, then transferring D4 to D0 in machine code. On early versions of the ROM, CALL will fail if it is used from a BASIC program occupying more than 32K bytes of memory.

EXEC

EXEC loads and executes a program. The program is given a slice of the processor time, but Basic still remains operational. This illustrates one of the major features of QDOS – the fact that it supports multitasking. Essentially, there is a routine within the operating system called the scheduler. This is normally invoked some 50 times a second by interrupts. It then looks at all the Jobs in the machine and decides which one should be running (in proportion to the Job's priority). The Job which is selected can then operate for the next fiftieth of a second.

For example:

```
EXEC MDV1_prog1
```

will load in prog1 and start it running, timeslicing it with Basic and any other Jobs in the QL.

Note that programs which are to be initialised using EXEC should have been saved using the SEXEC command.

EXEC_W

This is almost identical to EXEC. The only difference is that it will complete all processes before returning to BASIC. If the processes are not designed to return, then Basic will never return.

LBYTES

LBYTES will load a file into memory at a specified start address. For example:

```
LBYTES mdv1_program, start
```

will load in the file 'program' from microdrive 1, the first byte being loaded at address 'start'.

PEEK

PEEK is a function which will return the byte contents of a specified memory location. For example:

```
PRINT PEEK(100)
```

will return the contents of byte 100 (in the ROM).

PEEK_W

PEEK_W is a similar function to PEEK, but returns the word contents of a specified location (ie. 2 consecutive bytes). Note that the specified address must be an even one.

PEEK_L

PEEK_L is a similar function to PEEK, but returns the long word contents of a specified memory location (ie. 4 consecutive

bytes). Note that the specified address must be an even one. This is because all words and long words are aligned on word boundaries. Bytes can sometimes occupy half a word. In such cases, the rest of the word is normally filled with zeros and forgotten.

POKE

The POKE command allows the contents of a byte of memory to be changed. For example:

```
POKE 140000,52
```

will put 52 into memory location 140000. Note that POKEing into ROM will not change the contents of memory, because ROM cannot be altered.

POKE_W

POKE_W is similar to POKE, except that a word is POKEd into memory (ie. 2 bytes). Note that the specified address must be even.

POKE_L

POKE_L is similar to POKE, except that a long word is POKEd into memory (ie. 4 bytes). Note that the specified address must be even.

RESPR

RESPR is a function which allocates memory in the Resident procedures area of memory. It is normally used just after the machine has been booted to allocate memory for any permanent user code. It will not be able to allocate any memory if there is anything in the transient program area. For example:

```
PRINT RESPR (100)
```

will allocate 100 bytes of memory in the Resident procedures area. The returned value is the start address of the allocated memory block.

SBYTES

SBYTES allows areas of the QL's memory to be saved onto a QL device. For example:

```
SBYTES mdv1_screen,131072,32768
```

will store the entire screen contents in the file 'screen' on microdrive 1.

SEXEC

This command allows an area of memory to be saved in a form which is suitable for later execution by the EXEC or EXEC_W commands. It has the format:

```
SEXEC mdv1_prog, start, length, data space
```

where 'prog' is the name of the file stored on microdrive 1, 'start' is the address where the program is currently saved (and where it will be reloaded to), 'length' is the total length of the area and 'data space' is the amount of data storage memory required by the saved program.

VER$

This variable contains the current BASIC interpreter version number, so

```
PRINT VER$
```

will return the version of the ROM contained in the QL.

4.3 The EXPERIMENTOR program

4.3.1 Introduction

This section covers the QL Experimentor program. Because the QL does not have a resident Assembler, it is necessary to access QDOS from BASIC (unless you are fortunate enough to possess an Assembler). This program allows you to start getting inside

the system. It can't take you all the way, but should certainly provide a good start. The most important part of this section is experimenting on your own. The information in the reference sections of chapters 5, 6 and 7 will be found useful for this. Have fun!

4.3.2 The Experimentor program lisiting

```
100 REMark QL Experimentor program
110 REMark
120 REMark by A C Dickens 1984
130 REMark
140 REMark Allocate memory first
150 ad=RESPR(100)
160 screen
170 REPeat main
180 valin
190 trpgen
200 peekout
210 END REPeat main
220 STOP
230 REMark
240 DEFine PROCedure valin
250 REMark VALIN gets input values
260 valout
270 AT 4,7:INPUT a$
280 IF a$<>"" THEN trap=a$
290 AT 4,13:INPUT a$
300 IF a$<>"" THEN D0=hexdec(a$)
310 AT 8,4:INPUT a$
320 IF a$<>"" THEN D1=hexdec(a$)
330 AT 9,4:INPUT a$
340 IF a$<>"" THEN D2=hexdec(a$)
350 AT 10,4:INPUT a$
360 IF a$<>"" THEN D3=hexdec(a$)
370 AT 12,4:INPUT a$
380 IF a$<>"" THEN A0=hexdec(a$)
390 AT 13,4:INPUT a$
400 IF a$<>"" THEN A1=hexdec(a$)
410 AT 14,4:INPUT a$
420 IF a$<>"" THEN A2=hexdec(a$)
430 AT 15,4:INPUT a$
440 IF a$<>"" THEN A3=hexdec(a$)
450 END DEFine
460 REMark
470 DEFine PROCedure valout
480 REMark Printout values of call
490 REMark parameters
```



```
500 AT 4,7:PRINT trap
510 AT 4,13:Dechex(D0)
520 AT 8,4:Dechex(D1)
530 AT 9,4:Dechex(D2)
540 AT 10,4:Dechex(D3)
550 AT 12,4:Dechex(A0)
560 AT 13,4:Dechex(A1)
570 AT 14,4:Dechex(A2)
580 AT 15,4:Dechex(A3)
590 END DEFine
600 REMark
610 DEFine PROCedure peekout
620 REMark PEEK returned values
630 REMark from memory
640 RESTORE
650 DATA 24,8,24,9,24,10
660 DATA 24,12,24,13,24,14,24,15
670 REMark Print error return in decimal
680 AT 17,15:PRINT PEEK_L(ad+16);"  "
690 FOR p=1 TO 7
700 READ xp,yp
710 AT yp,xp:Dechex (PEEK_L(ad+16+4*p))
720 NEXT p
730 END DEFine
740 REMark
750 DEFine PROCedure trpgen
760 REMark Generate required trap
770 init
780 CALL ad,D1,D2,D3,D0,0,0,0,A0,A1,A2,A3
790 END DEFine
800 REMark
810 DEFine PROCedure init
820 REMark POKE the following code
830 REMark into the first part of
840 REMark screen memory
850 REMark MOVE D4,D0
860 REMark TRAP #n
870 REMark MOVEML D0D1D2D3A0A1A2A3,abs.L
880 REMark MOVEQ #0,D0
890 REMark RTS
900 REMark Data POKEd in here
910 POKE_W ad,8196
920 POKE ad+2,78
930 POKE ad+3,(64+trap)
940 POKE_W ad+4,18681
950 POKE_W ad+6,3855
960 POKE_L ad+8,ad+16
970 POKE_W ad+12,28672
980 POKE_W ad+14,20085
990 END DEFine
1000 REMark
1010 DEFine FuNction hexdec(hex$)
```

```
1020 REMark Convert hex to decimal
1030 dec=0
1040 FOR n=1 TO LEN(hex$)
1050 LET v=CODE(hex$(n))
1060 IF v>64 THEN v=v-55
1070 IF v>47 THEN v=v-48
1080 dec=dec+2^((8-n)*4)*v
1090 NEXT n
1100 RETurn dec
1110 END DEFine
1120 REMark
1130 DEFine PROCedure Dechex(dec)
1140 REMark Convert decimal to hex
1150 LOCal x,y
1160 x=dec
1170 c$="00000000"
1180 FOR y=1 TO 8
1190 c=INT(x/(16^(8-y)))
1200 x=x-16^(8-y)*c
1210 IF c<10 THEN
1220 c$(y)=CHR$(48+c)
1230 ELSE
1240 c$(y)=CHR$(55+c)
1250 END IF
1260 NEXT y
1270 hex$=c$
1280 PRINT hex$
1290 END DEFine
1300 REMark
1310 DEFine PROCedure screen
1320 CLS
1330 D0=0
1340 D1=0
1350 D2=0
1360 D3=0
1370 A0=0
1380 A1=0
1390 A2=0
1400 A3=0
1410 trap=1
1420 PAPER 4
1430 CLS
1440 PAPER 7:INK 4
1450 AT 0,0:PRINT " QDOS EXPERIMENTOR v1.0  by ACD 1984 "
1460 PRINT
1470 INK 2
1480 AT 2,1:PRINT "Enter all values as HEX in capitals"
1490 PAPER 1:INK 7
1500 AT 6,1:PRINT"Call Parameters"
1510 AT 6,21:PRINT"Return Values"
1520 AT 4,1:PRINT "Trap #";trap
1530 AT 4,9:PRINT " D0"
```



```
1540 FOR pos = 0 TO 20 STEP 20
1550 AT 8,pos:PRINT " D1"
1560 AT 9,pos:PRINT " D2"
1570 AT 10,pos:PRINT " D3"
1580 AT 12,pos:PRINT " A0"
1590 AT 13,pos:PRINT " A1"
1600 AT 14,pos:PRINT " A2"
1610 AT 15,pos:PRINT " A3"
1620 NEXT pos
1630 AT 17,1:PRINT"Error return: "
1640 AT 18,1:PRINT"(value in D0)"
1650 END DEFine
```

4.3.3 How the Experimentor works

The program runs in a continuous loop. You enter values which should be passed to a trap, the trap is called by the Experimentor, then you enter more values and so on.

The operation of the program is essentially covered under two general titles; initialisation and experimenting.

Initialisation consists of allocating some memory for the machine code subroutine. This subroutine is a short piece of 68000 code to generate a trap and return register values to Basic. RESPR (100) is used to allocate 100 bytes. This is then followed by the procedure *screen*. All of the variables are initialised to their default values and the background screen is printed out.

Experimentation is carried out from within the REPEAT loop *main*. This never ending loop continually cycles through the process of getting the trap parameters for the trap, generating a trap, then printing out the results on the screen.

The following PROCEDUREs are used:

valin is used to get all of the parameter values from the keyboard. Typing ENTER by itself will cause the default value to be used. Otherwise, all 8 digits of the long word parameters should be entered. Note that they must all be entered in hexadecimal (using capital letters only).

valout is used to print the current values of all the defined CALL addresses and data registers into the relevant boxes on the screen.

peekout is used to get the RETURNED values of the data and address registers from the special machine code subroutine (see later), and print them out on the screen.

trpgen actually generates the trap by calling a small machine code subroutine. This subroutine is explained under **init**.

hexdec is a function which converts a hexadecimal number in ASCII string format into a decimal number.

dechex is a procedure which converts the decimal parameter which is passed to it into a hexadecimal ASCII string. This string (8 characters long) is then printed out at the current cursor position.

screen initialises all of the variables and sets up the screen format ready to accept the parameter values.

init initialises the machine code subroutine in memory and then calls it. The space for init was allocated at the start of the program using RESPR. The routine is:

```
        MOVE D4,D0
        TRAP #N
        MOVEM.L D0-D3/A0-A3,DATA
        MOVEQ #0,D0
        RTS
*
DATA    return values go here
```

The MOVE D4,D0 simply transfers the contents of register D4 into register D0. This is necessary because the Basic CALL command does not pass D0.

The TRAP #N generates a trap number N. All of the register values which were entered on the screen are sent to this trap.

The MOVEM.L instruction is a multiple move. It stores all of the return registers D0, D1, D2, D3, A0, A1, A2, A3 into memory starting from DATA. This allows the Basic program to PEEK out the return values so they can be printed on the screen.

The MOVEQ #0,D0 is used to set register D0 to zero. If this were not done, any error returned from the trap would simply get printed on the screen upon return to Basic, causing program execution to cease.

The RTS simply returns back to BASIC.

4.3.4 Experiments

Now that the operation of the Experimentor program has been covered, the fun can start. First of all, the program will have to be typed into the QL and saved on a microdrive. It can then be run.

Assuming that the program was entered correctly, there should now be a display on the screen as illustrated below:

```
QDOS EXPERIMENTOR v1.0  by ACD 1984

Trap #1  D0 00000000

Enter all values as HEX in capitals

Call Parameters      Return Values

D1 00000000          D1
D2 00000000          D2
D3 00000000          D3

A0 00000000          A0
A1 00000000          A1
A2 00000000          A2
A3 00000000          A3

Error return:
(value in D0)
```

The Experimentor is now ready for use. The TRAP number and register contents to be passed to that trap can simply be typed in. Do note that all values must be typed in hexadecimal using upper case letters. It is convenient to use CAPS LOCK to select this. When a complete entry has been typed, press ENTER. If ENTER is typed without providing a new value, the previous value will be assumed.

Chapters 5, 6 and 7 should be used in conjunction with this section, to see exactly what is being done.

Getting information about the system status

As a first example, try using Trap #1 with D0=0. This is a *Manager trap* called MT.INF which returns information about the system. It does not require any parameters to be passed to it, so simply type ENTER until you get to register A3. Typing ENTER again at this point will cause the Trap to be generated. The return values which are printed up on the screen will be similar to those shown below:

```
QDOS EXPERIMENTOR v1.0  by ACD 1984

Trap #1  D0 00000000

Enter all values as HEX in capitals

Call Parameters     Return Values

D1 00000000         D1 00000000
D2 00000000         D2 312E3032
D3 00000000         D3 00000000

A0 00000000         A0 00028000
A1 00000000         A1 00000000
A2 00000000         A2 00000000
A3 00000000         A3 00000000

Error return: 0
(value in D0)
```

Referring to section 5.4, it will be found that the current Job ID is returned in D1, the ASCII version number in D2 and the system variables pointer in A0. BASIC is Job 0 and this program was run from Basic so D1 has been set to 0. The version number is returned in D2. Converting the HEX values into ASCII shows that this particular QL contained QDOS version 1.02 (quite an early one). It may well be different on other QLs. The value in A0 is $28000, which is one byte above the top of the screen in the QL's memory map. The system variables start at this point.

Finally, the error return is zero. This means that there were no errors generated by the trap. When errors are generated, a negative number will appear in this slot (see Appendix G for all of the error codes).

Suspending BASIC

Manager Trap MT.SUSJB is used to suspend a Job for a defined *timeout* period. D0 is set to $08 for this routine. Timeout is in frame periods, so a timeout of $100 (256 in decimal) is just over 5 seconds. This timeout value is passed in register D3. Try generating this Trap. Note that the cursor doesn't re-appear for just over five seconds. This is because Job 0 (BASIC) was suspended for 5 seconds. It can be suspended for much longer periods (up to 10 minutes) or even indefinitely if D3 is $FFFF.

```
QDOS EXPERIMENTOR v1.0  by ACD 1984

Trap #1  D0 00000008

Enter all values as HEX in capitals

Call Parameters     Return Values

D1 00000000         D1 00000000
D2 00000000         D2 00000000
D3 00000100         D3 00000100

A0 00000000         A0 0003DA00
A1 00000000         A1 00000000
A2 00000000         A2 00000000
A3 00000000         A3 00000000

Error return: 0
(value in D0)
```

Outputing a character to the screen

The command channel (the bottom 4 lines of the screen) is defined as channel 0 in BASIC. There is a Trap called IO.SBYTE which will allow characters to be output to a channel. IO.SBYTE is one of the I/O Utilisation Traps (3). The lower byte of D1 contains the Hex code for the character to be printed. Putting $41 in D1 will cause the character 'A' to be printed on the screen (provided that A0 is set to 0 for the channel at the base of the screen).

```
QDOS EXPERIMENTOR v1.0  by ACD 1984

Trap #3  D0 00000005

Enter all values as HEX in capitals

Call Parameters      Return Values

D1 00000041          D1 00000041
D2 00000000          D2 00000000
D3 0000FFFF          D3 0000FFFF

A0 00000000          A0 00000002
A1 00000000          A1 00000000
A2 00000000          A2 00000000
A3 00000000          A3 00000000

Error return: 0
(value in D0)
```

Panning a screen window

The entire contents of a window can be moved sideways by any number of pixels. I/O utilisation Trap SD.PAN is used for this. As with the last I/O Trap, D3 contains the *timeout* and A0 contains the *channel ID*. D1 contains the number of pixels required from the pan. A positive value moves the entire screen right by that number of pixels. The layout below shows a pan to the right of 64 pixels ($40). Negative values will cause the whole window to be shifted to the left.

```
QDOS EXPERIMENTOR v1.0  by ACD 1984

Trap #3  D0 0000001B

Enter all values as HEX in capitals

Call Parameters      Return Values

D1 00000040          D1 000000D8
D2 00000000          D2 00000000
D3 0000FFFF          D3 0000FFFF

A0 00000000          A0 00000000
A1 00000000          A1 00028E36
A2 00000000          A2 00000000
A3 00000000          A3 00000000

Error return: 0
(value in D0)
```


Setting the ink colour

I/O utilisation Trap SD.SETIN will set the colour of the INK used for printing characters.

If it is called with the parameters set up as below, all future character plotting will be in *cyan*. To see this, try calling IO.SBYTE (see above) again. The characters will be printed in cyan.

```
QDOS EXPERIMENTOR v1.0  by ACD 1984

Trap #3  D0 00000029

Enter all values as HEX in capitals

Call Parameters      Return Values

D1 00000005          D1 00000005
D2 00000000          D2 00000000
D3 0000FFFF          D3 0000FFFF

A0 00000000          A0 00000000
A1 00000000          A1 00028E3E
A2 00000000          A2 00000000
A3 00000000          A3 00000000

Error return: 0
(value in D0)
```

More complicated operations

The previous five examples have all been relatively straightforward in their implementation. If you have followed through those examples in conjunction with chapters 5 and 7, you will probably be aware that there are many more Trap calls. A considerable number of them can be called from the Experimentor program, with all of the required values being passed in the registers.

However, there are lots of Traps which require more information. This is normally stored somewhere in memory and is pointed to by one (or several) of the address registers. A typical example of a Trap which requires more information is SD.RECOL (Trap #3 with D0=$26). This Trap will recolour all of a window. Each colour in that window can be assigned a new

colour. The list which defines the new colour for each old colour is pointed to by A1. There is some space after our machine code routine in the resident procedure area which can be used to store these colour bytes.

Since the base address of the subroutine is pointed to by the Basic variable 'ad', and since the space above the first 60 bytes is not used, typing:

```
POKE (ad+60),7
POKE (ad+61),6
POKE (ad+62),5
POKE (ad+63),4
POKE (ad+64),3
POKE (ad+65),2
POKE (ad+66),1
POKE (ad+67),0
dechex(ad+60)
```

will POKE a suitable table of conversion values into memory. The dechex procedure will return the Hexadecimal value for the start of the table. Assuming that this was $3FFCA, the example shown below will recolour channel 0 (the bottom of the screen). Note that you must not RUN the Experimentor again, otherwise it will simply allocate memory in a different place. Simply GOTO 160 so as to avoid some new memory being allocated by RESPR(100).

```
QDOS EXPERIMENTOR v1.0  by ACD 1984

Trap #3  D0 00000026

Enter all values as HEX in capitals

Call Parameters     Return Values

D1 00000000         D1 000000D8
D2 00000000         D2 00000000
D3 00000000         D3 00000000

A0 00000000         A0 00000000
A1 0003FFCA         A1 0003FFCA
A2 00000000         A2 00000000
A3 00000000         A3 00000000

Error return: 0
(value in D0)
```


The ideas presented so far in this chapter can only be extended a little further from Basic. An Assembler is the next requirement if full advantage is to be taken of the powerful QDOS operating system. We will now leave the world of Basic and move into that of Assembler.

4.4 Programming in Assembler

The examples in this section should help to clarify some of the problems and misconceptions which can arise. Each example is complete in its own right, and will run if it is entered into a suitable Assembler.

4.4.1 Real time clock display

This program produces a continuously updated display of the clock time on the screen. In order to understand the program, it is suggested that you read through this explanation, together with the comments in the program. Constant reference should be made to both the Trap and Utility chapters.

```
*
* clock - a clock in the top RHS of window 0 (monitor mode)
*
ME        EQU    -1
MT.SUSJB  EQU    $08
MT.PRIOR  EQU    $0B
MT.RCLCK  EQU    $13
IO.SSTRG  EQU    $07
SD.TAB    EQU    $11
*
UT.SCR    EQU    $C8
CN.DATE   EQU    $EC
*
          BRA.S  CLOCK          branch to clock code
          DS.B   4              and skip along a bit so that
          DC.W   $4AFB          Job flag is in bytes 6 and 7
          DC.W   5              characters in Job name
          DC.W   'Clock'        followed by ASCII characters
*
CLOCK
          SUBA.L A6,A6          clear A6 for CN_DATE now
          MOVEQ  #MT.PRIOR,D0   set priority
          MOVEQ  #ME,D1         ... of this Job
          MOVEQ  #1,D2          ... to 1 (the lowest)
          TRAP   #1
```

```
*
        MOVE.W  UT.SCR,A2       open window for clock output
        LEA     SCR(PC),A1      address of definition
        JSR     (A2)
        MOVE.L  A0,A4           save channel ID
*
CLOCK_LOOP
        MOVEQ   #MT.RCLCK,D0    read the time into D1
        TRAP    #1
*
        LEA     BUF_TOP(PC),A1  use buffer from top down
        MOVE.W  CN.DATE,A2      to convert date into
        JSR     (A2)
*
        MOVEQ   #IO.SSTRG,D0    now send the resulting string
        MOVE.W  (A1)+,D2        of 20 characters
        MOVEQ   #-1,D3          ... with no timeout
        MOVE.L  A4,A0           to our window
        TRAP    #3
*
        MOVEQ   #SD.TAB,D0      now set the cursor at the
        MOVEQ   #0,D1           ... start of the line
        TRAP    #3
*
        MOVEQ   #MT.SUSJB,D0    suspend to reduce IO load
        MOVEQ   #ME,D1
        MOVEQ   #5,D3           for a 10th of a second
        SUBA.L  A1,A1           no flag
        TRAP    #1
*
        BRA.S   CLOCK_LOOP
*
SCR
        DC.B    0               no border
        DC.B    0               no border colour
        DC.B    4               green background
        DC.B    0               black characters
*
* window definition for monitor mode
*
        DC.W    120         pixels wide (20 chrs, all 6 wide)
        DC.W    10          10 pixels high
        DC.W    512-120     120 pixels from the RHS
        DC.W    206         and at top of default window #0
*
* alternative definition for TV mode
*
*       DC.W    240         pixels wide (20 chrs, all 12 wide)
*       DC.W    10          10 pixels high
*       DC.W    480-240     240 pixels from right hand margin
*       DC.W    216         and at top of default window #0
*
```



```
BUFFER
        DS.B    22
BUF_TOP
        END
```

First of all the program sets the Job priority to the lowest possible (1) using MT.PRIOR. Note that the Job ID is passed to MT.PRIOR in D1 as '−1'. This allows MT.PRIOR to change the current Job's priority.

The vectored utility UT.SCR is then used to open up a window on the screen. The window can either be defined for use on a monitor, or a television, depending upon the definition block placed at SCR.

The main part of the routine is contained within the CLOCK_LOOP. The binary value of the clock is read, converted into ASCII and sent to the screen within this loop. MT.RCLK gets the time from the RTC into D1. A 22 byte buffer is set aside for the converted ASCII string to go into. CN.DATE is used to convert the binary date into ASCII. IO.SSTRG then outputs this string to the screen.

Note the use of MT.SUSJB at the end of the loop. This manager trap will suspend the clock Job for a 10th of a second (because there is no point in continually updating a clock which only changes every second). All Jobs which can be suspended for some period of time should be, so that the central processor spends more time on other Jobs which need continuous attention.

4.4.2 The self-cloning program

This program illustrates the full power of multitasking on the QL. The aim of the program is to draw a set of coloured blobs in a line from the centre of the screen to the edge. However, it has one very special feature. After 25 blobs have been drawn, the program *clones* itself. The only difference between the original and new version is that the new one produces a different colour blob which moves in a straight line. Each clone splits until there are lots of separate blobs moving towards the side of the screen!

```
*
* program to set up a self cloning program
*
* The clones draw a line of dots in a direction and colour
* determined by the random number in the OS. This number is
* updated every time it is used.
```

```
*
ME        EQU    -1
MT.CJOB   EQU    $01
MT.RJOB   EQU    $04
MT.FRJOB  EQU    $05
MT.SUSJB  EQU    $08
MT.ACTIV  EQU    $0A
MT.PRIOR  EQU    $0B
SD.FILL   EQU    $2E
UT.SCR    EQU    $C8
SV_RAND   EQU    $2802E
JB_A7     EQU    $5C
*
          BRA.S  MCLONE          start at the master clone
          DS.B   4
          DC.W   $4AFB
          DC.W   6,'MClone'
MCLONE
          LEA    SCR(PC),A1      set up window for all screen
          MOVE.W UT.SCR,A2
          JSR    (A2)
          MOVE.W #254,D4         set starting point in centre
          MOVEQ  #127,D5         ... X = 254, Y = 127
          MOVE.L A0,A4           set A4 for create
          BSR.W  CREATE_JOB
          MOVE.L D1,D6           save new job ID
*
KILL
          MOVEQ  #MT.SUSJB,D0    suspend
          MOVEQ  #ME,D1          me
          MOVEQ  #50,D3          for a second
          SUBA.L A1,A1           no flag
          TRAP   #1
*
          MOVEQ  #MT.RJOB,D0     now try to kill all my jobs
          MOVE.L D6,D1
          TRAP   #1
          TST.L  D0              are they all inactive?
          BNE.S  KILL
*
          MOVEQ  #MT.FRJOB,D0    now kill myself
          MOVEQ  #ME,D1
          MOVEQ  #0,D3           no error
          TRAP   #1              (cannot return)
*
SCR       DC.B   0,0,208,7       no border, white on dark red
          DC.W   512,256,0,0     whole screen
*
* Self cloning program
*
SCLONE
          ADDQ   #2,A7           get call parameters off stack
          MOVE.L (A7)+,A4        first the channel ID
```



```
          ADDQ   #2,A7
          MOVEM.W (A7),D4/D5     then current X,Y coordinates
          MOVE.L #$00040003,-(A7) set block to 4x3 pixels
          MOVE.L A7,A5           block parameters now on stack
*
          MOVE.W SV_RAND,D6      get the system random number
          ROR.W  #2,D6           makes MSB change more quickly
          MOVE.W D6,D7           ... Y increment
          ROR.W  #4,D6           shift again for X increment
          MOVE.W D6,SV_RAND      update random number
*
          MOVEQ  #7,D1           set the block colour
          AND.B  D7,D1           to 0 to 7
*
          EXT.L  D6              make the increments long words
          EXT.L  D7
          ASL.L  #2,D6           up to 4 in most significant
          ASL.L  #2,D7           ... word
          MOVEM.L D6/D7,-(A7)    put the increments on stack
*
          MOVE.L D1,D6           and keep the colour safe
SET_LOOP
          MOVEQ  #24,D7          draw 25 blobs between clones
LOOP
          SWAP   D4              move the X,Y to MSW
          SWAP   D5
          ADD.L  (A7),D4         add the increments
          ADD.L  4(A7),D5
          SWAP   D4              and move them back again
          SWAP   D5
          MOVEM.W D4/D5,4(A5)    set the current X,Y
*
          BSR.S  WBLOB           write blob
          TST.L  D0              was alright?
          BNE.S  DONE            ... no, stop
*
          MOVEQ  #MT.SUSJB,D0    suspend
          MOVEQ  #ME,D1          me
          MOVEQ  #10,D3          for 10 frames
          SUBA.L  A1,A1          no flag
          TRAP   #1
*
          DBRA   D7,LOOP         and do it again
*
          BSR.S  CREATE_JOB      create a clone
          BRA.S  SET_LOOP        and carry on
*
DONE
          MOVEQ  #MT.PRIOR,D0    inactivate
          MOVEQ  #ME,D1          me
          MOVEQ  #0,D2
          TRAP   #1
```

```
*
WBLOB
        MOVEQ   #SD.FILL,D0     fill blob
        MOVE.L  D6,D1           with predefined colour
        MOVEQ   #-1,D3          wait
        MOVE.L  A4,A0           channel ID
        MOVE.L  A5,A1           address of fill block
        TRAP    #3
        RTS
*
* Produce a clone
*
CREATE_JOB
        MOVEQ   #MT.CJOB,D0     create a new job
        MOVEQ   #ME,D1          owned by me
        MOVEQ   #16,D2          enough room for the job name
        MOVEQ   #20,D3          enough room for the stack
        SUB.L   A1,A1           start address 0
        TRAP    #1
        TST.L   D0              did it succeed?
        BNE.S   CR_EXIT         ... no give up for now
*
        MOVE.L  A0,A6           base address of Job
        MOVE.W  #$4EF9,(A6)+    JMP.L
        LEA     SCLONE(PC),A1
        MOVE.L  A1,(A6)+        to SCLONE
        MOVE.W  #$4AFB,(A6)+    put standard job ID in
        MOVE.L  #$00065343,(A6)+ and name of job
        MOVE.L  #$6C6F6E65,(A6)
*
        LEA     36(A0),A6
        MOVEM.W D4/D5,-(A6)     put current position on stack
        MOVE.W  #4,-(A6)        (4 bytes long)
        MOVE.L  A4,-(A6)        and channel ID
        MOVE.W  #1,-(A6)        (just the one)
*
        MOVEQ   #MT.PRIOR,D0    use the set priority call
        MOVEQ   #0,D2           to find the saved stack
        TRAP    #1              ... pointer address
        MOVE.L  A6,JB_A7(A0)    and reset it
*
        MOVEQ   #MT.ACTIV,D0    activate job
        MOVEQ   #1,D2           at priority 1
        MOVEQ   #0,D3
        TRAP    #1
CR_EXIT
        RTS
        END
```


The program can be split into two major parts. The first part called MCLONE (the master clone) sets up the whole of the screen as a window, then starts off a clone in the centre. After waiting for a second, the master looks to see if all of the clones can be killed off yet. It continues to do this every second from this point onwards until all clones have reached the edge of the screen. At this point, it kills itself off as well.

In the second part of the program, the clones are buisily producing more clones. The colour of blobs which are drawn is selected by the random number held in the OS variables workspace at $2802E. Blobs are plotted 25 at a time within the LOOP. When finished, another clone is created. The birth of a cloned program occurs in CREATE_JOB. This uses the manager trap MT.CJOB to create new Jobs.

Jobs are inactivated whenever they reach the edge of the screen. The master then terminates that Job. Throughout the whole process, there is only one (re-entrant) copy of the code. Each subsidiary clone is just data!