QL Today's
QL Assembly Language Programming Series

Book One

Norman Dunbar

PUBLISHED BY MEMYSELFEYE PUBLISHING ;-)

http://qdosmsq.dunbar-it.co.uk/downloads/QLToday/QL_Assembly.pdf

First printing, December 2014
Second printing, June 2015

This pdf document was created on *D:20150717162214+01'00'*.

Contents

IV SuperBasic, QDOS and Other Interesting Stuff. Part 2

List of Tables

List of Figures

Listings

Introduction to Assembly Language

1. QL Assembly Language Programming

1.1 Introduction

Assembly language is very, very simple.

Not many people will agree at first, but if you think about it, it is. You have to tell the processor what you want it to do in very simple steps. In SuperBasic, you can multiply two numbers together easily - you can do it almost as easily in machine code too.

This series of articles is intended to let you in on the basic secrets of programming your QL in its own natural language - machine code or assembly language. (Actually, machine code is what the QL talks, we use assembly language which is 'English' sounding 'words' that get converted to machine code by an assembler - I will tend to use the two terms as one.) To talk directly to the QL, you must learn its language. This series should hopefully teach you how to do just that.

I make no assumptions about how much or how little you may already know - I will start very simple and continue from there. Hopefully you will have an assembler, but if not, see below!

I was going to base the series on George Gwilt's GWASS assembler, which is free and can be distributed easily. Unfortunately, it won't run on anything less than a 68020 which is no good for those of us who are still running on an original QL. George, however, has supplied another of his assemblers, GWASL for use in the series. Thanks George.

Most assembly language books tend to give little example programs as they go along to try to show the bits of the instruction set that you have just learned about. I will attempt to do likewise.

1.2 The 6800x Processor

The processor we are programming is one of Motorola's 68000 series. Be it a 68008 or a 68060 (if you are lucky) all of them have the same basic instruction set, although some of the more powerful processors have additional instructions. Partly because we have to cater for those on an original QL but mostly because I don't have a clue about these additional instructions, we will be dealing with the basic instruction set - there is enough there to keep us happy for a while. Inside the processor there are a few different parts, but we are only concerned with the registers - the rest just does the work and puts the results somewhere, setting a few flags along the way. Talking of flags, we will also take a look at the status register - a very important part of programming.

1.2.1 Registers

Registers are where numbers get loaded into, manipulated and written out from. Some instructions operate directly on memory locations, but to all intents and purposes, memory is just another register but outside of the processor and a lot slower. The 68000 - which is the term I shall use from now on to describe the entire family of processors - has different types of registers - data, address, status and program counter. Data held in registers and in memory is held in High Order format. This simply means that the numbers are stored in a similar manner to the way in which we would expect them to be - the 'rightmost' end holds the most significant bit and the 'leftmost' the lowest - just the way we write numbers down.

1.2.2 Data Registers

There are 8 data registers named D0 to D7 and these can be used to perform manipulations on the numbers that are held in them. Each register can hold 32 bits of information. (A bit is a single binary digit - basically a one or a zero). What these bits actually represent depends on the program running at the time. Data registers are normally used for manipulating data in the form of bytes, words and long words - these being 8, 16 and 32 bits long respectively.

1.2.3 Address Registers

There are 9 address registers named A0 to A7. A7 is sometimes known as the stack pointer or SP register. What about the other address register then? The ninth address register is a duplicate of A7 and is the SSP or Supervisor Stack Pointer. When coding the chip, you only have access to 8 address registers at any one time - you are either using the SP or SSP version of A7 but never both at the same time.

Address registers are normally used to hold memory addresses, stack pointers etc and cannot be used for byte sized manipulations.

1.2.4 Status Register

The status register holds a list of flags to tell the processor what is happening or has happened internally. The status register is a 16 bit register in two 8 bit halves. The user byte is held in bits 0 to 7 (the lowest end) and the system byte is held in the upper half or bits 8 to 15. The layout of the system byte is shown in Figure 1.1.

Bit 15	14	13	12	11	10	9	8
T		S			I	I	I

Figure 1.1: Status Register - System Byte.

- Bit 15, 'T' is the trace flag - this defines whether the processor is in 'single step' mode or running normally. If set to 1, the processor is tracing and if 0, is running normally. In trace

mode the processor 'stops' after each instruction has been executed and jumps to the Trace exception routine. Exceptions are covered later in the series.
- Bit 13, 'S' is the supervisor flag - this defines whether the code being executed is running in user or supervisor mode. If set, the processor is in supervisor mode otherwise it is in user mode.
- Bits 10, 9 and 8, 'III' is the interrupt mask and represents a value between 0 and 7 and indicates which of the seven interrupt levels are enabled.
- The other bits in the system byte are not used.

Bit 7	6	5	4	3	2	1	0
			X	N	Z	V	C

Figure 1.2: Status Register - User Byte.

The user byte contains the 5 condition code flags which are set or reset by certain instructions and then used by arithmetic or comparison instructions. The are used to tell later parts of a program what happened recently. The program can adjust its operations to suit. Figure 1.2 shows the layout of the user byte, the various flag bits are:

- Bit 4, 'X' is the extended flag. Which is very similar to the 'C' flag bit but is affected by fewer instructions than 'C' is. This is used when carrying out very large sized arithmetic instructions - such as 64 bit adds, for example. When affected it is set exactly like the 'C' flag.
- Bit 3, 'N' is the negative flag. It gets set to 1 if the last instruction created a negative number.
- Bit 2, 'Z' is the zero flag and is set to 1 if the last instruction generated a result of zero.
- Bit 1, 'V' is the overflow flag and is set to 1 if the last instruction generated an overflow during 2's complement arithmetic. See later for details.
- Bit 0, 'C' is the carry or borrow flag. And is used when a subtraction operation is carried out - be it an actual subtraction or an implied one.
- The other bits in the user byte are not used.

The flags are used by the branch on condition (`Bcc`) instructions, the Decrement and branch (`DBcc`) instructions or the Set (`Scc`) instructions. These will be explained later.

1.2.5 The Program Counter

The program counter does just that, it keeps track of where exactly the processor is within a program. The program counter always points to the address in memory of the next instruction to be executed. The program counter can of course be changed by a `JMP` (jump) instruction or a `BRA` (branch) but it is always ready with the next instruction to be executed.

1.3 Addressing Modes

The 68000 has a large number of addressing modes and these can often become overwhelming to a new machine code programmer - I know. It takes some time to understand each and every mode, what it does and why it is used. Having said that, you do not need to remember all of their names, just what they look like in source code and of course, what they do.

From here on, you need to be aware that numbers may be in decimal format or hexadecimal. All hexadecimal numbers are prefixed with the dollar sign ($) and wherever this is seen in front of a number (or in some cases, what appears to be a word) will be a hexadecimal number. (I will assume that you are familiar with hex.) A couple of examples of hexadecimal numbers are:

```
$100
$C0FFEE
```

which are equivalent to 256 and 12,648,430 respectively.

Without any further hesitation, lets dive right in with the addressing modes ...

1.3.1 Register Direct

This is an easy one to start off with. Register direct addressing mode simply means that both the source and the destination in the instruction are registers either data, address or a mixture of both.

Simple examples are:

```
        MOVE.L      A2,D1
        MOVE.W      D0,D1
        MOVE.L      A1,A3
```

These simply move (actually, they copy) data between various registers. The full meaning of the actual instructions will be described later on.

1.3.2 Absolute

In this mode, the operand of the instruction is simply a memory address. This is also quite simple. For example to 'zeroise' the contents of the first byte of screen memory (assuming a standard QL and this is the last time that I will assume anything!)

```
        CLR.B       $20000
```

There are two variations to this mode, absolute short and absolute long. If the address given is a 16 bit word (ie 0 to 7FFF hex or 32767 decimal) then it refers to addresses in the first 32K of memory. If the address given is 8000 hex or 32768 decimal and upwards it refers to address FFFF8000 and upwards due to sign extension of the address word. This is absolute short, best used for addresses of 0 to 7FFF hex only - to avoid confusion.

```
        MOVE.L      $1000,D1     get a long from address $1000
        MOVE.L      $9000,D1     get a long from address $FFFF9000
```

The other variation is absolute long, in this case, the address given is a full 32 bits long and refers to the actual address in memory - there is no ambiguity with absolute long. MOVE.L $123456,D1 - gets the long word at address $123456.

1.3.3 Relative

This mode will probably be the most used with QL programs as all code should be relocatable. This means that it never assumes that it is running at a specific location in memory. Some early QL programs were written to run at a specific location in memory and this caused no end of problems when memory expansions became available. I think Psion chess was one of the guilty ones.

However, relative addressing simply means, relative to where the program counter is. The program counter is always pointing at the address of the instruction in memory after the current one. An example of relative addressing is this small loop and the jump back to the start of the loop:

```
Start     MOVEQ #1000,D0
Loop      SUBQ  #1,D0
          BNE.S Loop(PC)
```

This is a small and totally useless fragment of code. The relative address mode is in the BNE.S LOOP(PC) instruction - it says - branch to the label called ‘loop’, relative to where the program counter is currently pointing, if the result of the subtraction was not zero. The jump is specified in the code as a negative number, not the actual address of where the label ‘loop’ is at.

This negative number (in the example above) is how many bytes are to be added to the program counter to get the address of the next instruction to be executed. The jump can be forwards as well as backwards.

Using relative addressing means that the program can be loaded anywhere in memory and still work. If absolute addressing was used, the program would always have to be loaded at the same address if a crash was to be avoided. The example above is the equivalent of the following SuperBasic code:

```
1000 REMark Start
1010 LET D0 = 1000
1020 REMark Loop
1030 LET D0 = D0 - 1
1040 IF D0 <> 0 THEN GOTO (1040 - 10)
```

Note Because of the slightly different way that assembler works, the calculation of the destination line is not quite accurate. When the `BNE.S` instruction is being executed, the program counter is already set to the following instruction. In the SuperBasic example above, the subtraction of 10 from 1040 should really be 20 from 1050. However, it shall remain as above for now.

1.3.4 Address Register Indirect

This mode is called ‘indirect’ because the address register in question is not the operand in the instruction. It simply serves as a pointer to the operand. In an earlier example we cleared out the first byte of screen memory by using absolute addressing like this:

```
        CLR.B     $20000
```

This instruction could have been carried out using address register indirect mode as follows:

```
        MOVEA.L   #$20000,A1
        CLR.B     (A1)
```

All that this is doing is setting address register A1 with the value 131072 (decimal) which is 20000 (hexadecimal). It then clears out the first byte at that address. This is the same as this SuperBasic example:

```
1000 LET A1 = 131072
1020 POKE A1,0
```

The register's name is put in between a pair of brackets to signify that it is the memory address held in the register that will be acted upon and not the register itself.

1.3.5 Register Indirect With Displacement

This mode is similar to the above, except that a displacement is added or subtracted from the address register to give the final address to be operated upon. Using the above example again, we can zeroise the first 4 byes of screen memory as follows:

```
            MOVEA.L     #$20000,A1
            CLR.W       (A1)
            CLR.W       2(A1)
```

This time we use word sized operations, these simply affect 16 bits instead of 8 as with the byte sized operations. The displacement is the number outside of the brackets and it is added to the address registers contents to create the address to be operated upon. The displacement can be any signed number that will fit into 16 bits. (-32768 to +32767)

1.3.6 Register Indirect With Displacement And Index

It's starting to get complicated now. This is another mode where we have an address register and a displacement to consider, but this time we have an index as well. In this case the displacement has been reduced to 8 bits only giving a range of -128 to +127. The format of this addressing mode is:

```
            CLR.W       2(A2,A0.L)
```

The contents of A2 is added to A0 to get the first address then the displacement is added to give the final result. The 16 bits of memory at the final address is cleared out. (Like POKE_W A2 + A0 + 2, 0). In this case the entire 32 bit value of A0 is added to A2, this is indicated by the '.L' after the second register - the index.

If the suffix had been omitted or was '.W' (which is the default if omitted) then the lower 16 bits of A0 would have been used instead of the whole 32. Take note that the 16 bits will be 'sign extended' to a full 32 bits and this can have unpleasant side effects if the value in bit 15 is a 1 as this will cause a negative index to be generated. There will be more on sign extension later.

The first register specified is always treated as 32 bit (.L) and does not require a '.L' suffix - most, if not all, assemblers will reject it anyway.

1.3.7 Register Indirect With Pre Decrement Or Post Increment

These addressing modes are used for stack operations, usually. The format of the pre-decrement instruction is:

```
            MOVE.L     D0,-(A7)
```

And for post-increment it is:

```
            MOVE.L     (A7)+,D0
```

The actions carried out are as follows for pre-decrement: The value in A7 is decremented (reduced) by the size of the data to be stored (byte, word or long) then the contents of D0 are stored at the location pointed to by A7. In SuperBasic this equates to the following code fragment:

```
1000 LET A7 = A7 - 4
1010 POKE_L A7, D0
```

The opposite action takes place with post-increment, as follows: The contents of the memory address pointed to by A7 is copied into D0, then the address held in A7 id incremented by the size of the data just copied (byte, word or long). Again, this equates to:

```
1000 LET D0 = PEEK_L(A7)
1010 Let A7 = A7 + 4
```

1.3.8 Immediate

This is probably the simplest of all the addressing modes. It simply means that the data specifies the address value. For example:

```
            MOVE.L     #100,D0
```

This copies the value of 100 into data register D0. The hash sign indicates that the data is copied directly into the register. Do not get confused between this instruction and:

```
            MOVE.L     100,D0
```

Note that there is no hash. This instruction means load the *contents* of address 100 into register D0. This is not the same! This is a good source of confusion for beginners - I know all about it, and sometimes still make this mistake!

1.4 Coming Up...

In the next chapter, we will take a closer look at some of the actual instructions in the 68000 instruction set.

2. The 6800x Instruction Set

2.1 Introduction

In part one, we learned some really boring stuff. Address modes are not what I would call interesting reading, and I suppose that most of you who are still reading this, would agree.

At this point, however, it gets worse. We are now going to delve into the instruction set of the processor.

2.2 The MOVE Instruction Family

The most common instruction in the entire world, is probably the `MOVE` instruction. It is actually wrongly named as it really does a COPY rather than a MOVE. The format of the `MOVE` instruction is:

```
        MOVE    source, destination
```

or:

```
        MOVE.size    source, destination
```

The data in source is copied to the destination. For example,

```
        MOVE    D0, D1
```

takes whatever data is in data register 0 (zero) and copies it into data register 1. How much data is moved? In this case, because no size is specified, the default size is always WORD so a single word of data is moved from D0 to D1. As there is space for 2 words in each of these registers, which word is moved from D0 to which word in D1?

All instructions work from the 'lowest' end of the register towards the highest (with the exception of `MOVEP` - see below). So, in the above example, the lowest 16 bits of D0 are copied to the lowest 16 bits of D1. The data in D0 is not altered in any way whatsoever. The same cannot be said for D1 as the original data in D1 has been replaced - but only the lowest 16 bits. The highest word has not been altered.

If D0 contained $01020304 and D1 contained $11223344 then after the above move, D0 would be unchanged and D1 would contain $1122*0304*.

If the size of the instruction had been specified, as follows:

```
        MOVE.B   D0,D1
```

Then only the lowest byte of D1 would have been altered. In this case D1 would have contained $112233*04* after the move. If the size specifier had been 'L' for LONG than the entire 4 bytes in D1 would have been overwritten by the 4 bytes from D0. After a long sized `MOVE`, both D0 and D1 would contain $01020304.

Because the move takes place into a data register the condition codes are affected. To copy data into an address register use the `MOVEA` instruction, but always remember that it does not affect the flags in the condition code register.

The changes that will take place every time a data register or memory location is used as the destination for a `MOVE` are:

- X flag is never affected. It remains as it is.
- N flag is set if the data moved was negative. If the data was positive, N is cleared.
- V is always cleared. You cannot move a value into a register that causes an overflow.
- C is always cleared for similar reasons.
- Z is set if the data moved was zero. It is cleared if it was any other value.

The `MOVE` instruction has many variations, most of them simple and easy to understand. These are:

`MOVE` as described above.

`MOVE CCR` - the size is always word although the upper 8 bits are ignored - effectively a byte sized move. The format of the instruction is :

```
        MOVE     source,CCR
```

Executing this instruction results in the condition codes being set as follows:

- X is set to bit 4 of source
- N is set to bit 3 of source
- Z is set to bit 2 of source
- V is set to bit 1 of source
- C is set to bit 0 of source

All the other bits are simply ignored.

`MOVE SR` - the size is always word and may not be specified in the instruction. This instruction copied the 16 bits of the condition code register to the destination. The instruction format is:

```
        MOVE    SR,destination
```

When the instruction has been carried out, the lower 16 bits of the destination contain a copy of the Status Register of the processor. The actual data in the status register is unaffected by the move.

There is a complimentary instruction to move data into the status register which is:

```
        MOVE    source,SR
```

Which takes the lower 16 bits of the source data and copies it into the status register. The lower 8 bits are used to change the flags in the CCR or Condition Codes Register (See `MOVE CCR` above). The SR is affected according to the lower 16 bits of the source data as follows:

- T is set to bit 15 of source
- S is set to bit 13 of source
- III is set to bits 10, 9 and 8 of source
- X is set to bit 4 of source
- N is set to bit 3 of source
- Z is set to bit 2 of source
- V is set to bit 1 of source
- C is set to bit 0 of source

The other bits are simply ignored. There is a slight problem, the instruction MOVE source,SR must be executed in Privileged mode or it will cause a 'Privilege Violation Exception' which on a normal QL will simply lock it up. (Exceptions are covered later on in the series.)

Warn

Note: on the 68010 and up, the MOVE SR,destination becomes a privileged instruction. There is a new instruction MOVE CCR,destination which allows access to the CCR part of the SR. Programs written for the 68000 and 68008 may require to be re-written with this in mind.

`MOVE USP` - A long sized instruction which copies data into the User Stack Pointer (USP) also knows as A7. This instruction is also privileged and requires that the system is running in supervisor mode. The format of the instruction is:

```
        MOVE    source,USP
        MOVE    USP,destination
```

Both source and destination must be an address register. None of the condition codes are affected by this instruction.

Why does this have to be run in supervisor mode? Well, if not, a privilege violation exception will be generated and these instructions allow the operating system to set the value of a job's stack pointer.

If you remember, there are two A7 registers, one used for supervisor mode and the other for user mode. Only one can be in use at any one time. This instruction allows the supervisor to set the USP without affecting its own version of the A7 register. Not used much, if at all on the QL.

`MOVEA` - the contents (remember that word!) of the source is moved into an address register. This instruction is either word or long sized and does not affect the condition codes. The format is:

```
MOVEA.size    source,An
```

Beware because if you move a word sized source, it will be sign extended to long (bit 15 will be copied into bits 16 to 31) before the data is copied into the address register.

For example:

```
MOVEA.W    #$0001,A0
```

This will set A0 to $00000001 after the move. Bit 15 of the data is a zero so this is copied into all the upper 16 bits of A0. The lower 16 bits are simply a direct copy of the data.

```
MOVEA.W    #$8000,A0
```

This will set A0 to $FFFF8000 after the move. Bit 15 is a one and this is copied into all the upper 16 bits of A0. The lower 16 are again a copy of the data.

Don't forget about sign extension!

`MOVEM` - a word or long sized instruction which allows you to copy data to or from a number of registers in a single instruction. The format of the instruction is:

```
MOVEM    register_list,destination
MOVEM    source,register_list
```

None of the condition codes are affected by this instruction.

The instruction is most often used to store a number of registers on the stack on entry to a subroutine, and to reinstate the original values on exit from the subroutine. The instruction stores the registers starting with D0, then D1 and so on up to D7, then the address registers are stored in order from A0 to A7 - assuming all registers are specified.

A register list takes the format of a starting register name, a hyphen then a finish register name. Another form is a start register name a slash and another register name. The two formats can be mixed to give almost endless possibilities. The following are all register list examples :

```
D1–D4
A0–A3
D1/D4–D7
D0–D2/D4/D7/A0–A3/A6
```

The hyphen means that all registers from the starting one to the finish one (inclusive) will be moved to the destination. The slash signals that there is a 'gap' in the register list. The above examples mean :

```
D1 and D2 and D3 and D4
A0 and A1 and A2 and A3
D1 and D4 and D5 and D6 and D7
D0 and D1 and D2 and D4 and D7 and A0 and A1 and A2 and A3 and A6.
```

The list can be specified in any order (unless the assembler rules differently) as each register detected is used to set a single bit in a 16 bit word. This word is used by the processor to determine which of the registers are to be copied.

This instruction will be most often used in its Post decrement and pre-increment forms:

```
        MOVEM.L     D0–D3,-(A7)
        MOVEM.L     (A7)+,D0–D3
```

MOVEP - Probably the strangest instruction in the 68000 set. This instruction transfers data from a data register to alternating bytes in memory. The data is transferred from the data register starting from the highest 8 bits, then the next 8 bits and so on. This is a word or long sized instruction. The condition code flags are not affected. (I have never used or seen this instruction used on the QL.) The formats are :

```
        MOVEP.size      Dn,displacement(An)
        MOVEP.size      displacement(An),Dn
```

The size is long or word, Dn is any data register, An is any address register and the displacement is added to the address register to get the first address to be filled with data. An example might make things clearer. If we assume that D0 holds $11223344 and A1 holds the address $000200000 then the instruction:

```
        MOVEP.L     D0,0(A1)
```

Copies the highest byte of D0 ($11) into address $20000, the next highest ($22) into address $20002, the next byte ($33) into address $20004 and finally the lowest byte of D0 ($44) into address $20006. Addresses $20001, $20003 and $20005 are not affected.

Had the displacement and A1 combined created an odd address then the odd addresses would have been filled with data and the even ones would not have been affected.

MOVEQ - This is a very useful instruction and you will see it used on many occasions in QL assembly language programs. It is the 'Move Quick' instruction and is used to quickly move any value between -128 and 127 into any data register. The value is sign extended to 32 bits or long sized and so fills the entire data register. The format is:

```
        MOVEQ       #data,Dn
```

The flags are affected by this instruction as follows:

- X flag is never affected. It remains as it is.
- N flag is set if the data moved was negative. If the data was positive, N is cleared.
- V is always cleared. You cannot move a value into a register that causes an overflow.
- C is always cleared for similar reasons.
- Z is set if the data moved was zero. It is cleared if it was any other value.

Remember, only 8 bit values are allowed and these must be between -128 and 127.

A number of 68000 instructions have this 'quick' mode, but why is it quick? Let us compare the MOVEQ \#0,D0 with its equivalent MOVE.L \#0,D0. We simply see two different forms of what is effectively the same instruction, the QL's processor sees things a bit differently, as follows :

First MOVEQ \#0,D0 is a 16 bit instruction in memory. MOVE.L \#0.D0 is also a 16 bit instruction but it is followed in memory by a long word (32 bit) holding the data, in this case zero. This makes the `MOVEQ` instruction 3 times smaller than the `MOVE.L` one. As the processor has less data to fetch from memory, it takes less time to read the instruction and its data, therefore it is quicker. Looking at the 68008 timing chart, it takes the `MOVEQ` instruction 8 clock cycles to execute and the `MOVE.L` 24 clock cycles.

And that is about it for the 68008's `MOVE` instructions. This is probably the instruction with the most variants and as I said before, probably the most used instruction in any program.

2.2.1 Exercise

1. Write down the correct instruction which will copy 4 bytes of data from address $20000 into data register D7.

2. What is the fastest way to get the 8 bit value of 17 into all 32 bits of register D2?

3. What instruction would you use to copy the lowest 16 bits of register D1 into the lowest 16 bits of register D3? What happens to the data in D1 after the move and what happens to the data that is currently held in D3?

4. How would you place the lowest byte of D1 into a memory location which is 10 bytes further on from the address currently held in A0?

5. Why is the `MOVE` instruction 'wrongly' named?

6. What does a privileged instruction require before it can be executed?

7. What happens if a privileged instruction is executed in user mode?

8. How many data registers does the 68008 have and how many address registers?

9. What values are set in each of the condition codes when the instruction MOVEQ #0,D1 is executed?

10. What values are set if the instruction executed was MOVEA.L #0,A0?

2.2.2 Answers

1. MOVE.L $20000,D7

2. MOVEQ #17,D2 or MOVEQ #$11,D2

3. MOVE.W D1,D3. Nothing happens to the data in D1. The highest word on D3 is not affected but the lower word is overwritten by the lowest word from D1.

4. MOVE.B D1,10(A0) or MOVE.B D1,$0A(A0).

5. The MOVE instruction actually copies data from source to destination, it does not move it in the traditional sense of 'it was over there but it has been moved to over there'.

6. The processor must be in supervisor mode.

7. A privilege exception will be generated (and the QL will probably hang).

8. There are 8 data registers and 9 address registers but only one of the A7 'twins' can be used at a time.

9. The Z flag is set to one and all the rest are reset to zero except the X flag which is unaffected and

keeps its previous value.

10. No flags are changed. They all keep their previous values.

2.3 The CMP Instruction Family

While all this talk of moving data around, be it in memory or within the processor's internal registers, is 'interesting', being able to move data is not much use if you cannot do anything with it when you have moved it. As the condition codes are affected by data movements we can sometimes determine the value of the data we moved. This is of course true only if we want to know if the value we moved was zero, or not zero, positive or negative but that's about as accurate as we can get using the `MOVE` instruction.

If we need to compare two values we will need to use the `CMP` family of instructions. `CMP` stands for 'Compare' and allows data to be compared against specific values, registers or memory contents.

The general format of the `CMP` instruction is:

```
        CMP.size    source,destination
```

The `CMP` instruction has the effect of carrying out a subtraction of source from destination without changing the destination at all. What it does change is the condition codes, and these will be set as follows :

- X flag is never affected. It remains as it is.
- N flag is set if the result was negative. If the result was positive, N is cleared.
- V is set if the result caused an overflow otherwise cleared.
- C is set if a 'borrow' was generated and cleared otherwise.
- Z is set if the result was zero. It is cleared if it was any other value.

This instruction can be carried out in all three sizes - byte, word or long.

One of the common uses of this instruction, and perhaps the easiest to understand, is testing to see whether two values are the same. If they are then the result of the 'subtraction' of source from destination will always be zero. If the result is zero then the Z flag can be tested (somehow - we shall see later) and then some actions taken if it is set while others can be taken if it is not set.

The instruction:

```
        CMP.L    D1,D2
```

Will set the Z flag if the same value is present in both D1 and D2. If they are different, then the Z flag will not be set.

There are only four variations of the `CMP` instruction - unlike `MOVE` which has a few more. The first is simply `CMP` itself. This is used when comparing with a data register as in the above example. The source, however, can be any of the 68000 addressing modes - although you cannot compare an address register and a data register using the BYTE size. This means that:

```
        CMP.W    A0,D2
```

is a legal instruction, but that:

```
        CMP.B    A0,D2
```

Is not. It is of course allowed that the data be POINTED to by an address register, as in:

```
        CMP.B    0(A0),D2
```

Which compares the byte of data at the address held in A0 with the byte of data held in the lowest byte of register D2.

`CMPA` - is the form of the instruction used when comparing against a destination which is an address register. It is very similar to the `CMP` variation, but only word and long sized comparisons can be made. If the word size is used, then watch out for the old favourite pitfall of sign extension. Whatever word sized data is used for the source of this comparison will be sign extended up to a long word and then compared with the entire 32 bits of the address register.

This means that:

```
        CMPA.W    #$FFFF,A3
```

Would set the Z flag if and only if A3 contained the value of $FFFFFFFF but would not set it if A3 contained the value $0000FFFF. Beware. If at all possible, make your code explicit. So if you want to test A3 as having $FFFF in its lower word, use CMPA.L \#\$FFFF,A3 instead of the word sized version.

`CMPI` - is the third variation and this one is used when testing any address mode destination (except PC relative or an address register's contents) against source data which is, quite simply, a number. This variation can be used in all 3 sizes. The format of the instruction is :

```
        CMPI.size    #data,destination
```

If the destination is a data register, then the instruction is equivalent to the `CMP` instruction.

`CMPM` - is the final variation. It is used to compare one memory location with another. It can be used in all 3 sizes but can only be used in a single address mode - address register with post-increment. The format is always:

```
        CMPM.size    (An)+,(An)+
```

The two address registers are pointers to the memory addresses to be compared and after this instruction, the flags have been set according to the result of the 'subtraction' while both address registers have been incremented by 1, 2 or 4 depending upon the size of the data being compared.

2.4 Signed and Unsigned Numbers

Before we take a closer look at the condition codes and how we can use them to alter the flow of a program - that is, how we can implement loops, if then else etc, we need to take a break and discuss the differences between signed and unsigned numbers.

When we `MOVE` some data into a data register the same number can actually mean two different things. Confused? You will be!

If we use an 8 bit number as an example, the data $FF can either mean 255 or minus one. In a 16 bit example, $FFFF can mean either 65535 or -1 and in a 32 bit long word, $FFFFFFFF means either $2^{32}-1$ or -1.

The important thing to remember is that it is *you*, the programmer, who decides which version is in use at any particular time.

Ok, how does it work? The 68000 family of processors can use signed or unsigned numbers. If the signed version is in use then the number will be either negative (less than zero) or positive (zero or greater). If unsigned numbers are being used then the value will always be positive. How can the processor tell the difference?

The answer to the question 'is this number signed or unsigned?' is either 'yes' or 'no' equivalent to one or zero in binary terms. This implies that a single bit can be used to hold the sign of the number and this is exactly how it happens. By convention the most significant bit of the number holds the sign. A one indicates that the number is negative while a zero indicated that it is not.

Those of you who are thinking ahead of me now might well be saying 'but surely using a single bit of the register will reduce the amount of numbers that can be represented by a factor of two?'. Not quite.

In binary, the numbers representing the hexadecimal values $00 to $0F will all fit into a half byte or nibble. A nibble is 4 bits and each bit represent a single power of two in the number.

Just as 1231 means $(1*10^3)+(2*10^2)+(3*10^1)+(1*10^0)$, which is, $(1*10*10*10)+(2*10*10)+(3*10)+(1*1)$ which is, $1000+200+30+1$ which is the number we have at the start of all this, the same is true in binary.

The binary nibble 1010 is $(1*2^3)+(0*2^2)+(1*2^1)+(0*2^0)$, which is $(1*2*2*2)+(0*2*2)+(1*2)+(0*1)$, which is $8+0+2+0$, which is 10 in decimal with converts to $0A in hexadecimal.

All the possible values that can be held in an unsigned nibble are 0000 (zero) up to 1111 (15 or $0F) and conversion is a matter of adding up each power of two in the number. From the right we have 2^0 which is simply one. Then 2^1 or two and so on.

In an unsigned nibble the most significant bit (2^3) is used to hold the sign, so all numbers below unsigned 7 are positive while those 'above' 7 are actually negative and so are below 0.

If the highest bit was not the sign bit it would represent 2^3 or 8. To convert into a signed value simply negate the 8 to get minus 8, and add all the other bit values to it. Taking the same binary example of 1010 as above, this is now $(-1*2^3)+(0*2^2)+(1*2^1)+(0*2^0)$. This gives minus 8 plus 2 which is minus 6. This implies that for a signed number the range is minus 8 to plus 7 which is still a possible 16 values as with the unsigned version, just shifted slightly down the number scale.

That is the only difference between signed and unsigned numbers. The ranges of values in a byte are minus 128 to plus 127, in a word it is minus 32768 to plus 32767 and for a long word it is minus 2147483648 to plus 2147483647.

When dealing with signed numbers any number which has a 8, 9, A, B, C, D, E or F in the most significant digit (hex that is) is negative. All the rest are positive. I find the quickest way to find the equivalent negative value is to subtract from 2^{bits}. For example -1 in an 8 bit byte is 2^8-1 which is $256-1$ which is 255. 255 in hex is $FF which is the 8 bit representation of -1. Similarly, -10

is 256 – 10 = 146 which is $F6. Use 65,536 for 16 bit words and 4,294,967,296 for 32 bit long words.

Enough for now. Just remember when coding a program in assembler that numbers can be two different values at the same time. You determine which one is appropriate at any one time. It is far easier to consider unsigned numbers all the time but this might not be applicable. Writing a program to record the number of sheep jumping over a fence need never use signed numbers, while the amount of money in your bank account probably will. Just remember to be consistent.

2.5 Testing Condition Codes and Branching

As you may remember when data is `MOVE`d into a *data* register or memory address, certain condition codes are set or unset. These codes can be used, along with the results of a `CMP` instruction and/or the discussion of signed and unsigned numbers above, to determine program flow. To change the flow, we use the branch instruction also known as `Bcc` or Branch on condition code. The general format of a `Bcc` instruction is:

```
        Bcc     label
```

The label part defines where the branch will be to (the destination) and is an offset from the current program counter and of course may be positive or negative.

A branch instruction is equivalent to a SuperBasic GOTO command. Much frowned upon by purists, but useful in certain situations. Never say 'Never use a GOTO' because in assembly language you almost always have one!

There are a number of 'branch' instructions that look at the condition codes and change the course of your program according to what they find. There are 14 of these and some appear remarkably similar to others. They are listed in Table 2.1:

There is one more branch instruction that does not care about the flags, this is the `BRA` or Branch unconditionally instruction. It is the most like a GOTO instruction as that is its exact purpose - goto some other place in the program.

If the displacement value will fit into a single byte (-128 to +127) then a 'short' branch will take place. This entire instruction fits into a single word. If the displacement is zero, then this would normally indicate a short branch to the next instruction in the program. As this is where the PC is pointing anyway the zero displacement is used to signify a long branch and the word following is used as a 16 bit displacement allowing relative values between -32768 to +32767.

The short branch is written as `Bcc.S` with the dot and 's' indicating the shortness. Most assemblers default to the long branch which adds 2 bytes to your program for every `Bcc` instruction in it. I find the 'best' way to reduce the 'wasted' bytes is to make all branches short and the assembler will reject those which are out of range.

One of the most confusing aspects of assembly language programming for new and experienced coders alike is 'which are the signed and unsigned tests?' I always have to look it up and I have never found a place where all the tests are listed together with the signed and unsigned comparisons. You won't have this problem as I have listed them all in Table 19.1.

In the above description of the `Bcc` instructions I state, for example, that the `BNE` instruction will branch if the last result was not zero. This is not quite the case. If I had just loaded a data register

Branch	Name	Signed/unsigned	Description
BCC	Branch Carry Clear	Unsigned	The branch is executed if the carry flag is not set - ie zero.
BCS	Branch Carry Set	Unsigned	The branch is executed if the carry flag is set - ie one.
BEQ	Branch Equal	Both	Branch only if the result of the last operation caused the zero flag to be set. MOVEQ #0,D0 for example.
BGE	Branch Greater or Equal	Signed	Branch if the last operation resulted in a signed number that was zero or greater.
BGT	Branch Greater Than	Signed	Branch if the last result was greater that zero.
BHI	Branch Higher	Unsigned	Branch if the last result was greater than zero.
BLE	Branch Less or Equal	Signed	Branch if the last result was zero or less.
BLS	Branch Lower or Same	Unsigned	Same as BLE, but 'equal' replaced by 'same'.
BLT	Branch Less Than	Signed	Branch only if the last result was less than zero.
BMI	Branch Minus	Signed	Branch if the result of the last operation was negative. Ie less than zero but not including zero.
BNE	Branch Not Equal	Both	Branch if the last operation resulted in a non-zero outcome. CMPI.L #1,D1 if D1.L is not holding the value 1.
BPL	Branch Plus	Signed	Branch if the result of the last operation is positive ie zero or greater.
BVC	Branch oVerflow Clear	Both	Branch if the last operation left the V flag unset.
BVS	Branch oVerflow Set	Both	Branch if the last operation left the V flag set.

Table 2.1: Branch on condition instructions.

Test for	Signed	Unsigned
Greater Equal	BGE	BCC
Greater than	BGT	BHI
Equal	BEQ	BEQ
Not Equal	BNE	BNE
Less Equal	BLE	BLS
Less than	BLT	BCS
Negative	BMI	Not applicable
Positive	BPL	Not applicable

Table 2.2: Signed & Unsigned Tests.

with some value which was not zero then the branch would be taken, as in the following fragment of code:

```
        MOVE.L    (A0),D1
        BNE.S     Somewhere
```

If, on the other hand, I was comparing two registers then the branch would have been taken if they did not have exactly the same contents :

```
        CMP.L     D3,D4
        BNE.S     not_equal
        BHI.S     greater
```

So you can see that there are more ways to use these conditional branches. Bear in mind, however, that the `CMP` is simply a subtraction with the answer 'thrown away' and it is that discarded result that is being checked. One other area of confusion is which register is greater in the `BHI` instruction above?

In a `CMP` instruction it should be read as Destination `CMP` source. If this is followed by a `Bcc` then it means branch if the destination is *condition* source. So in the above code fragment, we will branch to the label 'greater' if and only if D4 is greater than D3.

There are other instructions that affect the flow of a program and these are the 'looping' constructs or `DBcc` as they are written. These are the 'Decrement and branch *until* condition. Confused? All will be revealed in the next chapter.

2.6 Coming Up...

In the next chapter we will take a closer look at some more branching instructions and start thinking about the project[1].

[1] The project was to be a disassembler named "QLTdis", but it fell dramatically by the wayside and is not discussed further in this book.

3. The 6800x Instruction Set - continued

3.1 Introduction

The preceding chapter started off our great expedition into the various instructions used by the 6800x processor. In this chapter we continue in the same vein. There are still quite a few instructions to cover.

3.2 More Branches.

At the end of part 2, I left you with a promise that the `DBcc` instructions would be explained in this part, but just before we do that, there is the `BSR` instruction. This means ‘Branch to Sub-Routine’ and acts very much like GOSUB in SuperBasic (an instruction I have never used in SuperBasic, but use almost in every program in assembler - strange that.)

`BSR` comes in 2 sizes - byte and word. The format is:

```
            BSR.S       label
```

or

```
            BSR         label
```

Label is the destination of the subroutine to be executed. `BSR` is a PC relative instruction in that the destination is relative to the program counter - although it does not really look it.

The size of the instruction, byte or word, defines the size of the displacement from the PC of the *following instruction* to the address of label. This displacement is added to the PC and the next instruction executed is the one at that address (or PC + displacement). As the displacement is signed, the byte sized `BSR` can ‘gosub’ -128 to +127 bytes from the PC while the word sized `BSR` can 'gosub' -32,768 to +32,767 bytes from the PC. Although the resulting address must, of course, be even.

At this point, a small example will maybe make things a bit clearer. Consider this chunk of (useless) code. It serves no useful purpose apart from showing the use of `BSR` (and a few of the other instructions we have already discussed.).

Read through the following code and at the end I shall explain what it is doing. The only instruction not yet explained is `RTS` which for now simply means 'Return To Sender' - similar to RETURN or END DEF (sort of) in SuperBasic.

```
Start    MOVEQ    #0,D1

Again    BSR.S    Addon
         CMPI.L   #10,D1
         BNE.S    Again
         MOVEQ    #0,D0
         RTS

Addon    ADDQ.L   #1,D1
         RTS
```

Listing 3.1: BSR Example

The code starts by setting D1 to zero in all 32 bits - it is a long sized move. The label 'Start' simply identifies the start of the code fragment and need not be called start - it could be called fred. It acts like a line number in SuperBasic.

The second line of code calls a sub-routine called 'addon' which lives only a few bytes further on - for this reason the byte sized variant of `BSR` is used and this makes the program smaller and slightly quicker - as explained later. Had the distance to the sub-routine been more than 127 bytes (or less than -128) then the assembler would have complained and the source would have had to have been amended to remove the '.s' from the instruction.

The second line also has a label - 'Again'. Labels are used in assembler programs to mark significant places in the code. In SuperBasic every line must have a number - in assembler only those referenced in the code need have one, but there is no problem putting labels where it makes the code more readable.

Following on, there is a check to see if the value in D1.L is 10 (decimal) followed by a branch if not equal zero (`BNE.S`) to the label 'Again'. If the value in D1 is not 10 the Zero flag will not have been set and so the code will start executing from the label 'Again'. If D1.L does equal 10 then the branch to 'Again' will be ignored.

The next line sets D0.L to zero. This is because any code that runs on a QL either as a result of a CALL address or EXECing[1] a file returns any error codes to QDOS in D0.L and zero shows that no error has taken place. All this will be explained in a later article.

The `RTS` instruction ends a subroutine and means return to where you came from (almost). If the above code - beginning at 'Start' was called from SuperBasic, the `RTS` would return us to SuperBasic. If it was called from some other part of the assembler program, it would return us to the next instruction in that program.

The subroutine called from the second line begins at the label 'Addon'. It is very simply and adds 1

[1] Actually, only CALL and EXEC_W take any notice of the error code in D0 when returning to SuperBasic. Jobs executed with EXEC have no effect when they exit with D0 set to a non-zero value. More on this later in the series.

to the value in D1.L before the RTS returns to the place where it was called from.

Put simply. The code above loops around adding 1 to D1.L until such time as D1.L equals 10. At this point the code returns to wherever it was called from.

This is not quite true. The RTS instruction returns back to the instruction that follows the BSR one. So the above code returns to execute the CMPI.L \#10,D1 instruction after running the code in the 'Addon' subroutine.

Now that we have a few more instructions under our belts, there will be more bits of code appearing in the rest of the series. This allows the reader to alleviate the boredom of these articles and allows me to illustrate some examples of what I am trying to say!

```
For D0 = 10 to -1 step -1 ...
```

Looks a bit like SuperBasic that, but you can do the very same in assembler as well. The above code illustrating the BSR instruction can be rewritten to use the DBcc or 'Decrement and Branch' instructions. These are very similar to the Bcc instructions from part 2 of the series but they have an additional purpose. They allow a loop to be executed a set number of times and also can cause an exit from the loop if a certain condition occurs while executing the loop.

It might be better if these instructions were called DBUcc as in 'Decrement and Branch *Until* condition' because that is actually what they do. The full set of DBcc instructions is described in Table 3.1.

Mnemonic	**Branch Until Condition**
DBCC	Carry clear
DBCS	Carry set
DBEQ	Zero flag set
DBF (or DBRA)	Branch false or always
DBGE	Greater or equal
DBGT	Greater than
DBHI	Higher
DBLE	Less or equal
DBLS	Lower or same
DBLT	Less than
DBMI	Minus
DBNE	Not equal (zero flag not set)
DBPL	Plus
DBT	True. Very strange instruction, see below
DBVC	Overflow clear
DBVS	Overflow set

Table 3.1: Decrement and branch instructions.

The format of the instruction is:

```
        DBcc     Dn,label
```

The counter is always a data register, D0 to D7, and only the lowest word is affected. The label is specified as a 16 bit displacement from the PC to the next instruction to be executed. The displacement is, as usual, signed allowing branches of between -32,767 and +32,768 bytes.

This instruction does not affect the condition codes. They remain the same as they were before the instruction.

The operation of the instruction is in three parts:

- First, the condition is tested to determine if the termination condition of the loop has been detected. This is the cc part. So a `DBCS` checks to see if carry is set. If the condition is detected, no branch will be performed and no decrement of the data register will be carried out either.
- Second, if the condition is not detected, the lowest 16 bits of the data register is decremented by 1. If this results in a value of -1, then the loop is also terminated and no branch takes place.
- Third, the branch is taken to the label specified. (PC relative).

Another example:

```
Start    MOVEQ      #1000,D1
         MOVEQ      #0,D2
Loop     ADDQ.L     #1,D2
         CMPI.L     #100,D2
         DBNE       D1,Loop

More     ; More code here ...
```

Listing 3.2: DBNE Example

D1.L is initialised with 1,000 and D2.L is set to zero. Then the start of the loop (at label 'Loop') where 1 is added to D2.L. Following the addition, D2 is checked to see if it equals 100. The `DBNE` instruction checks the zero flag and if not set - therefore D2 is not equal 100 - subtracts 1 from D1 and if this does not result in D1 becoming -1, branches to the label 'Loop' to go round again.

At the label 'More' how can you tell which of the two cases ended the loop? As you know, the loop is ended when the condition is detected or the counter reaches -1 As the `DBcc` instructions do not change the flags you can make a simple check on the Zero flag or test D1 to see if it is -1 or not. So the code that goes in at label 'More' will be this:

```
More     BNE.S      Got_100
Not_100  ;          Process D1 = -1 here
         ;
Got_100  ;          Process D1 = 100 here
         ;
```

Obviously, if we run a loop 1001 times where D1 goes from 1000 to -1, adding 1 to D2 then at some point D2 must equal 100 and that will be the only termination of the loop. D1 will never get to -1.

There are two 'interesting' `DBcc` instructions. These are `DBF` (Decrement and Branch Until False) and `DBT` (Decrement and Branch Until True). What is so interesting about these two?

`DBF` is commonly written as `DBRA` which is more meaningful as it implies that a decrement will be done followed by a branch. This is exactly what happens. The condition FALSE can never be created so the instruction always branches until the counter becomes -1.

`DBT` is the opposite. It never branches because the condition is always detected. I have never seen a `DBT` instruction used in any program I have read, written or disassembled.

Note that the loop is terminated when the counter becomes set to -1. This means that the above loop will have 1,001 iterations assuming that D2 never became 100. This can cause confusion to programmers used to processors that stop at zero. I learned on a Z80 (Sinclair ZX81) and there was a DJNZ instruction which subtracted 1 from the B register and branched if it was non zero.

To loop around 10 times you set B to 10 and just did it. On the 68000 series, you would set the counter to 9 not 10. Some programmers do this and others do it with the counter set to 10 but skip the first iteration. The two examples shown in Listing 3.3 and Listing 3.4 are doing the same thing.

```
Start     MOVEQ     #10,D0
          BRA.S     Skip
Loop      BSR       Useful_code
Skip      DBF       D0,Loop
```

Listing 3.3: Looping Example

```
Start     MOVEQ     #9,D0
Loop      BSR       Useful_code
          DBF       D0,Loop
```

Listing 3.4: Another Looping Example

In Listing 3.3 the programmer sets the counter to the number of times the loop is to be executed but then skips over the loop code itself to the end of the loop. The counter is reduced to 9 and the loop is entered properly this time. The subroutine at label 'Useful_code' will be executed when the counter has values 9,8,7,6,5,4,3,2,1,0 or 10 times.

In Listing 3.4 the programmer sets the counter to 9 and then executes the code as normal. Once again the loop code at subroutine Useful_code will be executed 10 times once again, with the values 9,8,7,6,5,4,3,2,1 and 0 in the counter register D0.

Note

George Gwilt (the author of the GWASL assembler we are using in this series) points out that while the second example is better in terms of readability, there could be problems if the value in the counting register is zero. As George says, the method of subtracting one from the counter then dropping into the loop could lead to a loop that performs 65536 times rather than zero times - how can this be?

Assume that this subroutine is called from another part of some program with the loop counter in D1.W:

```
loopy_bit     SUBQ.W   #1,D1
loop          BSR      do_something
              DBF      D1,loop
              RTS
```

Listing 3.5: Potentially Bug-ridden Looping Example

Obviously, the problem is only apparent when the loop counter is set by some calculation elsewhere in the program, not when setting it directly with immediate data as in my examples above.

Why would this fail, or more to the point, when?

Imagine if D1.W was 1 then the above subroutine called, what would happen? Well, remember how the `DBcc` instructions operate in three parts :

- the condition, if any, is tested to see if it is true. In this case, the condition is ignored as the `DBF` instruction will always loop (it has no condition to check).
- the lowest word of D1 is decremented by one. Then tested to see if it is -1 yet. If it is, the loop is not taken and the `RTS` is executed
- Third, If the counter register is not -1 then the loop is taken to the code at label 'loop'.

So, with D1 set to 1 on entry, the loop is carried out once with D1 adjusted to zero by the SUBQ.W \#1,D1 instruction. The loop will then terminate. No worries here.

What happens if D1 was set to zero on entry?

D1 would be set to -1 by the `SUBQ.W` instruction, then the code at 'do_something' would be executed - but we had a zero count so this is wrong straight away. On return, the condition test would be checked - but as there is no condition with DBF, D1 would be decremented to -2. This does not equal -1 so the branch would be taken and taken again and again until D1 once more became -1. Then it would have been executed 65,536 times too many!

So beware. I can highly recommend the following code instead:

```
loopy_bit     BRA.S    skippy_bit
loop          BSR      do_something
skippy_bit    DBF      D1,loop
              RTS
```

Listing 3.6: Fixed Looping Example

Which will always avoid the above problem. Now if D1 was zero, it will be decremented to -1 when it skips to the `DBF` instruction and this will correctly terminate the loop without executing the code in the do_something sub-routine.

So keep in mind the fact that the loop stops when the counter reaches -1 and that the counter is decremented before testing for -1. Also bear in mind that George is a far better assembler programmer than I am - if he says something, believe it!!

Which is the best to use? It's up to you. Sometimes I use the first form and sometimes the second. As far as reading source code is concerned, I prefer the second method because you can write something like :

```
Start         MOVEQ      #10-1,D0
              :
```

Which at least shows better that the loop will be executed 10 times. Unfortunately, when you disassemble the above instruction the assembler has calculated that 10 - 1 is 9 and it has once again become:

```
Start         MOVEQ      #9,D0
              :
```

The first method, where the loop counter is initialised with the actual iteration count, then skips the loop loses out in that there is the extra `BRA.S` instruction which uses up 2 bytes every time it is used, and the `BRA.S` has to be executed as well as the jump - all of this takes time.

3.3 Counting

3.3.1 Adding and Subtracting

In the above code fragments, I introduced the `ADDQ` instruction to add a value to a register. There are a few arithmetic instructions covering addition, subtraction, division and multiplication.

```
              ADD.size    source,Dn
              ADD.size    Dn,destination
```

This adds the source to the destination. The destination is overwritten but source is not affected. The size can be byte, word or long. All the flags are affected as follows:

- N is set if the result is negative, cleared if not.
- Z is set if the result is zero, cleared if not.
- V is set if an overflow was generated, cleared if not.
- C is set if a carry was generated, cleared if not.
- X is set to the same value as the C flag.

Note that byte sized ADDs cannot be done if source is An. If destination is An then ADDA should be used, however, some assemblers will convert ADD Dn,An into ADDA Dn,An for you.

```
        ADDA.size       source,An
```

This adds the source to the address register specified. The size can only be word or long but note that regardless of the size of source, the whole of the address register is affected. Words are sign extended to 32 bits. This instruction has no effect on the condition codes.

```
        ADDI.size       #data,destination
```

This instruction adds immediate data to the destination. The flags are all affected as per the ADD instruction above. The size can be byte, word or long. It is not permitted to use this to add to an address register.

```
        ADDQ.size       #data,destination
```

This is a very quick version of the above ADDI but it can only be used to add values between 1 and 8 to the destination. The size is byte, word or long as required. This instruction is always 2 bytes long where the ADDI can be 4 or 6 bytes. Use ADDQ wherever a value between 1 and 8 is to be added.

The flags are affected as per the ADDI instruction. The difference between this and ADDI is that you can use ADDQ to add 1,2,3,4,5,6,7 or 8 to an address register. Useful in loops.

```
        ADDX.size       Dx,Dy
        ADDX.size       -(Ax),-(Ay)
```

This one adds with the X flag added as well. It is useful when adding numbers together that are more than a register long - 32 bits. If you were to write a program that used 8 bytes in memory to store a number, then you could add two of them together using ADDX.

The destination becomes set to the value source + destination + X flag.

The flags are affected as follows:

- N is set if the result is negative, cleared if not.
- Z is UNCHANGED if the result is zero, cleared if not.
- V is set if an overflow was generated, cleared if not.
- C is set if a carry was generated, cleared if not.
- X is set to the same value as the C flag.

Note the Z flag. If the result is zero it will be left as it is and not changed. If the result is non zero it is cleared. For this reason the Z flag should be set before any `ADDX`ing takes place so that at the end, the result of zero shows up by having the Z flag still set.

This instruction and the `SUBX` one are mostly used in multiple precision addition and subtraction routines.

```
            ABCD      Dx,Dy
            ABCD      -(Ax),-(Ay)
```

This is Add Binary Coded Decimal and is almost identical to `ADDX` above except that the values in the source and destination are treated as BCD instead of binary. Only 8 bits of the source and destination are affected.

```
Start       MOVEQ     #$19,D0
            MOVEQ     #$03,D1
            ABCD      D0,D1
```

Listing 3.7: ABCD Example

Assuming that the X flag is clear, this will result in D1 being set to $22 which is the result of adding 19 and 3 in DECIMAL. The hexadecimal numbers in the register $19 and $03 are interpreted as decimal digits, one digit for each 4 bits. The above example is actually adding 25 and 3 to make 34!

The flags are affected as follows:

- N is undefined.
- Z is UNCHANGED if the result is zero, cleared if not.
- V is UNDEFINED
- C is set if a DECIMAL carry was generated, cleared if not.
- X is set to the same value as the C flag.

The Subtraction instructions are exactly the same as the Addition flags, but subtract instead. I have listed them below, but not explained them - read the corresponding `ADD` instruction for details.

```
SUB, SUBA, SUBI, SUBQ, SUBX and SBCD.
```

3.3.2 Division and Multiplication

```
            DIVS  source,Dn
```

This instruction divides destination by source and puts the result into destination. Source is a word size and destination is long. The operation is carried out using signed values. The size is always word.

The destination *word* is divided by the source *word* and the result put into the destination *low word*. The remainder is placed in the destination *high word*.

Any attempt to divide by zero will cause a divide by zero exception to occur and on a standard QL this will lock up. If overflow is detected during the operation the overflow flag is set but the operation is aborted and the source and destination are unaffected.

The flags are affected as follows:

- N is set if the quotient is negative, cleared otherwise. Undefined on overflow.
- Z is set if the quotient is zero, cleared if not. Undefined on overflow.
- V is set if division overflow is detected. Cleared otherwise.
- C is always cleared.
- X is never affected. (Unchanged)

For those of us with short memories or a long period since our schooldays, the quotient is the result of the division. The remainder is what is left over.

```
Start       MOVEQ       #100,D0
            MOVEQ       #9,D1
            DIVS        D1,D0
```

Listing 3.8: DIVS Example

Results in D0 being set to \$00010009 which is 9 remainder 1. The 9 is in the lowest word while the 1 is in the highest word.

The instruction should be read as 'divide source into destination'.

```
            DIVU  source,Dn
```

This is identical to the above except that both operands are treated as unsigned numbers. The flags are affected as per the `DIVS` instruction. Although the quotient is always positive, the N flag is set to the value in the highest bit of the lower word of destination. (ie the sign bit of a 16 bit word.)

```
            MULS  source,Dn
```

Multiply the destination word by the source word and place the LONG result into the destination register. Both operands are treated as signed numbers.

The flags affected are:

- N - set if the result is negative, cleared otherwise.
- Z - set if the result is zero, cleared otherwise.
- V - Always cleared.
- C - Always cleared.
- X - Unchanged.

```
            MULU  source,Dn
```

Multiply the destination word by the source word and place the LONG result into the destination register. Both operands are treated as unsigned numbers. The flags are set or cleared as per the `MULS` instruction. The N flag is set to bit 31 of the result.

3.3.3 Negation

```
            NEG.size  destination
```

This instruction converts the binary value in the destination to its two's compliment value. This is done by subtracting the current value from zero, putting the result back into the destination and setting the flags. All the flags are affected by this instruction. The instruction can act upon byte, word or long sized values.

The flags affected are:

- N - set if the result is negative, cleared otherwise.
- Z - set if the result is zero, cleared otherwise.
- V - set if an overflow occurred, cleared otherwise.
- C - Cleared if the result was zero, set otherwise
- X - Set the same as the C flag.

```
        NEGX.size  destination
```

Same as NEG above except the value in the X flag is also subtracted to get the final result. The flags are not affected in the same way as NEG, but as follows:

- N - set if the result is negative, cleared otherwise.
- Z - set if the result is zero, *unchanged* otherwise.
- V - set if an overflow occurred, cleared otherwise.
- C - Set if a 'borrow' was generated, cleared otherwise.
- X - Set the same as the C flag.

```
        NBCD  destination
```

This instruction works on byte sized values only. It is similar to NEGX above, but the values are treated as decimal and not binary. The contents of the byte at 'destination' is subtracted from zero then the current value of the X flag is subtracted as well. The result is put back into 'destination' and the flags set as follows:

- N - undefined
- Z - set if the result is zero, cleared otherwise.
- V - undefined
- C - set if a borrow was required, cleared otherwise
- X - Set the same as the C flag.

3.4 Coming Up...

In the next chapter we shall continue our look at the instruction set with a look at the logical instructions.

4. The 6800x Instruction Set - continued

4.1 Introduction

Following on from the previous chapter, we now start to look at the logical instructions in the MC6800x instruction set.

Logic is the heart of all computer systems - well, all digital ones anyway. Logic is how the central processor works. The 68000 series of processors are no exception and in the instruction set, there are a few logical operations that can be carried out. This chapter discusses those instructions.

4.2 Tie the NOT

The logical `NOT` instruction is probably the simplest of all this family of instruction. It converts the destination address from its current state of ones and zeros into the exact opposite to zeros and ones. The format is:

```
        NOT.size  destination
```

Size can be byte, word or long. The instruction carries out a ‘ones compliment’ of the destination address. If you remember back to the discussion of ‘Twos compliment’ numbers earlier on in the series, you will remember that converting a positive number to negative involved flipping all the zeros and ones and then adding one to the result. The `NOT` instruction carries out the first part of flipping all the ones and zeros over.

If D0.W holds the value of $0001 then after a NOT.W D0, it will hold the value $FFFE. All the original zeros have become ones and vice versa.

NOT must not be confused with the arithmetic `NEG` instruction which carries out a ‘twos compliment’ negation of a value. (D0.W in the above example would become $FFFF which is equivalent to NOT.W D0 followed by ADDQ.W \#1,D0)

NOT affects the flags in the following way:

- N is set if the result becomes negative - the most significant bit becomes a 1. Cleared otherwise.
- Z is set if the result is zero, cleared otherwise.
- V is always cleared - you cannot create an overflow by inverting the bits.
- C is always cleared - there is no carry generated by flipping bits.
- X is not affected.

4.3 This OR That

Next up in the logical family is the `OR` instruction of which there are a few. `OR` is quite different from `NOT` in that it needs to have two operands in order to be used. The format of the `OR` instruction is:

```
        OR.size  source,Dn
```

or

```
        OR.size  Dn,destination
```

Note that in this form of the instruction either the source or the destination must be a data register. The size can be byte, word or long.

This is the 'inclusive or' instruction - there is also an 'exclusive or' variety which we will see later on in this article. An inclusive or works according to the truth table for Logical `OR` in Table 4.1.

Source	Destination	OR
0	0	0
0	1	1
1	0	1
1	1	1

Table 4.1: Truth Table for Logical OR.

Simply imagine each individual bit in the source is being `OR`'d with the same bit in the destination. The result - which will be stored in the destination bit - will always be a 1 if one OR other of the two bits being processed is a 1. If both are zero then the result will also be zero.

An example

D0.W contains $AAAA and D1.W contains $6543 the instruction

```
        OR.W D0,D1
```

Will result in D1.W being set to $EFEB and D0 will remain unchanged. How does this work? In binary:

```
D0 = $AAAA = 1010 1010 1010 1010
D1 = $6543 = 0110 0101 0100 0011
```

So using the truth table above, the result will be:

```
D1 = $EFEB = 1110 1111 1110 1011
```

The flags affected by `OR` are exactly the same as for `NOT` above.

The OR Immediate format of the `OR` instruction has the format :

```
        ORI.size #data,destination
```

and can be byte, word or long sized. It is used when the source value in the `OR` is immediate data as opposed to a register or memory address. Some, but not all, assemblers will allow you to write:

```
        OR.size #data,destination
```

But the actual instruction assembled will be `ORI` instead. Again the flags are affected as for `NOT`.

```
        ORI #data,CCR
```

This instruction is used to set the flags to a set of known values as supplied in the immediate data. This instruction only uses bits 0 through 4 of the data supplied as the other bits are not used in the 68008. As it is possible that future processors may introduce other flags, you are always best to make sure that bits 6 through 7 are zero when using this (and the following) instruction. That way, you won't cause any 'strange effects' on a different processor.

The flags are set as:

- C is set if value in bit 0 of the data is a 1 otherwise unaffected.
- V is set if value in bit 1 of the data is a 1 otherwise unaffected.
- Z is set if value in bit 2 of the data is a 1 otherwise unaffected.
- N is set if value in bit 3 of the data is a 1 otherwise unaffected.
- X is set if value in bit 4 of the data is a 1 otherwise unaffected.

```
        ORI #data,SR
```

This is a similar instruction to the one above, and does a similar job except it affects the entire status register. The other difference is that the processor must be running in Supervisor mode for this instruction to be carried out. If it is not then a privilege exception will be generated - this will hang the QL (usually)

As above, the flags are set according to the data - bits 0 to 4. The rest of the status register is set as follows:

- T (trace) is set if value in bit 15 of the data is a 1 otherwise unaffected.
- S (supervisor) is set if value in bit 13 of the data is a 1 otherwise unaffected.

The value in bits 10, 9 and 8 can be anything from 0 through 7. This is `OR`'d with the current value in the interrupt level bits of the SR and the new value becomes the new interrupt level mask.

Once again, all unused bits must be zero in the data to prevent unpredictable results on different processors. (it is called defensive programming.)

This instruction can be used to turn off all interrupts except level 7. These are known as non-maskable interrupts as they cannot be turned off.

```
        TRAP  #0
        ORI   #$0700,SR
```

This sets the QL so that only a level 7 interrupt will be actioned. The only problem here is that CTRL ALT and 7 activate a level 7 interrupt and effectively hangs your QL. After the above instructions, the supervisor mode is still in effect. (Work it out in binary!!) To exit from supervisor mode ANDI \#\$07FF,SR would need to be done - this leads us nicely into the AND family.

4.4 This AND That

In a similar manner to the OR instruction, the AND instruction needs two operands to work on to get a result.

The format of the AND instruction is:

```
        AND.size  source,Dn
```

or

```
        AND.size  Dn,destination
```

Note that as with the OR instruction, this form of the instruction requires either the source or the destination to be a data register. The size can be byte, word or long.

AND works according to the truth table for logical AND as per Table 4.2.

Source	Destination	AND
0	0	0
0	1	0
1	0	0
1	1	1

Table 4.2: Truth Table for Logical AND.

Simply imagine each individual bit in the source is being ANDed with the same bit in the destination. The result - which will be stored in the destination bit - will always be a 1 if and only if both bits being processed are 1. If either are zero then the result will also be zero.

Using the same example as for OR above:

D0.W contains \$AAAA and D1.W contains \$6543 the instruction:

```
        AND.W  D0,D1
```

Will result in D1.W being set to \$2002 and D0 will remain unchanged. How does this work? Once again, in binary:

```
D0 = $AAAA = 1010 1010 1010 1010
D1 = $6543 = 0110 0101 0100 0011
```

So using the truth table above, the result will be:

```
D1 = $2002 = 0010 0000 0000 0010
```

The flags affected by AND are exactly the same as for `NOT` above.

The `ANDI` (immediate) instruction has the same variations as the `ORI` instruction as described above. These being:

```
        ANDI.size #data,destination
```

And can be byte, word or long sized. It is used when the source value in the `AND` is immediate data as opposed to a register or memory address. Some, but not all, assemblers will allow you to write:

```
        AND.size #data,destination
```

But the actual instruction assembled will be `ANDI` instead. Again the flags are affected as for `NOT`.

```
        ANDI #data,CCR
```

This is an instruction that is used to reset or clear some or all of the flags. The flags are reset as follows:

- C is reset if value in bit 0 of the data is a 0.
- V is reset if value in bit 1 of the data is a 0.
- Z is reset if value in bit 2 of the data is a 0.
- N is reset if value in bit 3 of the data is a 0.
- X is reset if value in bit 4 of the data is a 0.

```
        ANDI #data,SR
```

This is another instruction which works on the status register but affects the entire width of the status register, not just the CCR byte.

As above, the flags are reset according to the value in bits 1 - 4 of the immediate data. The rest of the status register is reset as follows :

- T (trace) is reset if the value in bit 15 of the data is 0.
- S (supervisor) is rest if the value in bit 13 of the data is 0.

The value in bits 10, 9 and 8 is `AND`ed with the current value in the interrupt level bits of the SR and the new value becomes the new interrupt level mask.

All unused bits should be one in the data to prevent unpredictable results on different processors.

This instruction can be used to exit from supervisor mode. The instructions:

```
        TRAP  #0
        ANDI  #$D7FF,SR
```

Would set the QL so that supervisor mode was first switched on (by the (TRAP #0) and then only the supervisor bit in the SR was cleared (bit 13) so the QL would revert to user mode. All other modes and interrupt levels and flags would remain unchanged.

4.5 Exclusive OR Instructions

Having dealt with the inclusive or instructions above, it is now time for the exclusive or instructions. This has the format:

```
        EOR.size  Dn,destination
```

Where size can be byte, word or long. Notice this time that EOR source,Dn is not permitted? I wonder why? (I don't know - does anyone?)

This instruction also sets the flags as per the NOT instruction. In the truth table for inclusive or, there was a 1 bit set in the result when there was a 1 in either the source or destination or both. Exclusive or is different and only allows a 1 in the result when there is a single 1 in either the source or destination. As shown in the truth table for Logical EOR in table 4.3.

Source	Destination	EOR
0	0	0
0	1	1
1	0	1
1	1	0

Table 4.3: Truth Table for Logical EOR.

Using the same example as OR and AND above we now have the following :

D0.W contains $AAAA and D1.W contains $6543 the instruction

```
        EOR.W  D0,D1
```

Will result in D1.W being set to $CFE9 and D0 will remain unchanged. How does this work? Once again, in binary:

```
D0 = $AAAA = 1010 1010 1010 1010
D1 = $6543 = 0110 0101 0100 0011
```

So using the truth table above, the result will be:

```
D1 = $CFE9 = 1100 1111 1110 1001
```

One feature of EOR is that if you EOR the result of a previous EOR with the same value again, you get back to the original value. Using this code:

```
            MOVEQ  #$AAAA,D0
            MOVEQ  #$6543,D1
            EOR.W  D0,D1
            EOR.W  D0,D1
```

Will return us to the state we were in before the first EOR, in that D1 will once again hold the value $6543. Try to work it out for yourselves using the example above as a guideline.

This can be used in a sort of 'Pretty Bad Privacy' program where data is encrypted using EOR. The following small program demonstrates this.

```
START       MOVEQ       #7,D0
            LEA         DATA_STUFF,A1
            MOVEQ       #100-1,D1
LOOP        EOR.B       D0,(A1)+
            DBF.S       D1,LOOP
            RTS

DATA_STUFF              Put 100 bytes of data here!
```

Listing 4.1: Pretty Bad Privacy Example

The LEA instruction is a new one and will be discussed soon. Suffice to say that it simply loads the address of the label 'data_stuff' into the address register named. This must be used in QL programs as they have to be able to run at any memory address. The LEA instruction allows this.

The above code is very simple and assumes that there is exactly 100 bytes of data stored in memory at the location labeled 'data_stuff' To encrypt the data, simply call the routine at label 'start' and 100 bytes will be encrypted. To decrypt it, simply call 'start' again and the data will be restored.

Warn

This extremely bad for of encryption is extremely easily cracked because of the use of a single byte to encrypt the data so don't go using it for anything you value, such as your bank account details!

EOR has the usual variations:

```
            EORI  #data, destination
```

You will notice the absence of EORI source,Dn as mentioned above.

```
            EORI  #data,CCR
```

EORI \#data,CCR is an instruction that is used to change some or all of the flags in the user byte of the status register. The flags are changed as follows:

- C is changed if the value in bit 0 of the data is a 1.
- V is changed if the value in bit 1 of the data is a 1.
- Z is changed if the value in bit 2 of the data is a 1.
- N is changed if the value in bit 3 of the data is a 1.
- X is changed if the value in bit 4 of the data is a 1.

```
        EORI #data,SR
```

EORI \#data,SR works upon the entire status register.

As above, the flags are changed according to the data - bits 0 to 4. The rest of the status register is changed as follows:

- T (trace) is changed if the value in bit 15 of the data is 0.
- S (supervisor) is changed if the value in bit 13 of the data is 0.

The value in bits 10, 9 and 8 is EOR'd with the current value in the interrupt level bits of the SR and the new value becomes the new interrupt level mask.

4.6 Shifting And Rotating

There are 4 shift and 4 rotate instructions, 2 going left and 2 going right.

ASL and ASR are arithmetic shifts while LSL and LSR are logical shifts. What is the difference? Taking the logical shifts first we have :

```
        LSL.size Dx,Dy
```

or

```
        LSL.size #data,Dy
```

or

```
        LSL.W address
```

LSR has the same format, it just shifts in the opposite direction.

For the first two variations above, the data in Dy is affected and the size can be byte, word or long. The number of shifts that take place is defined by the value in register Dx or in the immediate data.

For the final variation, the size must be word only, and the data in that address and the address above it, is affected. For this format, there can only be a single shift at a time.

What happens is that the data is shifted by a single bit at a time. The bit that is shifted 'out' of the register is placed into the C and X flags, while the 'vacant' bit is filled with a zero.

Consider this example:

```
        MOVEQ   #$81,D0     ; D0.B is 1000 0001
        LSL.B   #1,D0       ; Now it is 0000 0010 and
                            ; C and X are 1
        MOVEQ   #5,D2
        LSL.B   D2,D0       ; Now D0.B is 0100 0000
```

Listing 4.2: LSL Example

Shifting the opposite way gives this:

```
            MOVEQ     #$81,D0       ; D0.B is 1000 0001
            LSR.B     #1,D0         ; Now it is 0100 0010 and
                                    ; C and X are 1
            MOVEQ     #5,D2
            LSR.B     D2,D0         ; Now D0.B is 0000 0010
```

Listing 4.3: LSR Example

`LSL` is a quick way of multiplying an unsigned number by 2 for each bit shifted.

`LSR` is a quick way of dividing an unsigned number by 2 - but the fractions are lost. Another couple of examples:

```
            MOVEQ     #8,D0         ; D0.L holds 8
            LSL.L     #1,D0         ; D0.L now holds 16
            LSL.L     #2,D0         ; D0.L now holds 64
```

Listing 4.4: LSL Multlication Example

```
            MOVEQ     #10,D0        ; D0.L holds 10
            LSR.L     #1,D0         ; D0.L now holds 5
            LSR.L     #1,D0         ; D0.L now holds 2 but
                                    ; note the remainder is ‘lost’
```

Listing 4.5: LSR Division Example

When specifying the number of shifts as immediate data, only values from 1 to 8 can be used. If the number of shifts required is greater than this, then a register counter has to be used. When shifting memory, the shift is always a single bit.

After a shift in either direction the flags are set as follows:

- N is set if the result became negative (MSB set to 1), cleared otherwise.
- Z is set if the result became zero, cleared otherwise.
- V is always cleared.
- C is set to the LAST bit shifted out, cleared if the shift count was zero.
- X is set to the LAST bit shifted out. *unaffected* if the shift count was zero.

The arithmetic shifts - `ASL` and `ASR` - preserve the sign of the value by duplicating the previous value of the sign bit in the new sign bit, so everything shifts as above, but the most significant bit of the byte, word or long being shifted, is shifted back into itself.

4.7 Coming Up...

In the next chapter we shall finish looking at the remainder of the instruction set for the MC6800x. That should conclude the most boring bits of learning about the processor.

5. The 6800x Instruction Set - continued

5.1 Introduction.

In this chapter we'll take a look at the remaining instructions which we have yet to cover. There are not many left now - you'll be glad to hear.

5.1.1 A Few Quickies!

This section deals with a few instructions that the QL programmers rarely, if ever, use. These instructions are:

```
CHK
ILLEGAL
RESET
RTR
STOP
TRAPV
```

The `CHK` instruction has the format:

```
            CHK  <ea>,Dn
```

and causes an exception to be generated if the value in Dn.W is less than 0 or greater than the value in the effective address. On a normal QL this is totally ignored - the exception that is - however, with a bit of deft QDOS programming, this can be redirected to your own routine. I have never seen this done in any programs - yet! By the way, the value in the effective address is a two's compliment signed number. The flags affected are:

- N - set if Dn.W is less than zero, cleared is Dn.W is greater than the effective address value. Otherwise it is undefined.

- Z - undefined.
- V - undefined.
- C - undefined.
- X - unaffected.

The format of the ILLEGAL instruction is quite simply:

```
        ILLEGAL
```

and all it does, by default, is to crash the QL! It can however be redefined to do something useful as with the CHK instruction. (We may get around to covering QDOS stuff in a much later episode.) This instruction also causes an exception to be generated. No condition codes are affected. The RESET instruction has the format:

```
        RESET
```

and causes the 'reset' line to be 'asserted' causing all external equipment interfaced to the processor to be reset. On the QL, it actually causes a system reset - similar to you pressing the reset switch. This instruction will only be executed if the processor is running in supervisor mode, in user mode, all that happens is that the program counter is incremented by 2 to skip over this instruction. No flags are affected.

RTR has the format:

```
        RTR
```

and is actually equivalent to the following two instructions:

```
        MOVE (A7)+,SR
        RTS
```

However, the MOVE (A7)+,SR instruction is privileged on the 68000 so can only be run in supervisor mode. Using RTR is not privileged so the two instructions can be combined as one. This is a useful instruction for subroutines where the status register is saved on the stack on top of the return address. The following code is an example.

```
start   BSR example
        ; more code here

example MOVE SR,-(A7)       ; Stack the status register etc
        ; do some code here

        RTR                 ; Unstack the status code
```

Listing 5.1: RTR Example

What happens when a subroutine is called is that the return address is placed on the stack and then the subroutine jumped to. In this example the status register is placed on the stack as well. This is a word sized SR on top of a long sized Program Counter.

The subroutine carries out various bits of processing - probably trashing the status codes etc as it does so. At the end, the old SR is put back into the SR and the return address placed in the PC by the `RTR` instruction.

It is a quirk of the 68000 that the instructions to move data from the SR are not privileged while those that move data into the SR are privileged. This is a handy way around this restriction.

Obviously, the various flags in the SR are changed according to the word removed from the stack *except for the supervisor bit which is unchanged.*

The `STOP` instruction has the format:

```
        STOP #data
```

and causes the processor to put the word of data into the SR, increment the PC to point at the instruction following this one, and then the processor just stops - until any trace, interrupt or reset exceptions are generated. The interrupt must be higher that the current processor interrupt level to have any effect. The flags are set according to the data word in the instruction. This is another privileged instruction and is the processor is in user mode, and a privilege violation exception will be generated.

The `TRAPV` instruction has the format:

```
        TRAPV
```

and is used to cause an exception if the V flag is set. (Overflow flag). Normally this is ignored on a QL but can be redirected with the afore mentioned QDOS jiggery pokery to do something useful. No flags are affected.

5.1.2 A Few Little Bits

This section deals with instructions that check, change, set or otherwise fiddle about with the individual bits in a register or memory address. All of these instructions have a similar format, which is:

```
Bxxx Dn,<effective address>
Bxxx #data,<effective address>
```

They all TEST the bit about to be fiddled with *before* fiddling with it. The flags are set according to the state of the bit *before* the fiddling was done. Remember this important fact. The bit number is either supplied in a data register or as immediate data.

When the bit number is being processed the 68000 makes sure that it is in range for the actual data being operated on. If the effective address is a data register (you cannot use these instructions on address registers) then the actual bit number is bit number MOD 32.

If a memory address is being manipulated, the range is adjusted to be 0 to 8 using bit number MOD 8.

The flags are all unaffected except for the Z flag which takes the state of the 'previous' value of the bit being manipulated.

The instructions are:

Instruction	What it does	Description
BCHG	Bit CHanGe	Changes the specified bit from a 1 to a zero or from a zero to a 1.
BCLR	Bit CLeaR	Puts a 0 into the specified bit.
BSET	Bit SET	Puts a 1 into the specified bit.
BTST	Bit TeST	Sets the Z flag to the value of bit specified.

Table 5.1: Bit Twiddling instructions.

This family of instructions are very useful when using a byte, word or long to hold 8, 16 or 32 different flags in a program as each one can be tested, set or reset individually and this takes place within QDOS in a number of places.

As a small example, imagine you were writing a program and you needed to check when the user typed an UPPERCASE character. Rather than checking every one for 'A' and 'Z' (which only apply to the English language remember, you could set up a bitmap table of 256 bytes and have a single bit represent uppercase, another could be for numeric, another for control/unprintable characters etc etc. As each character was read, index into the table on that character code and check the appropriate bit.

```
;
; Some code above to get a character from the user/file etc
; Assume D1.B holds the character code.
; Assume that bit 0 is the uppercase/lowercase flag bit.
;

checkUC LEA bitmap,A1          ; A1 is address of the bitmap table
        EXT.W D1               ; Ensure D1.W is the character code
        BTST #0,(A1,D1.W)      ; Is it uppercase?
        BEQ.S upper            ; Yes, if bit zero is set

lower   ; process lowercase here

upper   ; process uppercase here

bitmap  ; 256 bytes go here, one for every character.
```

Listing 5.2: Uppercase Check Example

The bitmap table has a single byte for each available character 0 to 255 and sets the bits in each one according to the character type. In this example we use bit 0 for upper/lower case only so wastes 7 bits of each byte, but remember, these extra bits could be used to define control characters, digits, hex digits, alphabetic, alpha-numeric, punctuation etc.

The advantage to this method is that different tables can be loaded for different languages. The disadvantage is that the program will be slightly longer because of the need to store the table.

5.1.3 Testing, Testing

In QLTdis[1], I have used the `TST` instruction to compare a value against zero. This is a useful instruction and replaces CMPI.size #0,Dn. The format is:

[1] QLTdis is a long abandoned project for this Assembly Language tutorial. It fell victim to a lack of planning, foresight and most likely, ability, on my part. When I say *abandoned* it hasn't been lost for good ....

```
        TST.size <effective address>
```

The flags are set differently from CMPI as well as the V and C flags are always cleared to zero. CMPI doesn't do this. The flags are:

- N is set if the operand is negative, reset if positive.
- Z is set if the operand is zero, reset otherwise.
- V is always cleared.
- C is always cleared.
- X is not affected.

Why use TST when CMPI will do as good a job? Well it is all down to three things really:

- Do you want to use TST or CMPI #0?
- Do you need to preserve the V and C flags?

TST is quicker. TST takes 8 clock cycles while CMPI takes 16, 24 or 26 depending on the operation. Both take the same time to work out the effective address calculation, but TST also needs fewer read cycles - 2 - while CMPI needs 4 or 6.

TAS is another testing instruction, which actually does two separate operations in one single *atomic*[2] step. The format is:

```
        TAS <effective address>
```

The size is always byte and need not be specified. The flags affected are:

- N is set if bit 7 of the operand was set, otherwise cleared.
- Z is set of the operand was zero, Reset otherwise.
- V is always cleared.
- C is always cleared.
- X is not affected.

The instruction reads the byte at *effective address*, checks bit 7, sets the flags and then sets bit 7. The modified byte is written back to the effective address. It is similar to the following code:

```
        BTST    #7,<effective address>
        BSET    #7,<effective address>
```

Obviously there are two instructions here which alter the flags, however, TAS does it in one. The main point about TAS is that it is a single instruction which cannot be interrupted once it has started. This makes it useful for multi tasking or multi processor systems where any sequence of instructions can be interrupted.

In the above example, the system could be interrupted by a floppy disc I/O request between the end of the BTST and the start of the BSET. This could result in a new value being placed into *effective address* by the interrupting routine. The BSET would then possibly give the wrong results after it executed.

[2]An atomic instruction is one that cannot be split, like the atoms in Chemistry *used* to be considered. The TAS instruction effectively carries out a BTST and then a BSET instruction. While the two instructions could be usurped by the scheduler the single TAS cannot. So rogue and intermittent problems cannot occur. TAS is useful when using semaphores in your code. But that's a whole different ball game!

This will not ever happen with the TAS instruction. If the above code was being used in a multi processor system to synchronise access to some system resource, the two instructions could lead to mis-synchronisation. Using TAS would not allow this to happen.

Finally in this section, although not quite a testing instruction, is the ‘set according to condition code’ instructions. These have the format:

```
        Scc <effective address>
```

The size is always byte and is not specified in the instruction. What happens is that the condition code is tested, and if found to be true, the byte in *effective address* is changed to be all ones otherwise it is changed to be all zeros. The condition codes are as for Bcc and DBcc.

This sets a memory address or a byte in a register to 255 or 0 for true or false. On QDOS systems we tend to use 1 for true and 0 for false. How can we quickly change from 255 and zero to 1 and zero?

The answer is quite simple, 255 is an unsigned number but if it was signed, it would be -1. Simply follow the Scc instruction with NEG.B as follows:

```
                        ; Do some code here to set condition flags.
SMI      D1             ; Set D1.B to $FF if ‘something’ was minus
NEG.B    D1             ; D1.B now is $01 or $00 which is what we want!
```

5.1.4 And Finally?

I think we are just about finished covering all those boring instructions, but we still have a couple to do yet. These don’t really fall under any of the headings I have used up until now, so I simply add them on at the end!

On the QL, assembly language programs must be written so that they are ‘relocatable’. All this means is that you must not assume that your code will always run from a specific address but that it could run from ANY address.

The LEA instruction which has been used quite a lot in QLTdis already allows just this to happen. This has the format:

```
        LEA <effective address>,An.
```

None of the flags are affected. So, a quick bit of revision, what is the difference between the following two instructions?

```
        MOVE <effective address>,A1
        LEA <effective address>,A1
```

MOVE calculates the effective address and reads its contents into A1 while LEA calculates the effective address and puts that into A1, not its contents.

This allows position independent code to be written and is a very much used instruction in QDOS programs. It also helps get around the fact that PC relative mode addressing is forbidden as the destination in a MOVE instruction. The following code will not assemble:

```
        MOVE.L   D0,buffer(PC)
```

But this will, and does what is required:

```
        LEA      buffer,A1
        MOVE.L   D0,(A1)
```

There is a similar instruction called Push Effective Address and this has the format:

```
        PEA <effective address>
```

and simply calculates the effective address and puts it onto the stack. The stack pointer is pre-decremented and none of the flags are affected. All this is very similar to the following:

```
        LEA      some_code,A1      ; Get the address of some_code
        MOVE.L   A1,-(A7)          ; Stack it
```

But why would you use PEA to do this rather that the above, and what use is it afterwards? Apart from it being shorter to code - one instruction instead of two - it doesn't require a register to be used. The address is on the stack, so what next?

Think about these instructions:

```
        PEA      some_code,A1      ; Get the address of some_code
        RTS
```

What has just happened? The address of the routine at 'some_code' has been placed on the stack, then when RTS is executed, it returns control to the address *which is on the stack*. So this is another way of doing this:

```
        LEA      some_code,A1
        JSR      (A1)
```

Why would you use this? I have absolutely no idea! But it is important to note that the first method will *never* return to the address after the RTS because there is no return address on the stack. The second and 'normal' method will return to the address after the JSR (A1) as the JSR stacks its return address.

The next and final two instructions are seldom used in normal assembler programs on the QL - at least, I have never seen one in all my years of reading & writing code.[3] They are probably used most by the code generated by various compilers that exists for the QL so that 'stack frames' can be built and parameters passed to sub-routines created by the compiler. The two instructions are LINK and UNLK and they do not affect any flags.

The LINK instruction has the format:

[3] And now, after all these intervening years, I've actually written an article on their very use after George pointed out that he uses them, frequently, in GWASL and GWASS etc.

```
        LINK An,#displacement
```

and carries out the following actions:

- First the stack pointer is decremented by 4.
- Then, the current contents of An are copied onto the stack.
- Then the stack pointer is copied to An.
- Finally, the stack pointer has the displacement ADDED to it.

`UNLK` has the format:

```
        UNLK An
```

and carries out the reverse of the `LINK` instruction in that the stack pointer is reloaded from An, then An is reloaded from the value on the stack and the stack pointer is incremented by 4.

Assuming that A7 is currently holding value $20000 and A4 is holding $00123456 the the sequence of instructions:

```
        LINK A4,-$10
                        ; do something here ...
        UNLK A4
```

will result in the following:

- A7 will be decremented by 4 to $1fffc
- A4 will be stored at this address ($1fffc)
- A4 will then have $1fffc loaded into it
- A7 will have $10 subtracted (because we supplied a negative displacement) to give $1ffec.

This means that the code between the `LINK` and `UNLK` instructions can use the free space between (a7) and -4(A4) for working space. There are 16 bytes available for use between these addresses and they can be accessed using A4 as a 'stack frame pointer' and using negative offsets. Any stacked valued set up prior to the `LINK` instruction can be accessed using positive offsets from A4.

Once the `UNLK` instruction is reached, we must not have changed the value in A4 or all hell will break loose!

- A7 is set to the value in A4 which should be $1fffc.
- A4 will be set to the long word at 0(a7) which is where its original value of $00123456 was stored by the `LINK` instruction.
- A7 will have 4 added to it giving the original $20000 that we had when the `LINK` was executed.

5.1.5 So Here We Are!

Well, that is the end of the most boring part of this series. I apologise for the dreary nature of the previous few chapters but I can't think of any other way to make a micro-processor's instruction set interesting reading!

We have now covered all the 68008 instructions and the time has come to start putting the information into practice. However, when I was learning all about 68000 assembly language, there were a few concepts that gave me troubles - and I still have to look them up even today!

To make things a bit easier for you, here are my bug-bears and an explanation of how to get around them.

Comparing Things

Comparing registers or registers and values etc always gives me problems. I can never remember which flags are set or which ones to check when using signed or unsigned values. The following should hopefully make life easier.

Remember, when using the `CMP` instruction, you should read it as 'if destination condition source'.

Equality checks - signed and unsigned are the same.

```
        CMP.L   D0,D1
        BEQ.S   equal       ; if d1 = d0 goto equal.
```

or

```
        CMPI.L  #10,D1
        BEQ.S   equal       ; if d1 = 10 goto equal.
```

Non-equality checks - signed and unsigned are the same.

```
        CMP.L   D0,D1
        BNE.S   not_equal   ; if d1 <> d0 goto not_equal.
```

or

```
        CMPI.L  #10,D1
        BNE.S   not_equal   ; if d1 <> 10 goto not_equal.
```

Greater than - unsigned only.

```
        CMP.L   D0,D1
        BHI.S   greater     ; if D1 > D0 goto greater.
```

or

```
        CMPI.L  #10,D1
        BHI.S   greater     ; if D1 > 10 goto greater.
```

Greater than - signed only.

```
        CMP.L   D0,D1
        BGT.S   greater     ; if D1 > D0 goto greater.
```

or

```
        CMP.L   #-5,D1
        BGT.S   greater       ; if D1 > -5 goto greater.
```

Greater Than or Equal - unsigned only.

```
        CMP.L   D0,D1
        BCC.S   greater_eq    ; if (D1 >= D0) goto greater_eq
```

or

```
        CMPI.L  #5,D1
        BCC.S   greater_eq    ; if (D1 >= 5) goto greater_eq
```

Greater Than or Equal - signed only.

```
        CMP.L   D0,D1
        BGE.S   greater_eq    ; if (D1 >= D0) goto greater_eq
```

or

```
        CMPI.L  #-5,D1
        BGE.S   greater_eq    ; if (D1 >= -5) goto greater_eq
```

Less than - unsigned only.

```
        CMP.L   D0,D1
        BCS.S   less          ; if D1 < D0 goto less
```

or

```
        CMPI.L  #5,D1
        BCS.S   less          ; if D1 < 5 goto less
```

Less than - signed only.

```
        CMP.L   D0,D1
        BLT.S   less          ; if D1 < D0 goto less
```

or

```
        CMPI.L  #-5,D1
        BLT.S   less          ; if D1 < -5 goto less
```

Less than or equal - unsigned only.

```
        CMP.L   D0,D1
        BLS.S   less_eq         ; if D1 <= D0 goto less_eq
```

or

```
        CMPI.L  #10,D1
        BLS.S   less_eq         ; if D1 <= 10 goto less_eq
```

Less than or equal - signed only.

```
        CMP.L   D0,D1
        BLE.S   less_eq         ; if D1 <= D0 goto less_eq
```

or

```
        CMPI.L  #10,D1
        BLE.S   less_eq         ; if D1 <= 10 goto less_eq
```

Signed Numbers being MOVEd

Remember also that flags and conditions are set when data is `MOVE`d into data registers, or after arithmetic etc, so the following are valid as well. Obviously, the following code will not work correctly if you find this in a real program - don't use it!

```
        MOVE    D1,D0               ; Copy D1 to D0 & set
                                    ; the flags accordingly
        BEQ.S   D1_is_zero          ; D1 is now 0
        BNE.S   D1_is_not_zero      ; D1 is not 0
        BGE.S   D1_is_0_or_more     ; D1 is now 0 or greater
        BPL.S   D1_is_0_or_more     ; D1 is now 0 or greater
        BGT.S   D1_is_1_or_more     ; D1 is now greater than 0
        BLE.S   D1_is_0_or_less     ; D1 is now less then 0
        BLT.S   D1_is_negative      ; D1 is now less than 0
        BMI.S   D1_is_negative      ; D1 is now less than 0
```

5.2 Coming Up...

That's it, there are no more instructions to learn. In the next chapter, we will investigate the various exceptions that can occur when things go wrong, and what if anything, we can do on the QL to prevent and handle them.

II SuperBasic, QDOS and Other Interesting Stuff. Part 1

6. 6800x Exceptions And Exception Handling

6.1 Introduction

In this chapter, we are going to get stuck into a fairly complex part of the 6800x processor's workings - exception processing.

6.2 Exceptions

As mentioned in the instruction summary in past articles, the QL processor runs in two modes - user and supervisor - and some instructions cannot be run in user mode without causing an exception to be generated. I promised to explain what these exceptions are, so here goes ....

An exception is an event or happening that causes the processor to deviate from its normal course of action and to jump to a predetermined place in the operating system where it starts executing a piece of code that handles such events. In QDOS (the QL's operating system) many of these routines have been 'botched' in an effort to save on memory and others simply do nothing. This is unfortunate, however, all is not lost.

All 68000 series processors have an area of memory set aside to hold the exception table. This table is 1024 bytes long and holds a full set of exception vectors - basically a long word holding the address of the sub-routine that handles the appropriate exception. In QDOS this table is only partially there as will become clear. There are 256 vectors normally, each one being 4 bytes long. Vector zero is at address zero in the memory map and vector 255 is at address $3FC.

The vector table *should* look like Table 6.1.

It can be seen that a huge number of the vectors are reserved for Motorola to use in future processors. The User vectors look interesting, but have been obliterated by some of the code in QDOS and cannot be used.

On the QL, the vectors are actually as per Table 6.2.

Vector	Address	Purpose
000	0000	Reset - SSP value
001	0004	Reset - USP value
002	0008	Bus Error
003	000C	Address Error
004	0010	ILLEGAL Instruction
005	0014	Divide by zero
006	0018	CHK instruction
007	001C	TRAPV instruction
008	0020	Privilege violation
009	0024	Trace
010	0028	Line 1010 emulator
011	002C	Line 1111 emulator
012	0030	Reserved for Motorola
013	0034	Reserved for Motorola
014	0038	Reserved for Motorola
015	003C	Uninitialised Interrupt
016	0040	Reserved for Motorola
...	...	...
024	0060	Spurious interrupt
025	0064	Interrupt level 1
026	0068	Interrupt level 2
027	006C	Interrupt level 3
028	0070	Interrupt level 4
029	0074	Interrupt level 5
030	0078	Interrupt level 6
031	007C	Interrupt level 7
032	0080	TRAP #0
033	0084	TRAP #1
034	0088	TRAP #2
035	008C	TRAP #3
036	0090	TRAP #4
037	0094	TRAP #5
038	0098	TRAP #6
039	009C	TRAP #7
040	00A0	TRAP #8
041	00A4	TRAP #9
042	00A8	TRAP #10
043	00AC	TRAP #11
044	00B0	TRAP #12
045	00B4	TRAP #13
046	00B8	TRAP #14
047	00BC	TRAP #15
048	00C0	Reserved for Motorola
...	...	...
064	0100	User vector 1
...	...	...
255	03FF	User vector 192

Table 6.1: MC6800x Exception Table

Vector	Address	Purpose	Comment
000	0000	Reset	SSP value
001	0004	Reset	USP value
002	0008	Bus Error	Ignored by QDOS
003	000C	Address Error	May be redefined
004	0010	ILLEGAL Instruction	May be redefined
005	0014	Divide by zero	May be redefined
006	0018	CHK instruction	May be redefined
007	001C	TRAPV instruction	May be redefined
008	0020	Privilege violation	May be redefined
009	0024	Trace	May be redefined
010	0028	Line 1010 emulator	Unusable
011	002C	Line 1111 emulator	Unusable
012	0030	Reserved for Motorola	Unusable
013	0034	Reserved for Motorola	Unusable
014	0038	Reserved for Motorola	Unusable
015	003C	Uninitialised Interrupt	Unusable
016	0040	Reserved for Motorola	Unusable
...	...	...	
024	0060	Spurious interrupt	Ignored by QDOS
025	0064	Interrupt level 1	Ignored by QDOS
026	0068	Interrupt level 2	QL System interrupt
027	006C	Interrupt level 3	Ignored by QDOS
028	0070	Interrupt level 4	Ignored by QDOS
029	0074	Interrupt level 5	Ignored by QDOS
030	0078	Interrupt level 6	Ignored by QDOS
031	007C	Interrupt level 7	Hangs the QL - May be redefined
032	0080	TRAP #0	Make a call to QDOS
033	0084	TRAP #1	Make a call to QDOS
034	0088	TRAP #2	Make a call to QDOS
035	008C	TRAP #3	Make a call to QDOS
036	0090	TRAP #4	Make a call to QDOS
037	0094	TRAP #5	Ignored by QDOS - May be redefined
038	0098	TRAP #6	Ignored by QDOS - May be redefined
039	009C	TRAP #7	Ignored by QDOS - May be redefined
040	00A0	TRAP #8	Ignored by QDOS - May be redefined
041	00A4	TRAP #9	Ignored by QDOS - May be redefined
042	00A8	TRAP #10	Ignored by QDOS - May be redefined
043	00AC	TRAP #11	Ignored by QDOS - May be redefined
044	00B0	TRAP #12	Ignored by QDOS - May be redefined
045	00B4	TRAP #13	Ignored by QDOS - May be redefined
046	00B8	TRAP #14	Ignored by QDOS - May be redefined
047	00BC	TRAP #15	Ignored by QDOS - May be redefined
048	00C0	Reserved for Motorola	Unusable
...	...	...	
064	0100	User vector 1	Unusable
...	...	...	
255	03FF	User vector 192	Unusable

Table 6.2: QDOS Exception Table

Note

All vectors marked "Unusable" have been botched in the ROM and have bits of code in place of the vectors. So you can see not much is left. The designers of QDOS didn't have enough room in the early ROMs to fit all the code in. Some early QLs came with a ROM 'dongle' hanging out of the external ROM slot so that all the code could fit.

Later versions got rid of the ROM dongle, but the exception vector table had then been 'redesigned' to make the code fit. Luckily the developers did allow a number of the exceptions to be redefined so that programmers could write their own routines to handle these exceptions.

6.3 Working QDOS Exceptions

RESET - vectors 0 and 1 - these two vectors are simply the values that are put into the SSP and USP on system power up. Vector 0 gives the value for the stack pointer for supervisor mode and vector 1 gives the stack pointer for user mode.

ADDRESS ERROR - this occurs whenever the processor tries to do a word or long sized operation or access at an odd address. For example, the following code fragment will cause an address error:[1]

```
        MOVEA.L     #1,A1
        MOVE.W      (A1),D0
```

On a normal QL this will usually cause the system to hang, but as the vector can be redefined, we can use it to point to an address that can correctly handle this error. More on this later.

ILLEGAL INSTRUCTION - this occurs when an instruction is executed that is not a valid instruction for the processor, or when the `ILLEGAL` instruction is executed. Illegal usually crashes the QL, but can be handled by our own routines.

DIVIDE BY ZERO - This should be obvious. This is ignored on the QL, but can be redefined for our own use.

CHK INSTRUCTION - Called when the `CHK` instruction is used and the value in a data register is out of bounds, Ignored on the QL but redefinable.

TRAPV INSTRUCTION - Called when the `TRAPV` instruction is executed and the V flag is set. Ignored by the QL but, once again, is redefinable.

PRIVILEGE VIOLATION - When a program running in user mode attempts to execute an instruction that is privileged, this exception is raised. Ignored by the QL, but redefinable.

TRACE - If the trace (T) bit is set in the Status Register, this execption is generated after each instruction. Can be redefined to call code in a machine code monitor program, but usually ignored by the QL.

INTERRUPT LEVEL 2 - there are 7 levels of interrupt on a normal 68000 series processor, but only one is used on the QL. The level 2 interrupt is generated by the internal electronics and causes the keyboard to be scanned, the scheduler to switch tasks etc. Levels 1 and 3 to 6 are ignored on the QL.

INTERRUPT LEVEL 7 - Level 7 is the non-maskable interrupt and is raised when you press CTRL ALT 7 together. When the QL hardware was being built and debugged, some external equipment was 'bolted on' and this combination of keys caused a level 7 interrupt which activated the debugging equipment. Unfortunately, when the QL went into production, the code was left in and pressing these keys together is a pretty good way to trash the system. May be redefined for our own use - this could be fun !

TRAP #0 - Switch the QL into supervisor mode and cause the SSP version of A7 to be used.

[1] Well, it will on a standard QL's MC68008 processor. With QPC's emulation of an MC68020 however, it will work quite happily.

TRAP #1 - this is the QDOS manager trap and is used to control resources in the QL such as baud rates, jobs, memory allocation and deallocation etc.

TRAP #2 - this is the QL's I/O manager trap and is used to open & close channels as well as formatting discs and deleting files.

TRAP #3 - This allows QDOS to read data from channels, queues, set colours etc.

TRAP #4 - Used by the SuperBasic interpreter to switch between A6 relative and Absolute addresses when calling various routines.

TRAP #5 to **TRAP #15** - these are unused on the QL but can be redefined.

6.4 What Happens When an Exception Occurs?

When an exception occurs, some data is put onto the stack prior to the exception being processed. Remember, the stack pointer is the SSP and not the normal USP variant of the A7 register.

For most exceptions, the data put onto the stack is simply the program counter and the status register as follows:

```
High address -> PC low word
                PC high word
SSP ----------> Status register word
```

so when the exception handler is running, the stack pointer holds the address of the SR at the time the exception was caused and 4(A7) holds the program counter where the exception was caused.

The above is true for all exceptions apart from BUS ERROR, ADDRESS ERROR or RESET. These three have a different stack frame:

```
High address -> PC low word
                PC high word
                Status register word
                Instruction register word
                Access address low word
                Access address high word
SSP ----------> Access type and function code (one word)
```

This additional data includes a copy of the first word of the instruction that was being processed when the exception was caused, the address that was being accessed when the exception was caused and a word describing what the processor was trying to do at the time.

Warn

Note that the value in the program counter on the stack is not always the *actual* address of the start of the instruction - it could be anything from the next word or even the address 10 bytes on from the actual address of the instruction - beware.

At the end of an exception processing routine an `RTE` instruction is used to restore the status register and the program counter from the stack. It follows then that in the case of an ADDRESS or BUS exception that this is going to fail unless the additional data is first cleared from the stack - or a 68020 used instead!

6.5 Building an Exception Handler.

I suppose we need to built an exception handler now! In the QL you build a table of vectors for the following exceptions:

```
Address error
Illegal
Divide by zero
CHK
TRAPV
Privilege
Trace
Interrupt level 7
Trap #5
Trap #6
Trap #7
Trap #8
Trap #9
Trap #10
Trap #11
Trap #12
Trap #13
Trap #14
Trap #15
```

And then tell QDOS to use this table for your job. Any exceptions that are generated and that are mentioned above will be handled by your own routine. Of all of these, the address error needs to have special treatment because it has the extra data on the stack.

The problem being that if your instruction caused an error, what happens when you handle the exception and RTE - does the program fail again because it tried to execute the same instruction again? Sometimes is the only answer.

The following code will be very useful when you first start writing assembler as it will trap the exceptions mentioned above and attempt to allow you to carry on. This example should be run on a 68000 or 68008 ONLY. I do not have the data for exception handling on a 68020 or above so Gold Cards, Super Gold Cards etc may cause problems. I don't know.

6.6 The Exception Handler Code.

```
*============================================================
* This code adds a 'protective barrier' to the QL so that silly
* programming errors can be intercepted and hopefully handled
* before the QL crashes out.
*
* This code should only be run on a 68000 or 68008 as the
* exception stack frame is (probably) different on 68010 and
* above.
*
* Copyright (c) Norman Dunbar 1999 but permission for unlimited
* use and abuse is given!
*============================================================

```

```
start       lea  exceptions,a1      ; Table of exceptions - empty
            lea  x_address,a2       ; Address exceptions
            move.l a2,(a1)+         ; Save in table

            lea  x_illegal,a2       ; Illegal exceptions
            move.l a2,(a1)+         ; Save in table

            lea  x_divide,a2        ; Divide by zero
            move.l a2,(a1)+         ; Save in table

            lea  x_check,a2         ; CHK instruction
            move.l a2,(a1)+         ; Save in table

            lea  x_trapv,a2         ; TRAPV instruction
            move.l a2,(a1)+         ; Save in table

            lea  x_priv,a2          ; Privilege violation
            move.l a2,(a1)+         ; Save in table

            lea  x_trace,a2         ; Trace exception
            move.l a2,(a1)+         ; Save in table

            lea  x_int_7,a2         ; Interrupt level 7
            move.l a2,(a1)+         ; Save in table

            lea  x_trap,a2          ; TRAP #5 to TRAP #15
            moveq #10,d0            ; 11 entries to fill

trap_loop   move.l a2,(a1)+         ; Save one in table
            dbra d0,trap_loop       ; Then do the rest

            lea  exceptions,a1      ; Exceptions table, now full
            moveq #-1,d1            ; Job id = 'this job'
            moveq #mt_trapv,d0      ; Set exception table
            trap #1                 ; And do it
            rts                     ; Exit to SuperBasic

*===============================================================
* Now the actual exception handlers themselves. Apart from the
* ADDRESS exception, all have 3 words on the stack when called.
*===============================================================

x_address   lea  t_address,a1     ; Message to print
            bsr  message_0        ; Print the message
            addq.l #8,A7          ; Tidy extra data off the stack
            rte                   ; Attempt to continue

t_address   dc.w 15
            dc.b 'ADDRESS error.'
            dc.b 10

x_illegal   lea  t_illegal,a1     ; Message to print
            bsr  message_0        ; Print the message
            addq.l #2,2(a7)       ; Don't do the instruction again!
            rte                   ; Attempt the next instruction
```

```
t_illegal   dc.w 21
            dc.b 'ILLEGAL instruction.'
            dc.b 10


x_divide    lea t_divide,a1         ; Message to print
            bsr message_0           ; Print the message
            rte                     ; Attempt to carry on

t_divide    dc.w 16
            dc.b 'DIVIDE BY ZERO.'
            dc.b 10


x_check     lea t_check,a1          ; Message to print
            bsr message_0           ; Print the message
            rte                     ; Attempt to carry on

t_check     dc.w 17
            dc.b 'CHK instruction.'
            dc.b 10


x_trapv     lea t_trapv,a1          ; Message to print
            bsr message_0           ; Print the message
            rte                     ; Attempt to carry on

t_trapv     dc.w 19
            dc.b 'TRAPV instruction.'
            dc.b 10


x_priv      lea t_priv,a1           ; Message to print
            bsr message_0           ; Print the message
            rte                     ; Attempt to carry on

t_priv      dc.w 21
            dc.b 'PRIVILEGE VIOLATION.'
            dc.b 10


x_trace lea t_trace,a1              ; Message to print
            bsr message_0           ; Print the message
            rte                     ; Attempt to carry on

t_trace     dc.w 25
            dc.b 'TRACE - not implemented.'
            dc.b 10


x_int_7 lea t_int_7,a1              ; Message to print
            bsr message_0           ; Print the message
            rte                     ; Attempt to carry on

t_int_7     dc.w 26
            dc.b 'DO NOT PRESS CTRL ALT 7!'
```

```
                dc.b  10


x_trap          lea  t_trap,a1          ; Message to print
                bsr  message_0          ; Print the message
                rte                     ; Attempt to carry on

t_trap          dc.w 39
                dc.b 'TRAP #5 to TRAP #15 - not implemented.'
                dc.b 10


*=====================================================================
* This routine prints a message to channel 0. The message is at
* 0(A1) in the usual QDOS format of a size word followed by the
* text. The UT_ERR0 routine expects an error code in D0 -or- the
* address of a user defined error message in D0 with bit 31 set
* to show that it is user defined.
*=====================================================================
message_0       move.l a1,d0            ; User defined message address
                bset #31,d0             ; Mark it as such
                move.w ut_err0,a2       ; Vectored routine
                jsr (a2)                ; Print the exception message
                moveq #0,d0             ; Ignore any errors
                rts

exceptions      ds.l    19              ; Space for 19 exception handlers

*=====================================================================
* HERE ENDETH THE CODE.
*=====================================================================
```

Listing 6.1: Exception Handler for the QL

6.7 How it Works.

Now that you have typed the above code into a file, I shall explain what is happening. The code begins at the label 'start' and sets A1 to the address of the label 'exceptions' within the program. This is where the `LEA` instruction is useful - when writing position independent programs. These are programs that can be run at any address and are a requirement if you want to write good QDOS programs.

The 'exceptions' label identifies the start of the 19 long words of data that hold the addresses of the 19 redefined exception vectors as detailed above. At the moment the table contains random garbage and needs to be initialised *before* we tell QDOS to use the new vectors.

The address of the routine to handle address exceptions, 'x_address', is loaded into A2 - again using position independent methods, and then placed in the table at the first location. You will note that 'address register with post-increment' addressing is used here. This means that A1 is automatically incremented by the correct amount - 4 in the case of the long sized move - ready for the next vector to be loaded.

This process is repeated for the illegal, divide by zero, `CHK`, `TRAPV`, privilege violation, trace and interrupt level 7 vectors.

There are 11 vectors left in the table for `TRAP #5` through to `TRAP #15`. Rather than give each of these an individual handler, we point them all to the same one as we intend to ignore these instructions when they occur. To set these 11 vectors up, we run through a small loop which counts D0 down from 10 to -1 setting the vector for each of the 11 TRAP exceptions to be the single routine at address x_trap.

Our exceptions table has now been defined and all we have to do is tell QDOS that we want to use it. Once again, A1 is set to the start address of the exceptions table as required by QDOS, D1 is then set to -1 which implies 'the current job' to QDOS. This is used in many of the QDOS routines which require a job ID, passing -1 means 'me'. As we are executing this code directly from SuperBasic, that is what the current job will be. Once the vectors have been set up for any job, all other jobs created by it will use the same vector table.

This means that as the initiating job is SuperBasic, and as most other jobs are created by SuperBasic, this means that we have effectively created a protection mechanism for every job in the system created *from this point onwards*! If this is the first code loaded on your system, then every single job created will be protected by this code.

Trap #1 is called with D0 set to the value `MT_TRAPV` - a fancy way of saying 7 - and we return to SuperBasic with any error codes that may arise. As there appears to be only 'invalid job' returned, it is unlikely that there will be any as we are using the current job's own id.

Now that the initialisation has been carried out, the exception handlers will just sit there until such time as they are activated.

Most of the handler code is the same - we simply trap the exception, print a warning message to channel #0 and attempt to carry on - but the Address and illegal exception handlers do additional processing.

In the case of an address error, there is an extra 8 bytes of data on the stack on top of the 'standard' stack frame as discussed above. These need to be cleared off before we execute the `RTE` instruction.

Warn

this is only true of a QL with 128K or a Trump Card etc. If you use QXL or some other card with an upgraded processor, then the stack is different and this code won't work properly.

An Illegal instruction also manipulates the stack, but this time, it adds 2 to the address of the failed instruction. This prevents it from trying to execute it again when we exit the routine. Of course this may not always be successful and can cause further errors along the way - if the instruction was followed by a word of data for example. Trying to execute the data could lead to another exception and so on. What would you rather have, a message telling you about it or a lock up with no indications?

The messages are defined in the standard QDOS manner of a size word followed by the bytes of the message. The appropriate message has its address loaded into A1 by the exception handler, and a branch is made to the sub-routine MESSAGE_0 which will attempt to display a message to channel #0. If this fails, it will try #1 before giving up.

If you have a QDOS manual and you look up `UT_ERR0` (that's a zero by the way!) you will see that it takes an error code in D0 as its only parameter. We are using it slightly differently as we are defining our own messages and not using the Sinclair defined ones such as 'invalid channel id' or 'bad parameter' etc.In order to do this, we load D0 with the address of the message but set bit 31 of D0 so that QDOS knows that it is an address and not an error code.

The UT_ERR0 routine lives in the ROM somewhere, I don't know where it lives in all ROMs as it could have been moved between ROM releases. Because of this, there is a vector table in the

ROM at a standard position. To get the address of the routine, we simply read the contents of the vector table into an address register and `JSR` to that address. (This will be explained later in the series when I cover QDOS).

So now that we have assembled the code all we do is LRESPR it (or RESPR(512), LBYTES and then CALL) and that is it. Whenever any exceptions occur, the above code will handle them and, most importantly, tell you what has happened. Your QL may still be hung - but at least you should know why!

6.8 Coming Up...

The is the end of quite a complex section on exceptions and their handling. If you have stuck with me this far, the rest should be quite a lot simpler - well, at least until we get to the graphics stuff that is :o)

However, the next chapter delves into the features of QDOS that allow the humble programmer the ability to write their own assembly language procedures and functions to extend the workings of SuperBasic. Along the way, the very important maths stack will be examined in quite a lot of detail

7. Extending SuperBasic

7.1 Introduction

This time we are looking at extending SuperBasic by adding extra procedures and functions which can be loaded once after boot up and then used by any SuperBasic program that we load or type in afterwards.

Along the way, we will have to take a look at the manner in which we do the following:

- link assembler extensions (procedures and functions) into SuperBasic
- fetching parameters
- testing separators (eg the '#' before a channel number etc)
- the maths stack - and all its problems
- returning values for functions
- accessing the SuperBasic channel table

7.2 Linking To SuperBasic

When you have written some code that defines a new SuperBasic procedure or function, you must tell SuperBasic what it is called and where it lives. There is a vectored routine to do this and it is called `BP_INIT` (in QDOS) or `SB_INIPR` (in SMSQ). As an old hand at QDOS, I still use the QDOS definitions and names. As we are all using George Gwilt's GWASL assembler, and it uses the QDOS names, we shall continue to do so in this series.

Start up the QED editor (or your favourite) and type the following in:

```
start   lea     define,a1       ; Pointer to the definition table
        move.w  BP_INIT,a2      ; The vector we need to use
        jsr     (a2)            ; Call the vectored routine
        rts                     ; Return to SuperBasic
```

Listing 7.1: Linking Extensions to SuperBasic

You will note that we only execute a small stub of code. This is simply because we are linking the new routines into SuperBasic and the actual code for the routines will be executed when a SuperBasic program uses one of the new routines. All will become clear.

The definition table required by `BP_INIT` has to be in the format shown in Table 7.1 and it must start at an even address. A1.L points at the table when `BP_INIT` is called:

Size	**Purpose**
word	How many new procedures (A1 points here)
	Repeat for each procedure:
word	Offset to code start for this procedure
byte	How many bytes in the procedure name
bytes	The procedure name
word	Zero = end of procedures
word	How many new functions
	Repeat for each function:
word	Offset to code start for this function
byte	How many bytes in the function name
bytes	The function name
word	Zero = end of functions & table

Table 7.1: Definition Block For BP_INIT

As an example, our code file will introduce 1 new procedure and the definition table will be set up like the following which you should now type into the editor following on from the code that is already there :

```
define  dc.w    1                   ; 1 new procedure
        dc.w    psi_cls-*
        dc.b    7,'PSI_CLS'
        dc.w    0                   ; End of procedures

        dc.w    0                   ; Number of functions
        dc.w    0                   ; End of functions
```

Listing 7.2: Example Extension Parameter Table

Notice that the format of the procedure name is slightly different from normal QDOS string in that the size of the name is stored in a *byte* and not in a word.

Now then, there is a caveat - isn't there always? If the average length of the names of all the procedures, or functions, is greater than 7 then the simple word for the number of procedures or functions is changed to the value given by this calculation:

(total number of characters in proc names + number of procedures + 7)/8

Checking our table above we have a total of 7 characters in the procedure name and there is 1 new procedure. This gives an average of 7 characters per name (round up always!) so we are ok.

And that is it. On QL's of JM vintage and below, the machine must be NEW'd before you can use them. On JS and above, this need not be done.

Once a set of procedures and/or functions has been linked into SuperBasic, the definition block is no longer required. If your code requires the use of some workspace, then you can use the definition table. Just make sure that you don't use more bytes that there are available !

So, let's write our first procedure.

7.3 Procedures

Procedures in assembly are very much like PROCedures in SuperBasic. For example, consider the following:

```
1000 DEFine PROCedure PSI_CLS(chan%, P%, S%, I%)
1010   PAPER #chan%, P%
1020   STRIP #chan%, S%
1030   INK #chan%, I%
1040   CLS #chan%
1050 END DEFine PSI_CLS
```

This simple routine is probably at the heart of many SuperBasic programs and is called like this:

```
100 PSI_CLS 1, 2, 4, 0
```

To give channel #1 red paper, green strip and black ink. Assembler procedures are very similar and in fact we shall now dive straight in and convert the above into assembler.

Back into the QED editor with the code from the start of this article typed in. We have so far typed the code to link the new procedure and the definition block for the new procedure, now we need to write the code for the procedure itself. Your file should look like this so far :

```
start   lea     define,a1       ; Pointer to the definition table
        move.w  BP_INIT,a2      ; The vector we need to use
        jsr     (a2)            ; Call the vectored routine
        rts                     ; Return to SuperBasic

define  dc.w    1               ; 1 new procedure
        dc.w    psi_cls-*
        dc.b    7,'PSI_CLS'
        dc.w    0               ; End of procedures

        dc.w    0               ; Number of functions
        dc.w    0               ; End of functions
```

Listing 7.3: PSI_CLS Definition Table

In the definition table there is an offset word to the start address of the new procedure. Ours is defined like this:

```
        dc.w    psi_cls-*
```

Which is a useful way to get the assembler to calculate the offset for us. The '*' is assembler short-hand for 'where I am now' or 'the current address'. Our example uses the label psi_cls so our code has to start there.

On with the procedure. In assembler you must take great care to ensure that you have enough parameters etc (see below) and that they are all the correct type. In this example, we will get using integer parameters but the first one must have a hash (#) in front of it. Of course, when using INK, PAPER etc in SuperBasic, you can default the channel number and #1 will be used instead. This means that the following statements are equivalent:

```
2000 PAPER #1,2
2010 PAPER 2
```

It would be nice if our PSI_CLS routine did a similar thing so that the following was equivalent:

```
2500 PSI_CLS #1, 2, 2, 0
2510 PSI_CLS 2, 2, 0
```

This turns out to be quite easy to do.

Here then, is a list of what our procedure must do:

- Count how many parameters were supplied. There must be 3 or 4.
- If 4 parameters supplied, check that the first parameter has a hash in front of it.
- Fetches all parameters onto the maths stack.
- Convert parameters from maths stack to registers & validates them.
- Set the paper, strip and ink colours for the correct channel, defaulting to #1 as appropriate, if only 3 parameters were supplied.
- Clear the channel using CLS.
- Returns to SuperBasic.
- Abort nicely whenever it detects an error.

Type the following after the definition block:

```
err_bp      equ     -15             ; Bad parameter error
err_no      equ     -6              ; Channel not open
bv_chbas    equ     $30             ; Offset to channel table
bv_chp      equ     $34             ; Offset to channel table end
bv_rip      equ     $58             ; Maths stack pointer

psi_cls     move.l  a5,d7           ; End of parameters
            sub.l   a3,d7           ; Minus start of parameters
            divu    #8,d7           ; How many parameters?
            cmpi.w  #3,d7           ; Defaulting channel id?
            beq.s   hash_ok         ; yes, skip hash check

*------------------------------------------------------------------
* We do not have 3 parameters so test for 4 and if not found,
* error exit. If we do have 4 then the first must have a hash in
* front.
*------------------------------------------------------------------
hash_check  cmpi.w  #4,d7           ; We need  4 parameters
            bne.s   error_bp        ; Oops!
            btst    #7,1(a6,a3.l)   ; Is there a # before p1?
            beq.s   error_bp        ; No, we reject it then

hash_ok     move.w  ca_gtint,a2     ; We want word integers
            jsr     (a2)            ; Fetch them all
            tst.l   d0              ; Did it work?
            beq.s   got_ok          ; Yes it did
```

```
               rts                          ; Bale out otherwise

*------------------------------------------------------------------------
* We expected to get 3 or 4 parameters and should have, but now
* that we have got them, check to make sure we have received that
* which we expected to.
*------------------------------------------------------------------------
got_ok         cmpi.w   #4,d3               ; Were there 4 of them?
               beq.s    got_4               ; Yes

               cmpi.w   #3,d3               ; Maybe default channel in use
               beq.s    got_3               ; So that is ok too

error_bp       moveq    #err_bp,d0          ; Bad Parameter error code
error_exit     rts                          ; Bale out with error

*------------------------------------------------------------------------
* We have 4 parameters, so fetch the channel id into D0 - this is
* the first of the parameters. We need to tidy the maths stack as
* well so that get_rest works correctly regardless of whether we
* have 3 or 4 parameters.
*------------------------------------------------------------------------
got_4          move.w   0(a6,a1.l),d0        ; Get channel id
               bmi.s    error_bp            ; We don't like -ve channels
               adda.l   #2,a1               ; Tidy stack pointer
               bra.s    get_rest            ; Skip default channel id bit


*------------------------------------------------------------------------
* At this point we default the channel being used to #1. By
* moving one to D0 and processing as normal, we can do this
* without much effort.
*------------------------------------------------------------------------
got_3          moveq    #1,d0               ; Default channel is #1

*------------------------------------------------------------------------
* Here convert the SuperBasic channel number in D0 into an
* internal id in A0 and bale out if it fails, or if the channel
* is not open or has been closed - there is a difference.
* A closed channel has a negative id while a channel not yet
* opened is not in the table.
*------------------------------------------------------------------------
get_rest       bsr      channel_id          ; Convert D0->QDOS id in A0.L
               bne.s    error_exit          ; Bale out if errors
```

Listing 7.4: PSI_CLS - The Final Version - Part 1

At this point we have (A6,A1) pointing to the paper parameter on the stack and A0.L holding the channel id for the requested channel (or the default of #1). Now we can set the paper colour (which does not set the strip like SuperBasic does!)

Looking at the QDOS documentation for SD_SETPA and the others, we see that A1 is 'undefined' on return from the routine. This is bad so we need to preserve it across calls or we can fetch all the parameters first. Registers D4 to D7 are not mentioned in the documentation so they are preserved/not used by the routines so we shall fetch the parameters into these registers first of all and this way we can also validate them for errors.

```
*------------------------------------------------------------------------
* Because we tidied the stack pointer in A1 when we fetched the
* channel id, the following code expects to see the paper colour
* at 0(A6,A1) and this is the same as if we never were supplied
* with a channel id in the first place - cunning stuff eh?
*
* Fetch the remaining 3 parameters into registers that will not
* be trashed by the QDOS routines that set the paper, strip and
* ink. We reject any parameter which is negative as we don't deal
* with negative colours and just in case, we also mask out the
* high work of the parameter to ensure it is in range 0 to 255.
*
* NOTE: we could do away with the negative check and just mask.
* This would in effect convert from a negative to a positive
* number - but this is the real world (?) and we have to perform
* parameter validation.
*------------------------------------------------------------------------
            move.w  0(a6,a1.l),d4   ; Paper in D4
            bmi.s   error_bp        ; Negative is bad news
            andi.w  #$00ff,d4       ; Force range 0 - 255

            move.w  2(a6,a1.l),d5   ; Strip in D5
            bmi.s   error_bp        ; Negative is bad news
            andi.w  #$00ff,d5       ; Force range 0 - 255

            move.w  4(a6,a1.l),d6   ; Ink in D6
            bmi.s   error_bp        ; Negative is bad news
            andi.w  #$00ff,d6       ; Force range 0 - 255

            adda.l  #6,a1           ; Tidy the stack

            moveq   #sd_setpa,d0    ; Paper trap code
            move.w  d4,d1           ; Paper colour
            moveq   #-1,d3          ; Infinite timeout
*                                   ; Channel id is still in A0
            trap    #3              ; Set the paper
            tst.l   d0              ; OK?
            bne.s   error_exit      ; No bale out

*------------------------------------------------------------------------
* Now the paper has been set, and the documentation says that A0
* is preserved along with D3, we can set the strip colour now.
*------------------------------------------------------------------------
            moveq   #sd_setst,d0    ; Strip trap code
            move.w  d5,d1           ; Strip colour
            trap    #3              ; Set the strip
            tst.l   d0              ; OK?
            bne.s   error_exit      ; No bale out

*------------------------------------------------------------------------
* Now the strip has been set, and the documentation says that A0
* is preserved along with D3, we can set the ink colour now.
*------------------------------------------------------------------------
            moveq   #sd_setin,d0    ; Ink trap code
            move.w  d6,d1           ; Ink colour
            trap    #3              ; Set the Ink
```

```
                tst.l    d0                 ; Ok?
                bne.s    error_exit         ; No bale out

*------------------------------------------------------------------------
* And finally, we can CLS the screen.
*------------------------------------------------------------------------
                moveq    #sd_clear,d0       ; CLS whole screen
                trap     #3                 ; Do it
                bra.s    error_exit         ; All done


*------------------------------------------------------------------------
* This routine takes a SuperBasic channel number in D0 and
* converts it into a QDOS internal channel id in A0. If the
* channel is closed or not yet opened, the routine returns D0 =
* ERR_NO and A0 is invalid.
* D0 will be zero if all is ok.
*------------------------------------------------------------------------
channel_id      mulu     #$28,d0             ; Offset into channel table
                add.l    bv_chbas(a6),d0 ; Add table start address
                cmp.l    bv_chp(a6),d0      ; Valid?
                bge.s    ch_bad             ; No, channel # off end
                move.l   0(a6,d0.l),d0      ; Channel id
                bmi.s    ch_bad             ; Channel closed
                move.l   d0,a0              ; We need id in A0
                moveq    #0,d0              ; No errors
                rts                         ; Finished

ch_bad          moveq    #err_no,d0         ; Channel not open (-6)
                rts                         ; Bale out
```

Listing 7.5: PSI_CLS - The Final Version - Part 2

Save the file and assemble it using GWASL. Once all errors have been sorted out, either LRESPR it or ALCHP/LBYTES/CALL in the normal manner. If you have a JM and below, type NEW then try this:

```
PSI_CLS #1, 2, 4, 0  (or PSI_CLS 2, 4, 0)
```

And see what happens when you

```
PRINT 'Hello world' (or PRINT #1, 'Hello world')
```

If you have a JS or above, then just try it without the NEW.

You should see the words ‘Hello world’ written in black, on a green strip on red paper - assuming your display can handle the colour mixture !

In the code, you will notice that whenever I detect an error, I simply return to SuperBasic with the error code in D0. This doesn’t look very friendly does it? Actually, QDOS is very friendly when it comes to procedures because in the event of an error, QDOS will do all the tidying up that we need to do so we don’t have to worry about it. This is discussed below in Section 7.4 Functions and in Section 7.8 The Maths Stack.

7.4 Functions

Wouldn't it be nice to do this instead of the above:

```
PSI_CLS #1, RED, GREEN, BLACK
```

In SuperBasic this would be done either by:

```
DEFine FuNction RED
    return 2
END DEFine RED

DEFine FuNction GREEN
    return 4
END DEFine GREEN

DEFine FuNction BLACK
    return 0
END DEFine BLACK
```

OK, I know it could be done like this:

```
RED = 2
GREEN = 4
BLACK = 0
```

but we are dealing with machine code functions and this is more illustrative of what we are about to do. (So there!)

We shall now extend our original example so that we can specify colour values by name - this is much more friendly in my opinion.

The following two lines in the definition block need to be removed :

```
        dc.w    0               ; Number of functions
        dc.w    0               ; End of functions
```

Listing 7.6: Colour Functions

And replaced by the following:

```
        dc.w    8               ; There are 8 functions

        dc.w    black-*         ; First function
        dc.b    5,'BLACK'

        dc.w    blue-*          ; Second function
        dc.b    4,'BLUE'

        dc.w    red-*           ; Third function
        dc.b    3,'RED'

        dc.w    cyan-*          ; Fourth function
        dc.b    4,'CYAN'
```

```
        dc.w    green-*         ; Fifth function
        dc.b    5,'GREEN'

        dc.w    magenta-*       ; Sixth function
        dc.b    7,'MAGENTA'

        dc.w    yellow-*        ; Seventh function
        dc.b    6,'YELLOW'

        dc.w    white-*         ; Eighth function
        dc.b    5,'WHITE'

        dc.w    0               ; End of functions
```

Listing 7.7: Colour Functions

The following is the code for the new functions, type it into the file after the end of the 'channel_id' subroutine:

```
black       moveq   #0,d7
            bra.s   return_d7

blue        moveq   #1,d7
            bra.s   return_d7

red         moveq   #2,d7
            bra.s   return_d7

magenta     moveq   #3,d7
            bra.s   return_d7

green       moveq   #4,d7
            bra.s   return_d7

cyan        moveq   #5,d7
            bra.s   return_d7

yellow      moveq   #6,d7
            bra.s   return_d7

white       moveq   #7,d7

*----------------------------------------------------------------
* This routine returns the word value in d7 to SuperBasic as the
* result of the function we are running. It requires two bytes on
* the top of the maths stack and because there were no parameters
* supplied to any of the functions, I can safely ask QDOS for
* these two bytes.
*----------------------------------------------------------------
return_d7   move.l  bv_rip(a6),a1     ; Because we had no params
            moveq   #2,d1             ; Two bytes required
            move.w  bv_chrix,a2       ; Allocate maths stack space
            jsr     (a2)              ; Go get some space.
*                                     ; No errors are returned.

*----------------------------------------------------------------
```

```
* The maths stack has been extended by two bytes BUT it may have
*  moved around in memory so we need to get the stack pointer
* into A1 again.
*------------------------------------------------------------------------
            move.l   bv_rip(a6),a1    ; New top of stack
            subq.l   #2,a1            ; Space for our result
            move.w   d7,0(a6,a1.l)    ; Stack the result
            move.w   #3,d4            ; Signal word result on stack
            move.l   a1,bv_rip(a6)    ; Store new top of stack
            clr.l    d0               ; No errors
            rts                       ; Return result to SuperBasic
```

That is the end of the code. Assemble it, debug it and test it using the following:

```
PAPER GREEN
STRIP RED
INK BLACK
CLS
PRINT "Hello world"
```

or, if you like:

```
PSI_CLS GREEN, RED, BLACK
PRINT "Hello world"
```

In the procedure, PSI_CLS, we obtained some parameters for the various colours and channels. I shall now discuss how this is done in much more detail.

7.5 Getting Parameters

On entry to a machine code extension (ie not an EXEC'd job or a CALLed routine) certain registers are set up with very useful values. These are shown in Table 7.2.

Register	Value
A1	*Allegedly* points to the top of the maths stack relative to A6, however, see below.
A3	Points to the start of the name table entry for the first parameter.
A5	Points to the first byte *after* the name table entry for the last parameter.
A6	Base address of SuperBasic. Do not change this register.

Table 7.2: Register Settings On Entry To SuperBasic Extensions.

A1 is supposed to point at the top of the maths stack (see below) relative to A6, but I have found out the hard way that this is only the case when the procedure or function being executed has some parameters and they have been fetched. A1 is set to the amount of space used (or free) on the maths stack on entry to a procedure. (See Maths Stack below for full details.)

A3 points at the address of the first byte of the first entry in the name table for this procedure or function. Again, this is relative to A6.

A5 points at the address of the first byte *after* the last name table entry for this procedure or function. Again this is relative to A6.

A6 should never be changed as it points to the base of the SuperBasic job and almost all the various routines involving the maths stack and getting/returning parameters rely on addresses being relative to A6.

So we can now check to see how many parameters we have by the following calculation:

$(A5-A3)/8$

There are 8 bytes in each name table entry. Full details of the name table entries are given below.

If we have 3 parameters, then the name table entries will look like Figure 7.1:

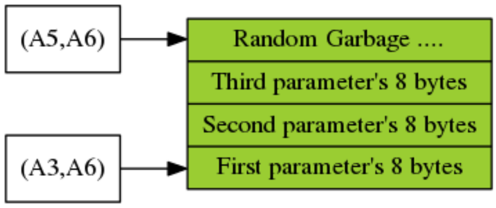


Figure 7.1: Name Table Entries for Three Parameters.

So (A3,A6) points to the first byte of the first parameter and is the lowest address, (A5,A6) points to the first byte past the last parameter and is the highest address.

The first name table entry starts at 0(A3,A6) and ends at 7(A3,A6). The second starts at 8(A3,A6) and ends at 15(A3,A6) and the last starts at 16(A3,A6) and stops at 23(A3,A6).

When fetching parameters from the name list onto the maths stack, we can use some vectored utilities to get them for us. These allow the retrieval of strings, long words, integers (short words) and floating point values. They all expect A3 and A5 to be set up correctly as above. A3 and A5 are trashed by the routines, so if you have to check any parameter separators etc, then you must do it before calling the fetch routines.

When the routines return, they set D3.W to the number of parameters fetched and set A1 to the correct value for the top of the maths stack - relative to A6 of course. Now we can access the values of each parameter separately as we like. On return the first parameter in the list is stored at 0(A1,A6), the next is above the first and so on. When fetching parameters p1, p2, p3 from a procedure or function call, they will end up on the maths stack in the correct order - 0(A1,A6) will be pointing at p1 on the stack.

The parameter fetching routines are listed in Table 7.3.

Vector	**Purpose**
`CA_GTINT`	fetch integer parameters (2 bytes each).
`CA_GTLIN`	fetch long parameters (4 bytes each).
`CA_GTFP`	fetch floating point parameters (6 bytes each).
`CA_GTSTR`	fetch string parameters (variable length).

Table 7.3: Vectored Routines For Parameter Fetching.

They require to be called as follows:

```
start   move.w  ca_gtint,a2      ; Fetch all params as word ints
        jsr     (a2)             ; Do it
        tst.l   d0               ; Did it work?
        beq.s   ok               ; Yes
        rts                      ; Return to SuperBasic

ok      ...                      ; carry on here
```

Listing 7.8: Using the Vectored Parameter Fetching Utilities

At this point, D3.W can be tested to check that the correct number of parameters has been fetched.

```
start   cmpi.w  #4,d3            ; Were there 4 parameters?
        beq.s   ok_4             ; Yes
        moveq   #-15,d0          ; Bad parameter error code = -15
        rts                      ; and back to SuperBasic

ok_4    ...                      ; Carry on here
```

Listing 7.9: Checking Parameter Counts

To access the parameters we need to get the data off of the maths stack and into our working registers, as follows:

```
        move.w  0(a6,a1.l),d1    ; Parameter one
        move.w  2(a6,a1.l),d2    ; Parameter two
        move.w  4(a6,a1.l),d3    ; Parameter three
        move.w  6(a6,a1.l),d4    ; Parameter four
```

Listing 7.10: Fetching Parameter Values

and so on. Now that we have our parameters, we need do nothing more with the maths stack if we are inside the code of a procedure. If we are in a function then we *must* tidy the maths stack. This is simply done by adding the size of all parameters on the stack to A1. In our example we have 4 word length parameters, so we should add 8 to A1 as follows :

```
        adda.l  #8,a1            ; Reset maths stack
```

As mentioned, there is no need to do this in a procedure, but if you have to learn to do it for a function, you are as well to learn to do it for everything - that way you don't forget to do it and cause a hanging QL.

Tidying a stack with strings on is more difficult and it is probably best done as each one is removed. For example, say we have two strings on the stack after a call to `CA_GTSTR` then we get them off as follows :

```
    cmpi.w  #2,d3            ; Were there two strings?
    beq.s   ok               ; Yes
    moveq   #-15,d0          ; Bad parameter
    rts                      ; Exit to SuperBasic

ok  lea     buffer_a,a2      ; Destination for one string
    lea     0(a6,a1.l),a3    ; Source for string
    bsr     copy_str         ; Copy
    move.w  0(a6,a1.l),d0    ; Size word
```

```
    addq.w  #3,d0           ; Make bigger
    bclr    #0,d0           ; Make even
    add.w   d0,a1           ; This will sign extend remember!
```

Listing 7.11: Tidying a String from the Maths Stack - Part 1

Ok, so we added the size of the first string plus 2 for the size of the size word as well, to A1 having made it even so the stack is now cleared of the first string. This leaves one string with its size word sitting at 0(A6,a1.l) ready for the next copy:

```
    lea     buffer_b,a2     ; Destination for next string
    lea     0(a6,a1.l),a3   ; Source for string
    bsr     copy_str        ; Do the copy
    move.w  0(a6,a1.l),d0   ; Size word
    addq.w  #3,d0           ; Make bigger
    bclr    #0,d0           ; Make even
    add.w   d0,a1           ; This will sign extend too!
```

Listing 7.12: Tidying a String from the Maths Stack - Part 2

and there you have a tidy stack once again.

You could ask ‘if we have to restore A1 to its value on entry, why not just save A1 and then restore it afterward?’. Like this:

```
start   move.l  bv_rip(a6),a1   ; Fetch top of Maths Stack
        move.l  a1,-(a7)        ; Stack it for later

        ; Do lots of stuff here - fetching parameters etc

        move.l  (a7)+,a1        ; Restore A1

        ; and so on
```

Listing 7.13: How to Hang the QL

Well, you could, but at certain times there will be a hung QL and you will not know why. The reason is simple, but difficult to find or trace. When you fetch parameters onto the maths stack, it can *move around in memory*. Preserving the original value is fine if the stack stays put, but if it moves and you set BV_RIP to the old value, you can get into all sorts of trouble. It is best to keep the stack tidy using the methods described above.

7.5.1 Keeping Things Even

You may well also ask “What is all this add 3 and clear bit 0 nonsense then?” Think about it in binary for a bit. We have the word size of the string in D0.W and we must ensure that we add an even number of bytes to A1. We must also remember to add 2 to A1 for the size of the size word itself.

Lets try this with an even number first of all. Even numbers are detected by bit zero being clear, so:

So you can see what is happening. D0 always ends up being D0 + 2 and is always even. This is good as it is what we want. What about odd numbers then?

So is this good then? Remember that the maths stack must be kept even. When odd length strings are copied onto it by `CA_GTSTR` it pads out the space on the stack with a rubbish byte (CHR$(0) to be precise) which is never used. The size word remains odd.

D0	D0 + 3	Result
2	5	4
4	7	6
10	13	12

Table 7.4: Keeping even numbers even.

D0	D0 + 3	Result
3	6	6
5	8	8
11	14	14

Table 7.5: Keeping odd numbers even.

So for an odd sized string we need to add 2 for the size word, the odd number of bytes and one spare for the padding. Our 3 lines of code handle this for all cases - even or odd sized strings. The code is good!

Of course it would be simple to do this:

```
        move.w  0(a6,a1.l),d0     ; Size word
        btst    #0,d0             ; Is it even?
        beq.s   even              ; Yes
        addq.w  #1,d0             ; Add 1 for padding byte

even    addq.w  #2,d0             ; Add 2 ro the size word
        add.w   d0,a1             ; And add with sign extension
```

Listing 7.14: Long Way to Keep Things Even

But this is extra typing and takes longer, so the simple case shown above, works all the time.

7.5.2 Two Of These And One Of Those Please

What do you do if you want to get hold of two long words and a string?

Let us assume that you are writing an extension procedure that has this format:

```
DO_SOMETHING long_1, long_2, string_1
```

This has two different types of parameters and we cannot fetch them all in one go unless we can read the long parameters as strings and convert them ourselves. It is quite easy to fetch these parameters - you just do it in two goes.

In the code we know that A3 and A5 hold the start and stop addresses of the parameters in the Name Table. If we set A5 to be A3 + 16 and then collect long words we will get our two long words. We can then set A5 back to its original value and set A3 to this less 8 and fetch the final parameter as a string. Here we go then:

```
get_longs   move.l  a5,-(a7)        ; Save last parameter pointer
            lea     16(a3),a5       ; Set A5 for two parameters
            move.w  ca_gtlint,a2    ; Fetch all (2) longs
```

```
                jsr     (a2)            ; Do it
                tst.l   d0              ; OK?
                beq.s   got_long        ; Yes
                rts                     ; Exit with error code

got_long        cmpi.w  #2,d3           ; Were there two?
                bne.s   bad_params      ; No, bale out
                move.l  (a7)+,a5        ; A5 holds address of p3
                lea     -8(a5),a3       ; There can be only one!
                move.w  ca_gtstr,a2     ; Fetch as strings now
                jsr     (a2)            ; Do it
                tst.l   d0              ; OK?
                beq.s   got_string      ; Yes
                rts                     ; Exit with error code

bad_params      moveq   #-15,d0         ; Bad parameter error
                rts                     ; Exit to SuperBasic

got_string      ; continue from here
```

Listing 7.15: Fetching Mixed Type Parameters

Ok, so now what does the maths stack look like? Remember when fetching parameters they end up on the stack in the order you want them with the first at the lowest address and the next above it and so on. This time, we fetched two longs and a string in two different calls. This means that after the first fetch the maths stack looks like Figure 7.2:

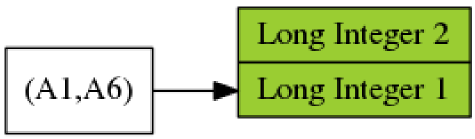


Figure 7.2: Maths Stack After Fetching Two Long Integer Parameters.

But then we fetched a string and it got put onto the maths stack so it now looks like Figure 7.3:

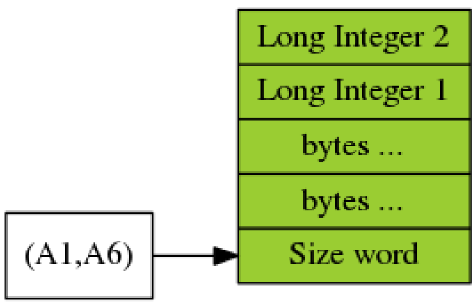


Figure 7.3: Previous Maths Stack After Fetching a String Parameter.

QDOS is very helpful here. If during the course of fetching the string, the maths stack had to be moved in memory, QDOS will preserve the current contents so that 'long_1' and 'long_2' will still be there when you come around to using their values. Nice!

In this discussion we mentioned the name table. This is discussed in detail next. Do you get the feeling that this chapter is written upside down?

7.6 Name Table Entries

The name table is a list of 8 byte entries which define all the names used in SuperBasic (or extensions to SuperBasic written in assembler), the type of each entry and where it lives in the name list and the SuperBasic variables area.

As per the description above (GETTING PARAMETERS), the name table is also used to store details of the parameters passed to our assembly routine. So for parameters passed, a copy is made and stored at the end of the name table. The A3 and A5 registers are set up to point at the first and last parameter and for these, the format of the name table is as follows:

Bytes 0 & 1	**Bytes 2 & 3**	**Bytes 4 to 7**
Type & separator flag word.	Pointer to a NAME LIST entry which *may* be an odd address.	Pointer to value in the variables area.

Table 7.6: Parameter format on the name table.

The low byte of the type word tells us what type of parameter we are dealing with and its separator(s) as shown in Table 7.7.

Bytes 0 & 1	**Bytes 2 & 3**
Bit 7	0 = There is not a hash (#) in front of this parameter
Bit 7	1 = There is a hash (#) in front of this parameter
Bits 6 - 4	000 = No separator after this parameter
Bits 6 - 4	001 = Comma (,) after this parameter
Bits 6 - 4	010 = Semi-colon (;) after this parameter
Bits 6 - 4	011 = Back-slash (\) after this parameter
Bits 6 - 4	100 = Exclamation mark (!) after this parameter
Bits 6 - 4	101 = TO after this parameter
Bits 3 - 0	0000 = Null
Bits 3 - 0	0001 = String
Bits 3 - 0	0010 = Floating point
Bits 3 - 0	0011 = Integer

Table 7.7: Parameter types and separators.

For the first parameter, the type byte is at 1(a6,a3.l) as opposed to 0(a6,a3.l).

For the rest of SuperBasic, the name table uses bytes 0 and 1 to define the type of the entry as shown in Table 7.10.

Note

The REPeat and FOR loop identifiers are hard coded to be of type floating point. This represents the internal values for SuperBasic. I suspect that this is the reason that FOR loop identifiers cannot be integer.

Byte 0 Value	Description
$00	Undefined
$01	Expression
$02	Variable
$03	Array or substring
$04	SuperBasic PROCedure (Byte 1 is always zero)
$05	SuperBasic FuNction

Table 7.8: SuperBasic specific parameter details - byte 0.

Byte 1 Value	Description
$00	Substring (Internal use only!)
$01	String
$02	Floating point.
$03	Integer

Table 7.9: SuperBasic specific parameter details - byte 1.

Bytes 0 & 1 Value	Description
$0602	REPeat loop identifier
$0702	FOR loop identifier
$0800	Assembly language procedure
$0900	Assembly language function

Table 7.10: SuperBasic specific parameter details - bytes 0 and 1 together.

SBASIC, on the other hand, under SMSQ allows integer FOR loops and I presume that the internal format for these will be $0703 - I am sure that Jochen will correct me if I am wrong!!

For all entries in the name table, be they parameters or ‘proper’ names, have a word in bytes 2 & 3 which points to the entry in the name list for this ‘name’. This simply gives an easy way of storing the names all in one place. Note that this value is simply the offset from the start of the name table where the bytes of this name can be found. A fuller description of the name list follows on below.

If the value is -1, then this is an expression and has no name.

Finally, there is a long word which is the pointer to the variables area. If this value is negative then the variable is undefined and has no entry there. Again, this value is an offset into the variables area and not an absolute address.

7.7 Name List

The name list is a simple structure in SuperBasic. It holds the names of all procedures, variables, functions etc that have ever been used in this session at the QL. It is odd in that each name is preceded by a *byte* defining its length as opposed to a word in the normal QDOS manner. This implies that names can be up to 255 characters long. There are no padding bytes to force even addresses in the name list either. Beware when accessing this area that you only do byte sized operations!

The name list starts at the address BV_NLBAS(A6) to BV_NLP(A6) with BV_NLBAS(A6) being the lowest address and BV_NLP(A6) pointing to the first byte *after* the last entry in the name list. As usual, the offsets you get from these basic variables are themselves relative to A6!

To explain further, Fetch the offsets from BV_NLBAS(A6) into A0. The address 0(A6,A0.L) is the start of the name list. Or, in code:

```
start   move.l  BV_NLBAS(a6),a0
        lea.l   0(a6,a0.l),a0
        move.b  0(a0),d0            ; D0 = size of the first entry
        ...                         ; More code here
```

Now A0 has the start of the name list, but beware of doing this in case SuperBasic gets moved. It is best to stay relative as in the following:

```
start   move.l  BV_NLBAS(a6),a0
        move.b  0(a6,a0.l),d0       ; D0 = size of the first entry
        ...                         ; More code here
```

This is much safer.

The internal structure therefore looks like Figure 7.4.

How is the name list useful to us in writing procedures and functions? consider these commands:

```
OPEN_IN #3,'ram1_test_file'
OPEN_IN #3,ram1_test_file
```

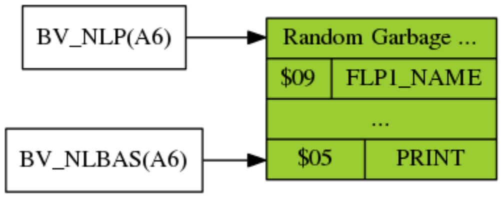


Figure 7.4: SuperBasic Name List Structure.

What is the difference? In the first case, the parameter for the filename is a quoted string and internally, the OPEN_IN routine can fetch it using `CA_GTSTR` as described above. In the second, it will fail if it uses `CA_GTSTR` because without quotes, the parameter is a NAME and not a STRING.

The procedure/function writer must check for a string parameter or a name parameter and treat each accordingly. How is this done? - use the name table type byte as described above.

In the procedure or function, process a name as follows:

Assuming that A3 points to the name table entry for this parameter, then if bits 0 to 4 of 1(a6,a3.l) is zero then we have a name and not a variable. We must copy the bytes of name, from the entry in the name list, to the stack (or to the appropriate buffer) making sure that the size byte in the name list is converted to a size word on the stack or in the buffer. The following fragment of code gives the general idea:

```
name_test    move.b   1(a6,a3.l),d0
             andi.b   #$0f,d0
             bne.s    not_name
got_name     ;
             ; Must be a name so process accordingly here
             ;
not_name     ; Process a string here
```

So when a name is detected we have to make space for it, copy the size *byte* from from the name list into the size *word* in our string buffer (which has to be word aligned on an even address) and then copy the individual bytes from the name list to the string buffer. At this point we are in the same situation we would be in had we fetched a string using `CA_GTSTR` and copied it from the maths stack into our buffer. Simple? (In my famous DJToolkit extensions I never actually bothered doing this and I simply fetched all filenames etc as strings - if the user supplied a name instead, the procedure or function complained. So far no-one has requested that it be updated to allow names!)

How about a bit of fun - lets write a procedure that prints the entire name list to a channel. It shall be called nlist and it shall take one parameter which is the channel number - this will default to #1 if no parameter supplied.

```
bv_nlbas     equ      $20                 ; Base of name list
bv_nlp       equ      $24                 ; End of name list
bv_chbas     equ      $30                 ; Base of channel table
bv_chp       equ      $34                 ; End of channel table
err_no       equ      -6                  ; Channel not open error
err_bp       equ      -15                 ; Bad parameter error

start        lea      define,a1           ; Pointer to the definitions
```

```
                move.w   BP_INIT,a2        ; The vector we need to use
                jsr      (a2)              ; Call the vectored routine
                rts                        ; Return to SuperBasic

*--------------------------------------------------------------------
* Definition table for one new procedure
*--------------------------------------------------------------------
define          dc.w     1                 ; 1 new procedure
                dc.w     nlist-*           ; Offset to procedure
                dc.b     5,'NLIST'         ; Size and name
                dc.w     0                 ; End of procedures

                dc.w     0                 ; Number of functions
                dc.w     0                 ; End of functions

*--------------------------------------------------------------------
* Procedure NLIST starts here ...
*
* Check for one or zero parameters - if not then error exit
*--------------------------------------------------------------------
nlist           cmpa.l   a3,a5             ; No parameters?
                beq.s    nl_none           ; Yes, skip
                move.l   a5,d0             ; Last parameter pointer
                sub.l    a3,d0             ; minus first
                cmpi.w   #8,d0             ; One parameter?
                beq.s    got_one           ; Yes

bad_par         moveq    #-15,d0
error_exit      rts

*--------------------------------------------------------------------
* If one parameter, must have a hash else error exit
*--------------------------------------------------------------------
got_one         btst     #7,1(a6,a3.l)     ; check for a hash
                beq.s    bad_par           ; Not got one

*--------------------------------------------------------------------
* It has a hash - fetch the channel id. If this fails, error exit.
*--------------------------------------------------------------------
get_one         move.w   ca_gtint,a2       ; Vector for word integers
                jsr      (a2)              ; Fetch!
                tst.l    d0                ; Ok?
                bne.s    error_exit        ; No, bale out
                cmpi.w   #1,d3             ; One only?
                bne.s    error_exit        ; No, bale out
                move.w   0(a6,a1.l),d0     ; Fetch channel number
                addq.l   #2,a1             ; Tidy stack
                tst.w    d0                ; Set flags
                blt.s    bad_par           ; Negative is a bad id
                bra.s    chan_ok           ; skip default handling

*--------------------------------------------------------------------
* No parameters supplied - default channel number to #1
*--------------------------------------------------------------------
nl_none         moveq    #1,d0             ; Default to channel #1
chan_ok         bsr.s    channel_id        ; convert to channel id in A0
```

```
                bne.s    error_exit         ; Oops!

*--------------------------------------------------------------------------
* Fetch the start of the name list from BV_NLBAS(A6). The result
* of this is an offset from A6 to where the namelist actually
* starts.
*--------------------------------------------------------------------------
                move.l   bv_nlbas(a6),a3    ; Start of name list
*                                           ; Relative to A6!

*--------------------------------------------------------------------------
* Our main loop starts here. We test to see if we are finished
* and if not copy the (next) name to the buffer formatting it as
* a QDOS string.
* D3 is preserved inside the loop, so set it once just before the
* loop starts.
*--------------------------------------------------------------------------
                moveq    #-1,d3             ; Timeout for the channel

nl_loop         cmpa.l   bv_nlp(a6),a3      ; Compare offsets - done?
                bge.s    nl_done            ; Yes
                moveq    #io_sstrg,d0       ; Print some bytes please
                move.b   0(a6,a3.l),d2      ; Counter byte from name list
                ext.w    d2                 ; Needs to be word sized
                lea      1(a6,a3.l),a1      ; Start of bytes to print
                adda.w   d2,a3              ; Adjust to end of bytes
                addq.l   #1,a3              ; A3 = next entry, size byte
                trap     #3                 ; Print the name
*                                           ; Preserves A0, A3 and D3
                tst.l    d0                 ; Ok?
                bne.s    error_exit         ; Oops - failed

nl_nl           moveq    #io_sbyte,d0       ; Code for 'send one byte'
                moveq    #10,d1             ; Newline character
                trap     #3                 ; Print newline
*                                           ; Preserves A0, A3 and D3
                tst.l    d0                 ; Ok?
                bne.s    error_exit         ; Oops - failed

                bra.s    nl_loop            ; Lets go round again!

*--------------------------------------------------------------------------
* If there is no more to do, return to SuperBasic.
*--------------------------------------------------------------------------
nl_done         moveq    #0,d0              ; No errors
                rts                         ; Exit to SuperBasic

*--------------------------------------------------------------------------
* Copy the above code for the CHANNEL_ID subroutine to here as it
* is required.
*--------------------------------------------------------------------------
channel_id ...
```

Listing 7.16: Procedure to Print the Entire Name List

Save the file, assemble, fix typing errors and test - super stuff this eh?

When this procedure runs, you can see all the internal names like PRINT, CLOSE etc and also all your own stuff like NLIST, GREEN, RED etc and also any filenames that you have used without quotes around them. These are names like anything else.

if you try the following:

```
open_new #3,ram1_test
nlist #3
close #3
```

then load ram1_test into your editor (or copy to scr_), the last entry in the name list will be ram1_test - because you didn't use quotes. If you now try:

```
open_new #3,"ram1_test_again"
nlist #3
close #3
```

This time, ram1_test_again will NOT be in the list because it is not a name, simply a string. This routine can be used to get a list of all procedures, functions, names etc that are loaded into your QL.

7.8 The Maths Stack

The maths stack is where all internal mathematical calculations of floating point variables are done. It is also used to allow parameters passed to machine code procedures and functions to be 'collected' from the user and passed to the registers etc for use by the procedure or function.

The maths stack is simply an area of memory which can be used for all these fancy calculations, parameter handling etc. There is nothing (much) special about it and it is *always* addressed internally using register A1 (relative to A6 - but you knew that didn't you?)

One of the first things I learned when writing extensions to SuperBasic was that on entry to a function or procedure, the A1 register is set to a value corresponding to the top of the maths stack. This is a *myth* and is not correct.

The value in register A1 can be anything on entry to a machine code function or procedure. I have done a lot of investigating (thanks to QMON2) and come up with the following rule:

If you want a suitable value in A1 for the top of the maths stack, then either fetch some parameters, or, load it from BV_RIP.

This means that if a function wants to return a value - which functions usually do - and the function has no parameters then you must load A1 from BV_RIP(A6) before calling the `BV_CHRIX` vector to reserve space. As I found out to my cost, not setting A1 is a good way to trash the system!

If your function does have parameters, then AFTER they have been fetched, A1 is set ok, up until that time, it is not and has the following possible values:

7.8.1 A1 Is Negative

If A1 is a negative number, then your function has been called as part of an expression such as:

```
PRINT 10 * MY_FUNCTION(p1, p2, p3 ....)
```

The number in A1.L is the number of bytes that have already been used on the maths stack for the '10' in this case. This will be -6 as the 10 will be stored as a floating point number.

7.8.2 A1 Is Zero

If the number in A1 is zero, then your function has been called thus:

```
PRINT MY_FUNCTION(p1, p2, p3 ....)
```

or

```
PRINT MY_FUNCTION(p1, p2, p3 ....) + 10
```

and no bytes have been used on the maths stack yet.

7.8.3 A1 Is Positive

If A1.L is greater than zero then this implies that there are A1.L many bytes available on the maths stack and calling `BV_CHRIX` to allocate stack space will not move the maths stack around in memory.

Warn

I have *never* seen this documented and it has been discovered by me during long debugging sessions. Now that SMSQ is here, the above information may no longer be valid. The *only* thing to remember is that on entry to a procedure or function, A1 *does not* hold a suitable value for the top of the maths stack as stated in various documents.

So that is the real situation and not as specified in the documentation. I took ages to debug one simple function I wrote, which had no parameters and required some space on the maths stack for its result. Take a look at the code in the colour functions (green, red etc) we wrote back at the start of this article and you will see the following code:

```
return_d7   move.l   bv_rip(a6),a1    ; Because we had no params
            moveq    #2,d1            ; Size of stack space needed
            move.w   bv_chrix,a2      ; Allocate maths stack space
            jsr      (a2)             ; Get some space
```

As you can now see, we load A1 from BV_RIP because none of the functions had any parameters passed. Had that one line of code been missed out, your QL would have crashed. Try it if you like!

Values on the maths stack must be stored at even addresses. For integers, long integers and floating point values, this is not a problem. Strings, on the other hand, must be set up correctly with the word defining the size n an even address and the bytes of the string following. Odd length strings should have an extra padding byte to keep the A1 maths stack pointer even.

If you read back to section 7.5.1 'Keeping things even' then you will see how to do this. If you are returning a string from a function, you will need to reserve space for the string, its word count and a possible spare byte for padding. Refer to the explanation above and you will see why the following code 'just works' :

```
ret_string  move.w  (a0),d1       ; Assume string is at (A0)
            addq.w  #3,d1         ; Add size word + padding
            bclr    #0,d1         ; Force even size
            move.w  bv_chrix,a2   ; Allocate maths stack space
            jsr     (a2)
```

Of course, I am assuming that A1 holds a suitable value. The code above will request an even amount of space for a string result. First we fetch the length into D1 - this is the number of characters in the string only.

We then add 3 to D1. This is 2 for the word count and one for a possible padding byte. By clearing bit zero of D1 we force the number to be even and can then carry on with the request for space etc. Easy stuff this!

7.9 Returning Values From Functions

When returning values on the maths stack you must be very careful. When a function exits there must be a value on the top of the maths stack the pointer to this value needs to be stored in BV_RIP(A6) and D4 has to have a values in it which defines the returned parameter type. See Table 7.11.

D4	Return Parameter Type
1	String
2	Floating point
3	Word integer

Table 7.11: Function Return Data Types

Notice anything missing? Although we are allowed to fetch long integers as parameters, we are not allowed to return them. This is a problem and the usual fix is to convert a long integer to a floating point and return that instead. This will be covered in another thrilling episode !

7.10 Channel Tables

In our procedure PSI_CLS, we use a channel number in SuperBasic. In assembler, this is no use to us as all internal operations that require a channel (CLS, PAPER etc) require a channel id which is a 32 bit long number which bears no resemblance (or only coincidentally) to a SuperBasic channel number.

In QDOS there is a channel table - for the entire system, and there is the SuperBasic channel table which is used to convert channel numbers into channel ids which is what we require. SuperBasic keeps us away from nasty things like internal representations - assembler does not.

The routine we used above, channel_id, is all that is required to convert a channel number to a channel id. It looks at the SuperBasic channel table and for each channel that has been opened (even if it is now closed) there will be an entry in the channel table. Each entry is $28 bytes long (40 bytes) and has the structure shown in Table 7.12.

When a channel is opened in SuperBasic, an entry is created (or reused) in this table. At startup channels #0, #1, and #2 are pre-created and that is all. If you now open #4, a new entry will be

Offset	Size	Purpose
$00	Long	QDOS internal channel id
$04	6 bytes	Graphics cursor X position (Floating Point format)
$0A	6 bytes	Graphics cursor Y position (Floating Point format)
$10	6 bytes	Turtle angle (Floating Point format)
$16	Byte	Pen status (0 = up or 1 = down)
$20	Word	Character position on line for PRINT and INPUT etc
$22	Word	Width of the channel. Set by WIDTH command in SuperBasic but defaults to 80 when OPEN is called.
$24	Long	Spare - currently unused

Table 7.12: SuperBasic Channel Table Definition

created for it. If you open channel #10, then blank entries are created for all the 'in-between' channels (5 to 9) and entry 10 is then created and initialised on top.

A channel that has never been opened can therefore still have an entry in this table - channels 5 to 9 in the above example. All of these use memory so it is advisable to start with 3 and work upwards opening channels as you go, rather than opening #100 or something similar which needlessly wastes 40 bytes of memory for each unused channel.

A channel that is closed, or has never been opened, has a QDOS channel id which is negative.

In the Basic variables area in QDOS (to be covered in a later issue - and by the way, I refer to the variables that hold information about SuperBasic, and not variables you create in SuperBasic!) BV_CHBAS holds the offset from A6 to the first entry in the table (ie channel #0) and BV_CHP holds the offset from A6 to the first byte after the last entry in the channel table. Don't forget that these are offsets and that everything in SuperBasic is relative to A6 - simply because by doing this the base address for the job (SuperBasic is just another job in the machine) is held in A6. If everything else is stored as an offset then moving the job in memory is simple as only the A6 register has to be updated.

Take a look at the code for channel_id again and note how we are using addresses that are relative to A6. Make sure that you understand because all fiddling in the bowels of SuperBasic requires that you understand relative addressing.

Most of the time you will only be interested in the conversion from SuperBasic channel number to QDOS channel id.

7.11 Exercise

As an exercise, why not add a new procedure called PSI to the code for PSI_CLS. This new procedure will carry out all the same work as PSI_CLS but it will not do the CLS part of it. This will be useful when you want to set the colours for a window but not clear it. I will NOT be giving the answers out next time, but here are a few hints:

- update the definition table with details of the new procedure.
- in the proc's code, set D6.B to zero for PSI and 1 for PSI_CLS. Do this as the first instruction in both procedures.
- In the PSI procedure, simply set D6 and jump to the code in PSI_CLS.
- Just before doing the actual CLS part of PSI_CLS, check the value in D6.B and if zero, don't

do the CLS simply BRA.S to error_exit instead.

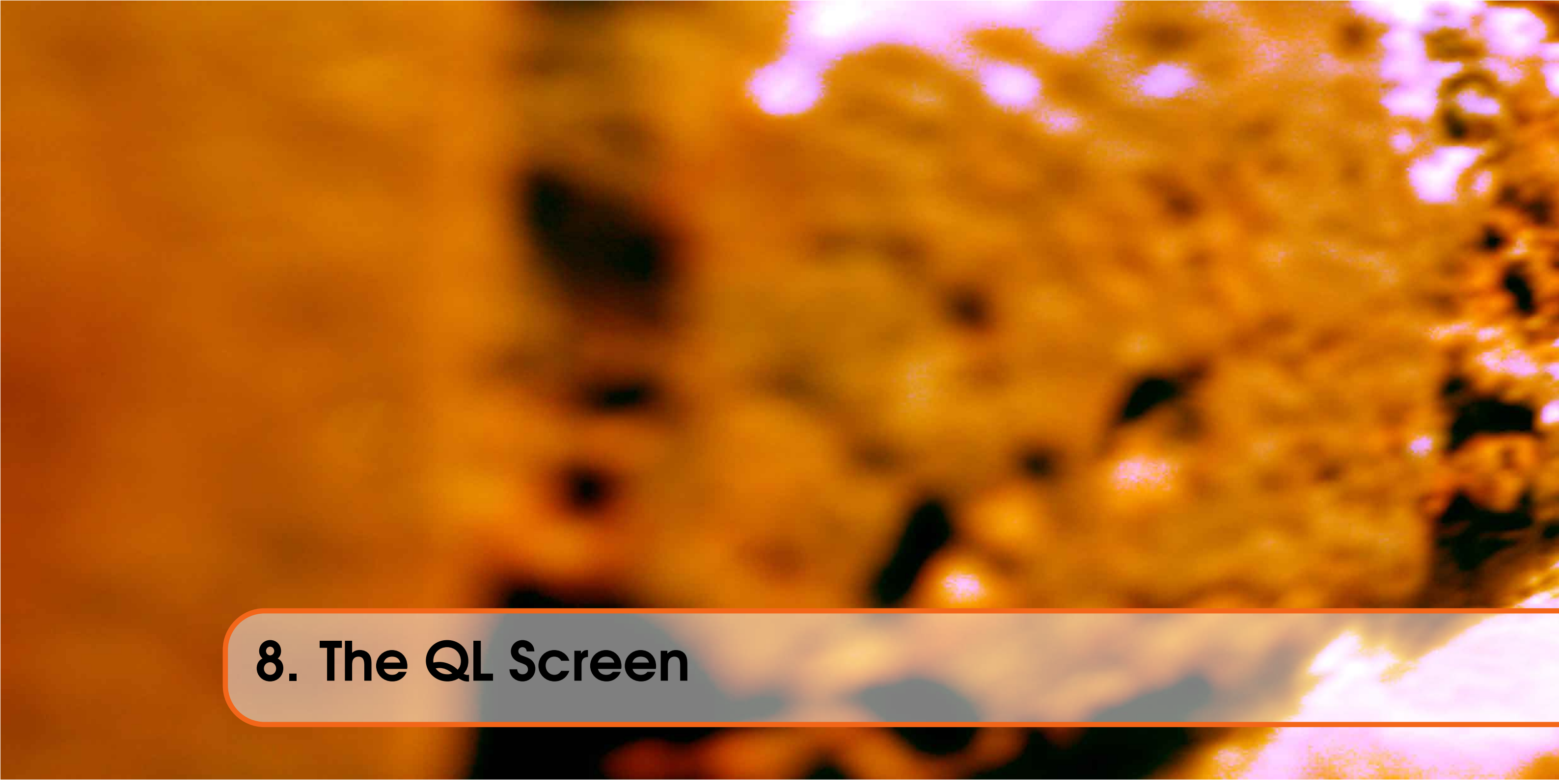
All in all, I think this can be done in about 10 extra lines of code, maybe less, not counting the extra lines in the definition block.

Warn

Adding even a few lines of code can sometimes cause any ‘short’ branches to go out of range and this will cause errors in the assembly. If this happens, simply find the ones in error and remove the ‘.s’ from the ‘bsr’ or ‘bra’ instructions.

7.12 Coming Up...

In the next chapter we delve into the QL’s screen layout and using our new found knowlege of assembly language programming, we will develop a mode 4 ‘plot’ routine in assembler. If you find this easy, there is an exercise for the reader - to develop the corresponding mode 8 ‘plot’ code !



8. The QL Screen

8.1 Introduction

In the last chapter, we looked at how easy it was to extend SuperBasic with new procedures and functions. Hopefully you all tried out the exercise I left for you to do, if not, there will be points deducted from your final score at the end of the course!

In this chapter, we shall take a look at the QL's screen memory and how to play around with it. I won't be writing any extensions to SuperBasic this time, but you could extend some of the routines to do so yourselves, and extend SuperBasic to your heart's content.

8.2 The Screen

Inside the original QL, there were supposed to be two screen areas. As it turned out, the final product only had one, but some memory was still left around for the second. Unfortunately, the second screen's memory has been partially overwritten by the system variables and so cannot be safely used. To all intents and purposes, we can ignore that second screen and concentrate on the primary screen itself. This is the one we can all use.

Nowadays, we have all sorts of screen modes and resolutions and with the coming of the Q40 & Q60, we have numerous colours as well. As an old lag, I deal in mode 4 and mode 8 only but as I use a QXL mostly (I am awaiting delivery of QPC 2 even as I type, and hopefully it will have arrived by the time you read this!) I also have more resolution that the old 512 by 256 that the original QL was limited to.

I also have no documentation regarding the resolutions available on other emulators, cards etc so I cannot deal with those here - perhaps someone with more details/knowledge could write a follow up article for an Aurora, Super Gold Card, Q40 etc. (Please!)

In the old days, 512 by 256 was the best you could expect - and only on 4 colours - red, black, green and white. If you wanted more colours, you only had 256 by 256 to play with, however you did get to use blue, yellow, magenta and cyan as well - it was a trade off, as with most things computer related.

OK, here is how it was in the old days .... the screen starts at address $20000 or 131,072 in the QL's memory. Each line on the screen, all 256 of them, use 128 bytes to hold the colour information for the pixels in the line. This implies that a QL screen takes up 32K of memory, and indeed this is the case. To get the screen memory address of pixel x,y (x = dots across and y = dots down) a calculation similar to the following was used:

$address = 131072 + (y * 128) + INT(x/4)$

This is because each scan line (or row down the screen) starts 128 bytes on from the previous line hence $(y * 128)$. Each row has 512 pixels in it (even in mode 8!) so the dots across are $512/128 = 4$. This is why the dots across (or x) must be divided by 4.

Warn

Don't ever assume that the two paragraphs above are true. The various new cards and graphics modes have changed all of the above. On my QXL, I can see the screen at the above address only when I run it in QL 512 by 256 mode. The other modes use more memory and in different places, so any program that writes to the screen at the original addresses will probably cause carnage within the QXL and lead to unexplained crashes later on - if not straight away. It must always be assumed the the old ways have gone forever and we must always calculate the screen start address and how long a scan line is before trying to access the memory.

For those of you who care about these things, the base of the screen address is at offset $32 in the channel definition block, while the size, in bytes, of a scan line is at offset $64. (Except if the QDOS version is less than 1.03, in which case, the scan line size is always 128 bytes.)

How to get this information? Easy, given the following code which assumes that A0.L holds a channel id for a scr_ or con_ channel:

```
scr_stuff    moveq   #sd_extop,d0       ; Trap code
             moveq   #-1,d3             ; Timout
             lea     extop,a2           ; Routine to call via sd_extop
             trap    #3                 ; Do it
             tst.l   d0                 ; OK?
             bne.s   done               ; No, bale out D1 = A1 = garbage

got_them     move.w  d1,-(a7)           ; Need to check qdos, save scan_line
             moveq   #mt_inf,d0         ; Trap to get qdos version
             trap    #1                 ; Get it (no errors)
             move.w  (a7)+,d1           ; Retrieve scan_line value
             andi.l  #$ff00ffff,d2      ; Mask the dot in QDOS "1.03" etc
             cmpi.l  #$31003034,d2      ; Test "1x03" where x = don't care
             bcs.s   too_old            ; Less than 1.03 is too old
done         rts                        ; Finished

too_old      move.w  #128,d1            ; Must be 128 bytes
             rts                        ; All done

extop        move.w  $64(a0),d1         ; Fetch the scan_line length
             move.l  $32(a0),a1         ; Fetch the screen base
             moveq   #0,d0              ; No errors
             rts                        ; done
```

Listing 8.1: Obtaining the Screen Address with SD_EXTOP

So given that we have a channel id in A0 we can extract the required information from the channel definition block by using the `SD_EXTOP` trap. This trap takes the address of a routine to call in A2, parameters for the routine in D1, D2 and A1, a channel id in A0 and returns with D1 and A1 holding values returned from the routine called and an error code in D0.

The way we are using it here we don't need any parameters on the way in, but coming out, D1.W holds the scan_line size and A2.L holds the address for the start of the screen memory.

The actual routine itself get presented with the channel definition block's address in A0, not the channel id. Within the routine we copy the screen base address into A1 and the scan_line size into D1.W and return.

On exit, we need to know if the scan_line size is correct so we call QDOS again to get the version of QDOS in D2. As this corrupts D1 we first save it on the stack. After the trap, D2 holds the ASCII representation of the QDOS version, for example '1.02' or '2.10' or possibly '1m03' for some foreign ROMS. (Foreign as in not UK!)

To test for the version we simply mask out the dot or the 'm' or whatever from D2 and if the version is less than 1x03, we simply set D1.W to 128 as this is the only value allowed. All other QDOS versions from 1x03 onwards have the correct scan_line size in D1.W.

So, on exit, A1.L holds the screen address and D1.W holds the scan_line size in bytes. This scan width is useful because we can use it to discover the maximum width of the screen in pixels, provided we know the mode - and I am talking about mode 4 and 8 only here because that is all I know about!

If we have, as I have on my QXL, a scan_line of 160 bytes, what is this telling me? It says that the number of pixels across the screen will fit into one scan_line of 160 bytes. In mode 4 I know that one word of memory holds the data for 8 individual pixels. In mode 8, I know that one word in memory holds the data for 4 pixels. (Or, as My wife Alison refers to them, 'pixies'.)

As there are 16 bits in a word we can assume correctly that two bits hold the data for mode 4 pixels and 4 bits hold the data for mode 8 pixels. Thus we have 160 bytes times 8 bits and divided by 4 to give 640 pixels across in mode 4. In mode 8 the answer will be 320 BUT the screen width is always the mode 4 width. Only the pixels double up in mode 8, so plotting point 639,0 in mode 8 still works! (or is it 0,639 - I can never remember!)

Our calculation above still works because the memory address of a pixel is now:

$$screen\_base + (y * screen\_width) + INT(x/4)$$

and this works even on a QXL. We come back to this later.

8.3 Mode 4 - screen memory usage

So, as I said above, we have two bits per pixel (or 8 pixels per memory word) in mode 4. How does this work? Mode 4 allows 4 colours, in binary the numbers from 0 to 3 can be represented by two bits. Colours are also represented by 'digits' in that if you add two colours together you get a different colour

The word in memory looks like Table 8.1.

In the above table, G7 refers to bit 7 of the green byte. The green byte is always even and lower in memory than the red byte which is always odd.

The colour codes for the allowed mode 4 colours are as per Table 8.2.

Green byte bits (even address)	Red byte bits (odd address = green address + 1)
G7 G6 G5 G4 G3 G2 G1 G0	R7 R6 R5 R4 R3 R2 R1 R0

Table 8.1: Mode 4 Screen Memory Word Format

Colour	GR (Binary)	Value (Decimal)
Black	00	0
Red	01	1
Green	10	2
White	11	3

Table 8.2: Mode 4 Colour Codes.

So white is represented by both colours mixed together, black by the lack of both colours and red and green by themselves.

If in memory we have the green byte and the red byte in each word set up as follows, we can add the corresponding bit in each byte to represent the colour for a single pixel as follows:

```
Green byte = 0000 1111
Red byte   = 0101 0101
```

Which gives us the following:

Bit	GR (Binary)	Colour
7	00	Black
6	01	Red
5	00	Black
4	01	Red
3	10	Green
2	11	White
1	10	Red
0	11	White

Table 8.3: Mode 4 Example Bits

And that is how it works in mode 4. So we know the screen address (or do we? Think about it) and we know how to poke values into the correct location so we can now write directly to the screen can't we? More later, keep those brain cells ticking over for now. There is something I have not yet mentioned.

8.4 Mode 8 - screen memory usage

In mode 8 we have 8 different colours. To represent the values 0 to 7 we need at least 3 bits. As there is flashing allowed in mode 8, we need a bit for flash on or flash off as well. 4 bits per pixel is what we need and that is what we use.

In this mode, the green byte and the red byte are at the same addresses as in mode 4 with the green being even and the red being odd, but the layout is different. The green byte shares with the flash bit where the green bit is the odd numbered bit (7, 5, 3, 1) and the flash bits are in the even bits (6, 4, 2, 0). A similar arrangement goes on in the red byte with the red bits being even and the blue being odd. So the layout looks like Table 8.4.

Green byte bits (even address)	**Red byte bits (odd address = green address + 1)**
G3 F3 G2 F2 G1 F1 G0 F0	R3 B3 R2 B2 R1 B1 R0 B0

Table 8.4: Mode 8 Screen Memory Word Format.

Again the values for the colours represent the mixing of the reds, greens and blues - much like colours in nature are just mixes of red, blue and yellow. (Light and inks mix differently and so have different primary colours. In photography, we use yellow, cyan and magenta!)

The colours are as per Table 8.5:

Colour	**GRB (Binary)**	**Value (Decimal)**
Black	000	0
Blue	001	1
Red	010	2
Magenta	011	3
Green	100	4
Cyan	101	5
Yellow	110	6
White	111	7

Table 8.5: Mode 8 Colour Codes

So given the following bit pattern in mode 8:

```
Green byte = 0x0x 1x1x
Red byte   = 1001 1110
```

and ignoring the flash bits (shown as ‘x’ above)and combining the appropriate GRB bits from each byte we get the results shown in Table 8.6:

Bits (in each byte)	**GRB (Binary)**	**Colour**
76	010	Red
54	001	Blue
32	111	White
10	110	Yellow

Table 8.6: Mode 8 Colour Bits.

The flash bits are strange. At the beginning of each scan line, the flashing is turned off until such time as a flash bit is set - this turns flashing on until the next flash bit which is set is found. This turns flash off again - so the flash bits act like a toggle turning flash on and off each time a set bit is found.

Note

Most books I have read on the subject totally ignore the flash bits after this discussion - I am going to go into it in much more depth. Well that was a lie, I’m not!

8.5 That calculation again!

Have you had a good think about calculating screen addresses for pixels then? Better still, have you thought about the problem I hinted at above? What is the problem then?

If each word of the screen memory holds data for either 8 or 4 pixels, then how can we calculate the correct address for each pixel, because it is (now) obvious that the address for the first 8 pixels in each row will be the same in mode 4 (or 4 pixels in mode 8) so our wonderful calculation above needs a bit of tweaking to make it work correctly.

In mode 4, the screen address changes every 8 pixels across. So where x is 0 to 7, the screen address is the same, for x = 8 to 15 it is the next word of memory and so on. The word that the x pixel lives in is found by the calculation, but the actual pixel within that group of 8 pixels is not found. Follow?

Assume row zero and pixel 2, this gives screen address =

$base\ address + (0 * scan\ width) + INT(2/4)$

or

$base\ address + 0 + 0$

or

$base\ address$

This is the same address for pixel 0 through pixel 7. For pixels 8 to 15 it will be as follows (using 8 in the calculation):

$base\ address + (0 * scan\ width) + INT(8/4)$

or

$base\ address + 2$

so we know the memory word, but not the actual bits within it. Remember bits 7 = pixel 0, bit 6 = pixel 1 and so on down (up?) to bit 0 for pixel 7. How do we get to a value between 0 and 7 from any x value? If we AND the x value with 7 that will give us a value between 0 and 7 won't it - lets see:

X	X AND 7
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	0
9	1
10	2

Table 8.7: Truth Table for X AND 7

And so on. Are these the correct values for the bits in the word that we want? Try this and see if we get the results shown in Table 8.8:

X AND 7	Correct Bit
0	7
1	6
2	5
3	4
4	3
5	2
6	1
7	0

Table 8.8: X AND 7 plus the Bits Required

Not quite it would appear, but we could always subtract $(x\,AND\,7)$ from 7 couldn't we? That would give the correct answer. So a solution is at hand. If we subtract the result of $(x\,AND\,7)$ from 7, we get the correct bit number in each byte of the calculated memory word. Yippee (or is it - read on.)

Not quite, I'm afraid. If we have the memory address, we can extract the current contents - we must preserve the other 7 pixels when we plot this one remember - so we need to mask out the same bit in each byte of the screen word. If we used the subtraction method identified above, we would needs bucket loads of testing and masking to figure out which bit is required. We need another method. Before we get to that, how exactly shall we preserve the current pixels?

Remember that a pixel is defined by a single bit in the green byte and the corresponding bit in the red byte of the screen word. To set a pixel we must first set its two bits to zero (or black) and then set the two bits according to the requested colour. This turns out to be quite simple.

First create a mask where the bit to be changed in the red and green bytes are set to zero and every other bit is set to 1. If we AND this mask word with the screen word we effectively set that one pixel to black. So far so good. Next set a new mask where the single bit in each byte is the requested green or red bit and all the rest are zero. If we now OR this word with the screen word we have set the pixel to our requested colour. Too many words, lets have an example.

Our screen shows the following colours in the first 8 pixels:

```
red  green  green  black  black  white  red  white
```

This means that we have the following two bit values for each pixel:

```
01  10  10  00  00  11  01  11
```

Which means that we have the following word in memory:

```
01100101  10000111  =  $6587
```

Now let us assume that we want to colour the first pixel (currently red) to white. So our mask to clear that bit (bit 7 in each byte) needs to be set to

```
01111111  01111111  =  $7f7f
```

Now we AND this word with the screen word to get the following :

```
01100101 00000111 = $6507
```

Note now that the first pixel has been set to 00 (bit 7 from both bytes) so it has effectively been set to black.

Next we need a white pixel so the colour mask for white must have a 1 in bit 7 of each byte. The rest must be zero to preserve the current colours of all the other pixels. Our mask must be:

```
10000000 10000000 = $8080
```

So if we now OR this into the (new) screen word - currently $6507 - we get the following:

```
11100101 10000111 = $E587
```

Taking all the bits into colour values we get this:

```
11 10 10 00 00 11 01 11
```

which translates back to the following colours:

white green green black black white red white

Success, we have preserved all other pixels and set the first one to white. Now we know how to do it to one pixel, it is the same for all the other 7, but the masks need to be changed for each pixel. How?

If we decide to change pixel 0 (as above) the masks are $7f7f and $8080. This is easy. If we want pixel 1 to be changed the masks are rotated one bit to the right becoming $bfbf and 4040 and so on. Look again at our table above where we show the result of $(x\ AND\ 7)$ and the correct bit in the screen word - notice that if we assume that pixel 0 is being changed we can rotate the masks by $(x\ AND\ 7)$ bits to get the correct masks for whichever pixel we try to set, as Table 8.9 shows:

Pixel	X AND 7	AND Mask	OR Mask
0	0	01111111	G0000000 R0000000
1	1	10111111	0G000000 0R000000
2	2	11011111	00G00000 00R00000
3	3	11101111	000G0000 000R0000
4	4	11110111	0000G000 0000R000
5	5	11111011	00000G00 00000R00
6	6	11111101	000000G0 000000R0
7	7	11111110	0000000G 0000000R

Table 8.9: Bitmaps for Mode 4 pixel masking.

Note

I have only shown one byte of the AND mask, the other byte is identical as we are masking out the same bit in each byte.

Looking at the table, we see that the result of $(X\ AND\ 7)$ is the pixel we need to set in the screen. If we start with a mask suitable for pixel 0 and ROTATE it to the right by $(X\ AND\ 7)$ bits, we get the correct mask for that pixel. This also works for our colour mask as well. Things sometimes become clear when you switch to binary, especially in graphics situations!

We now have the basics for a mode 4 'pixel setting' routine. Lets try it out.

Assume that we want to set the colour of any pixel on the screen to any of the 4 colours we want in mode 4. We can actually use any of the mode 8 colours because only bits 2 and 1 will be used. This means that a mode 8 colour of blue (value 001) will result in a mode 4 value black (value 00) being set for the appropriate pixel. This is exactly how SuperBasic would handle it.

We will use the registers as follows:

```
D1.W = x (across)
D2.W = y (down)
D3.W = colour (0 to 7)
```

Here's the code in all its glory:

```
*======================================================================
* In D3 bit 2 is green and bit 1 is red, we don't need any other bits,
* so get rid of them now. Then shift the Green bit into bit 15 of D4
* and the red into bit 7 of D3 ...
*======================================================================
start       bra      plot_init        ; Call start+4 to initialise things

plot_4      bsr.s    calc             ; Get A1 = screen address
            andi.w   #6,d3            ; D3 = 00000000 00000GR0
            lsl.w    #6,d3            ; D3 = 0000000G R0000000
            move.w   d3,d4            ; D4 = 0000000G R0000000
            lsl.w    #7,d4            ; D4 = GR000000 00000000
            or.w     d4,d3            ; D3 = GR00000G R0000000
            andi.w   #$8080,d3        ; D3 = G0000000 R0000000

*======================================================================
* D3.W is now set to a colour mask for pixel 0. This is where we want
* to start. Now we need to build a mask to clear out pixel 0 as well.
* This is easy - use the value from the table above. Then we can start
* rotating them into the correct position as detailed above.
*======================================================================
            move.w   #$7f7f,d2        ; AND mask = 10000000 10000000
            andi.w   #7,d1            ; (x AND 7) in d1
            ror.w    d1,d2            ; Build correct AND mask
            ror.w    d1,d3            ; Build correct OR mask (colour)
            and.w    d2,(a1)          ; AND out the changing pixel
            or.w     d3,(a1)          ; OR in the (new) colour
            moveq    #0,d0            ; No errors
            rts                       ; All done

*======================================================================
* Calculate the screen address for the x and y values passed in D1 and
* D2. Trashes A1, D4 and D5.
* The routine plot_init must have been called to initialise the screen
* addresses and scan line widths BEFORE calling this routine.
*======================================================================
```

```
calc        lea      scr_base,a1         ; Storage for screen base address
            move.l   (a1)+,d0            ; Fetch the screen base address
            move.w   (a1),d6             ; And the scan line size
            movea.l  d0,a1               ; Save it

*=====================================================================
* D1.W = x across value
* D2.W = y down value
* D3.W = ink colour required
* D6.W = scan line size
* A1.L = screen base address
*=====================================================================
            move.w   d2,d5               ; Copy y value (down)
            ext.l    d5                  ; We get a long result next ...
            mulu     d6,d5               ; Multiply by scan_line size
            adda.l   d5,a1               ; A1 = correct scan line address

            move.w   d1,d4               ; Copy x value
            lsr.w    #2,d4               ; D4 = INT(x / 4)
            bclr     #0,d4               ; Even address = green byte
            adda.w   d4,a1               ; A1 = correct screen word address
            rts                          ; Done

*=====================================================================
* This routine must be called once before using the plot routines. It
* initialises the screen base address and scan line width from the
* channel definition block for SuperBasic channel #0.
*=====================================================================
plot_init   suba.l   a0,a0               ; Channel id for #0 is always 0
            lea      scr_base,a1         ; Parameter passed to extop routine
            lea      extop,a2            ; Actual routine to call
            moveq    #sd_extop,d0        ; Trap code
            moveq    #-1,d3              ; Timout
            trap     #3                  ; Do it
            tst.l    d0                  ; OK?
            bne.s    done                ; No, bale out D1 = A1 = garbage

got_them    move.w   d1,-(a7)            ; Need to check qdos, save scan_line
            moveq    #mt_inf,d0          ; Trap to get qdos version
            trap     #1                  ; Get it (no errors)
            move.w   (a7)+,d1            ; Retrieve scan_line value
            andi.l   #$ff00ffff,d2       ; Mask the dot in QDOS "1.03" etc
            cmpi.l   #$31003034,d2       ; Test "1?03" where? = don't care
            bcs.s    too_old             ; Less than 1.03 is too old

save        move.w   d1,(a1)             ; Store the scan_line size

done        rts                          ; Finished

too_old     move.w   #128,d1             ; Must be 128 bytes
            bra.s    save                ; Save D1 and exit

extop       move.l   $32(a0),(a1)+       ; Scan_line length - stored
            move.w   $64(a0),d1          ; Screen base - not stored
            moveq    #0,d0               ; No errors
            rts                          ; done
```

```
*==========================================================================
* Set aside some storage space to hold the screen base and scan_line
* width. This saves having to calculate it every time we plot a pixel.
*==========================================================================
scr_base    ds.l    1
scan_line   ds.w    1
```

Listing 8.2: Mode 4 Screen Plotting

And that is the end of the code. To use the above in your assembly language programs simply call plot_init once to set up the screen base and scan line widths, then call plot_4 as often as you like. Easy stuff.

To test this code out from SuperBasic, ALCHP (or RESPR) some heap and LBYTES the code file to that address and CALL it. This initialises the system by calling plot_init. Now, simply CALL address, x, y, colour and the points will be plotted. Make sure you are in mode 4 or the results may be a bit crazy! An example program follows:

```
1000  PLOT_INIT = RESPR(256): REMark Enough space for plot_8 as well!
1005  PLOT_4 = PLOT_INIT + 4
1010  LBYTES flp1_plot_bin , PLOT_INIT
1015  CALL PLOT_INIT
1020  FOR across = 0 to 100
1025    FOR down = 0 to 100
1030      CALL PLOT_4, across , down, RND(0 to 7)
1035    END FOR down
1040  END FOR across
```

8.6 Problems

Ok, so what, if anything is wrong with the plot_4 routine? The answer is that there is no checking to see if the x and y values are out of range. If you try to plot say pixel 2000,494 the chances are that it would corrupt something in memory (probably a system variable) with either immediate or later results.

It is probably easy to check the x value (or across) because there are 8 pixels per word in mode 4 so multiplying the scan line width (in bytes) by 4 should give the maximum resolution across. Indeed, on my QXL, this works out. My scan line is 160 bytes and the maximum resolution is 640 across by 480 down. 160 times 4 is indeed 640. Unfortunately, I cannot think or find a method of calculating the maximum display resolution in the ‘downward’ direction.

It may be true that all current display resolutions that are 640 across must be 480 down, but is this true or not? It appears not. A quick check with the demo version of QPC 2 (an old demo version at that) shows that It can have the resolutions listed in Table 8.10 (across by down):

So we can already see that detecting a 640 pixels across resolution leads to a decision about the downward resolution, is it 400 or 480?

I feel the need to be told if there is a way, simple and effective and which works on all machines, whether they are black box QLs or Q40s or emulators, to tell the maximum screen resolution. Anyone got any ideas? If so, Dilwyn will be glad to print the article you are about to write!!

X (Across)	Y (Down)
512	256
640	400
640	480
800	600
1024	768
1152	864
1280	1024
1600	1200

Table 8.10: QPC Screen Dimensions

8.7 Exercise

For this exercise, I want you to write a mode 8 plot routine in a manner similar to the plot_4 routine shown above. Here are some hints :

- Avoid the flash bits like the plague. Simply mask them out and set them to zero.
- The calc routine works for mode 8 as well. No need to change it.
- The mask for pixel 0's colour needs to be GF000000 RB000000.
- The mask to clear pixel 0 needs to be 01111111 00111111 ($7f3f).

The algorithm is as follows:

- Calculate the screen address by calling calc. Sets A1 = screen address.
- Mask out all but bits 0, 1 and 2 of D3.W This is the pixel colour. D3 = GRB.
- Shift D3.W LEFT by 6 bits.
- Copy D3.W to D4.W
- Shift D4 left by 7 bits.
- `ANDI.W D4.W` with $8000 to preserve only bit 15 = G.
- `ANDI.W D3.W` with $C0 to zero the G bit currently in bit 8.
- `OR.W D4` into D3 to give the correct colour mask for pixel 0.
- `ANDI.W D1` with 6 to get the correct number of rotates (6 makes it even which it must be because we need to rotate two bits for each pixel.)
- Rotate right, the two mask words, the correct number of bits.
- `AND.W` the mask with the screen word.
- `OR.W` the colour mask with the screen word.
- Clear D0 and return.

The results of (x and 6) are as follows:

And so on. Because we are using two bits of the green and red bytes to represent our colour, we need to always rotate by an even number.

To test it all out, add the code to the end of the original file which has plot_4 in it and change the first two lines from this:

```
start       bra      plot_init
plot_4      bsr.s    calc
```

to the following:

```
start       bra      plot_init
```

X	X AND 6
0	0
1	0
2	2
3	3
4	4
5	4
6	6
7	6
8	0
9	0
10	2

Table 8.11: Truth Table for X AND 6

```
plot_4      bra      plot_4
plot_8      bra      plot_8
```

This means that plot_init is the start address, plot_4 is at address + 4 and plot_8 has been inserted at start address + 8, as follows:

```
1000  PLOT_INIT = RESPR(256): REMark Enough space for plot_8 as well!
1005  PLOT_4 = PLOT_INIT + 4
1010  PLOT_8 = PLOT_INIT + 8
1010  LBYTES flp1_plot_bin , PLOT_INIT
1015  CALL PLOT_INIT
```

Have fun.

8.8 Answer

```
plot_8    bsr.s     calc              ; Get A1 = screen address
          andi.w    #7,d3             ; D3 = 00000000 00000GRB
          lsl.w     #6,d3             ; D3 = 0000000G RB000000
          move.w    d3,d4             ; D4 = 0000000G RB000000
          lsl.w     #7,d4             ; D4 = GRB00000 00000000
          andi.w    #$8000,d4         ; D4 = G0000000 00000000
          andi.w    #$00C0,d3         ; D3 = 00000000 RB000000
          or.w      d4,d3             ; D3 = G000000G RB000000
          move.w    #$7f3f,d2         ; AND mask = 01111111 00111111
          andi.w    #6,d1             ; (x AND 6) in d1
          ror.w     d1,d2             ; Build correct AND mask
          ror.w     d1,d3             ; Build correct OR mask (colour)
          and.w     d2,(a1)           ; AND out the changing pixel
          or.w      d3,(a1)           ; OR in the (new) colour
          moveq     #0,d0             ; No errors
          rts                         ; All done
```

Listing 8.3: Mode 8 Screen Plotting

8.9 Coming Up...

That's all about the QL screen for the moment. Coming up in the next chapter , we will start to take a look at subroutines in Assembly Language and build a (hopefully) useful subroutine library which will allow us to include them in any new programs we write.

III

A Small Diversion into Subroutines.

9. Subroutines

9.1 Introduction

Here we are in part 9 of the series on assembly language for the QL and what we will look at today are subroutines.

9.2 Subroutines

A subroutine is simply a piece of code that you call lots of times within your program. Because it is called so many times, you extract the working code, move it somewhere safe and add an RTS at the end. This is your subroutine - in its draft form!

Where the code used to be in the main source, now simply has a BSR sub_routine in its place. The more times a routine is called, the bigger the saving in your typing and memory usage in the final program. Another major advantage of using subroutines is that you only need to change or correct them once - of course, if you make a mistake then every call to that subroutine is flawed as well!

For example, in a program you have written, you might find that you write the same piece of code numerous times to clear the screen, something like that shown in Listing 9.1:

```
start    blah  blah  blah
      :
      :
        move.l   channel_id,a0        ; First channel id
        moveq    #sd_clear,d0         ; CLS
        moveq    #infinite,d3         ; Infinite timeout
        trap     #3                   ; CLS title window
      :
      :
        move.l   other_channel_id,a0  ; Another channel id
        moveq    #sd_clear,d0         ; CLS
        moveq    #infinite,d3         ; Infinite timeout
```

```
        trap    #3                  ; CLS title window
        :
        :
        move.l  another_id,a0       ; And another channel id
        moveq   #sd_clear,d0        ; CLS
        moveq   #infinite,d3        ; Infinite timeout
        trap    #3                  ; CLS title window
        :
        :
        rts                         ; All done
```

Listing 9.1: Example of Repetitive Code

and so on. The above code looks duplicated and where you have duplication, you can usually - but not always - extract the duplicate code to a subroutine. We can now rewrite the code above as follows:

```
start   blah blah blah
        :
        :
        move.l  channel_id,a0       ; First channel id
        bsr     cls
        :
        :
        move.l  other_channel_id,a0 ; Another channel id
        bsr     cls
        :
        :
        move.l  another_id,a0       ; And another channel id
        bsr     cls
        :
        :
        rts                         ; All done

*------------------------------------------------------------------------
* Subroutine to clear the SCR or CON channel whose ID is held in A0.
*------------------------------------------------------------------------
cls     moveq   #sd_clear,d0        ; CLS
        moveq   #-1,d3              ; Infinite timeout
        trap    #3                  ; CLS title window
        rts
```

Listing 9.2: Example of Non-repetitive Code

The code that does the setting up of the various parameters for the system call to clear a channel has been extracted and placed at the end all by itself. An `RTS` instruction has been added to allow us to go back to where we came from. The second piece of code is easier (?) to read and will be smaller when finished.

So that is all there is to it. If you remember back to the boring part of this series (what do you mean ‘which boring part?’) where I discussed the inner workings of the `BSR` instruction, you will remember that `BSR` stacks the address of the instruction that will be executed next (after the `BSR`), jumps to the address given and continues executing from there until it finds an `RTS` instruction.

The `RTS` instruction stop the program in its tracks, sets the PC (2 points if you can remember what PC stands for ...) to the address that was stacked and proceeds to execute from there again. Those of you who are ahead of me at this point will realise that the `RTS` instruction takes the top 4 bytes off of the stack *regardless* of what they are. If they are a valid return address then fine, no problems. If, on the other hand, they are some data, then who knows what will happen when the RTS is executed.

For this reason, it is very important that your stack should be exactly the same on the way out of a subroutine as it was on the way in. Don't do something like that shown in Listing 9.3, for example:

```
start   blah blah blah
        :
        :
        move.l  channel_id,a0        ; First channel id
        bsr     cls
        :
        :
        rts

*------------------------------------------------------------
* BROKEN subroutine to clear the SCR or CON channel whose ID is in A0.
*------------------------------------------------------------
cls     move.l   d0,-(a7)             ; Preserve D0 until later
        moveq    #sd_clear,d0         ; CLS
        moveq    #-1,d3               ; Infinite timeout
        trap     #3                   ; CLS title window
        rts                           ; Program explodes here!
```

Listing 9.3: Example of a Messed up Stack!

In this example, the old value of D0.L is on the stack on top of the return address. When the RTS instruction is executed it doesn't know (or care) about what is on the stack, it just grabs the top 4 bytes and sets the PC to that 'address'. (You get 2 points if you remembered PC = Program Counter!)

9.3 Building A Library

As you progress with assembly language programming, you may find that you build up a lot of subroutines in your programs. What to do with them all?

Why not build yourself a library of routines that you can include in every program that needs them. This way, you have a full set of tried and tested bits of code - which you should document somewhere - that can be reused over and over again. The rest of this article will help you on your way by building a number of useful (well, I have found them to be useful over the years) subroutines that you can use.

9.4 Documentation

As with all good things, documentation is a must. If you have a large number of useful routines then they should be documented somewhere. This will allow you to look for a routine in your library and from that, find out its input & output parameters and which file it lives in.

A suitable template that you could use for each subroutine is as follows:

```
*------------------------------------------------------------
* NAME
* DEPENDENCY (1)
* DEPENDENCY (2)
```

```
* PURPOSE
* INPUTS
* OUTPUTS
*-----------------------------------------------------------------------------
```

The above looks very like comments in a source file - this implies that we could add the documentation to the source file and then run a utility program to extract the details and store them in a text file - which you can edit and/or print as desired.

Having a standard header above each subroutine also implies that you could write a utility program to scan the entire library and ask you which ones you want to include in your output file - which will be you source file for your next masterpiece - before extracting them all and writing them to this file.

As for the subroutines themselves, I mentioned above that they exist in a draft form when you simply extract the code from the 'wordy' source and add an `RTS` to the end. This is fine, but it could be that you need to preserve certain registers so that the code calling the subroutine doesn't need to keep saving and restoring them. The updates required are:

Check which registers will be used by the code explicitly - save them before and restore them after the main part of the subroutine code.

Check which system calls are made by the subroutine and look up the QDOSMSQ documentation to see which registers are trashed by the system call. Add these registers to the save and restore routines.

Save the registers as the first line of code in the subroutine and restore them as the line immediately before the `RTS` (or as near to the `RTS` as possible).

Always have the subroutine return an error code and/or the flags set to signal if an error occurred or not.

An actual example is shown in Listing 9.4:

```
*-----------------------------------------------------------------------------
* NAME           CLS
*-----------------------------------------------------------------------------
* DEPENDENCY     None
* PURPOSE        To clear a screen/console channel.
* DESCRIPTION    Clears the screen channel whose ID is supplied in A0.
* INPUTS:
*                A0.L = channel ID
* OUTPUTS:
*                D0 = Error code
*                Z flag set if no errors, unset otherwise.
*-----------------------------------------------------------------------------
cls              move.l   d1/d3/a1,-(a7)  ; Corrupted by SD_CLEAR
                 moveq    #sd_clear,d0    ; SD_CLEAR defined in GWASL
                 moveq    #-1,d3          ; Infinite timeout
                 trap     #3              ; CLS the window
                 move.l   (a7)+,d1/d3/a1  ; Restore corrupted registers
                 tst.l    d0              ; Set Z flag if all ok
                 rts
```

Listing 9.4: A Subroutine Example

In the above example, I have extended the 'cls' code from our original subroutine as follows:

- - I have added a documentation header comment.
- - I have preserved D3 because I use it in the code myself.
- - I have preserved D1 and A1 because the QDOSMSQ documentation states that these two registers are 'undefined' on return from the system trap `SD_CLEAR`.
- - I have restored all 3 of these registers before the `RTS`.
- - I have added a TST.L D0 instruction to set the Z flags according to whether an error was detected or not.

Note that although D0 is used by the code and by the system call, I have not preserved it. This is quite simply because I use D0 to return any error codes back to the caller. As I have documented its corruption in the header, I assume that the user of the subroutine will read this and know all about it!

Bullet proofing the code like this helps to reduce unexpected bugs in your programs when you forget to save a register and after a subroutine call, assume it still has the same value as before. I know, I have been there. Of course, there is not much you can do to prevent the documentation you use from being wrong (been there too) but at least you did your best!

9.5 The Subroutine Library

Onwards with the code for my (useful) subroutines.

9.6 STR_COPY

```
*------------------------------------------------------------------------------
* NAME           STR_COPY
*------------------------------------------------------------------------------
* DEPENDENCY     None
* PURPOSE        Copy the string at (A2) over the string at (A1).
* DESCRIPTION    Copy the string whose address is passed in A2 over the
*                string whose address is passed in A1 thus overwriting
*                the old contents of the receiving string.
* INPUTS:
*                A1.L = Address of the receiving string
*                A2.L = Address of the sending string
* OUTPUTS:
*                A1.L = Address of the receiving string (preserved)
*                A2.L = Address of the sending string (preserved)
*------------------------------------------------------------------------------
str_copy     movem.l d0/a1-a2,-(a7)   ; Preserve working register
             move.w  (a2)+,d0         ; Get size of 'from' string
             move.w  d0,(a1)+         ; Set new size of 'to' string
             bra.s   sc_next          ; Skip the dbra stuff first time
sc_moveb     move.b  (a2)+,(a1)+      ; Move a single byte
sc_next      dbra    d0,sc_moveb      ; And the rest
             movem.l (a7)+,d0/a1-a2   ; Restore working registers
             rts                      ; Exit
```

Listing 9.5: STR_COPY

9.7 STR_APPEND

```
*------------------------------------------------------------------------------
* NAME            STR_APPEND
*------------------------------------------------------------------------------
* DEPENDENCY      STR_COPY
* PURPOSE         Append string at (A2) to the end of string at (A1).
* DESCRIPTION     Copy the string whose address is passed in A2 to the
*                 end of the string whose address is passed in A1. The
*                 old contents of both strings will be preserved -
*                 except A1 which will be extended of course!
* INPUTS:
*                 A1.L = Address of the receiving string
*                 A2.L = Address of the sending string
* OUTPUTS:
*                 A1.L = Address of the receiving string (preserved)
*                 A2.L = Address of the sending string (Preserved)
*------------------------------------------------------------------------------
str_append  movem.l  d0/a1-a2,-(a7)  ; Save the working register
            move.w   (a2)+,d0        ; Size of 'from' string
            move.w   (a1),d1         ; Size of 'to' string
            add.w    d0,(a1)+        ; New size of 'to' string
            adda.w   d1,a1           ; New 'to' string end position
            bra.s    sc_next         ; Copy bytes over using STR_COPY
                                     ; D0 is restored by STR_COPY
                                     ; STR_APPEND exits via STR_COPY.
```

Listing 9.6: STR_APPEND

9.8 STR_REVERSE

```
*------------------------------------------------------------------------------
* NAME            STR_REVERSE
*------------------------------------------------------------------------------
* DEPENDENCY      None
* PURPOSE         Reverse the bytes in the string at (A1).
* DESCRIPTION     Reverses the bytes in the string with address (A1).
* INPUTS:
*                 A1.L = Address of the string to be reversed
* OUTPUTS:
*                 A1.L = Address of the string to be reversed (Preserved)
*------------------------------------------------------------------------------
str_reverse move.l   d0-d1/a1-a2,-(a7)   ; Save working registers
            move.l   a1,a2               ; Copy start address
            move.w   (a1)+,d0            ; Fetch length word
            beq.s    sr_quit             ; Nothing to do
            adda.w   d0,a2               ; Near the end of the string
            addq.l   #1,a2               ; The last char in the string
            lsl.w    #1,d0               ; D0 = INT(D0/2)
            bra.s    sr_next             ; Skip the first one for DBRA
sr_loop     move.b   (a2),d1             ; Fetch the last character
            move.b   (a1),(a2)           ; Move the first byte to last
            move.b   d1,(a1)+            ; Move the last byte to first
            subq.l   #1,a2               ; And adjust last
sr_next     dbra     d0,sr_loop          ; And do the rest
sr_quit     movem.l  (a7)+,d0-d1/a1-a2   ; Restore the working registers
            rts
```

Listing 9.7: STR_REVERSE

9.9 STR_INSERT

```
*------------------------------------------------------------------------------
* NAME          STR_INSERT
*------------------------------------------------------------------------------
* DEPENDENCY    STR_APPEND
* PURPOSE       Insert string at (A2) into string at (A1) at pos D0.W.
* DESCRIPTION   Inserts the string with address at (A2) into the string
*               with address (A1) at the position passed in D0.W so the
*               first character in the inserted string is at (A1,D0.W)
*               after the insertion. (0 is the very first character!)
*               If D0 >= length (A1) then call STR_APPEND.
* INPUTS:
*               A1.L = Address of the receiving string
*               A2.L = Address of the string to be inserted
*               D0.W = Position (starting at 0) where to insert before
* OUTPUTS:
*               D0 = Error code
*               A1.L = Address of the receiving string (preserved)
*               A2.L = Address of the string to be inserted (preserved)
*               Z flag set if no errors, unset otherwise.
*------------------------------------------------------------------------------
str_insert  cmp.w    d0,(a1)           ; Are we appending perhaps?
            bge      str_append        ; Yes, easy case to deal with!
            tst.w    d0                ; Is there anything in D0?
            bge.s    si_ok             ; Yes, negatives are bad!
            moveq    #-15,d0           ; Bad parameter
            rts                        ; Z is unset, D0 = error code

si_ok       movem.l  d1/a1-a4,-(a7)    ; Save those workers
            move.l   a1,a3             ; A3 = Address of A1 string
            adda.w   (a1),a3           ; Plus the size ...
            addq.l   #2,a3             ; A3 = last char of string +1
            move.l   a3,a4             ; A4 = new last char afterwards
            adda.w   (a2),a4           ; Add the extra length
            addq.l   #2,a4             ; And now we are there (+1)
            move.w   (a2),d1           ; Size of inserted string
            bra.s    si_dnext          ; Skip dbra
si_dmove    move.b   -(a3),-(a4)       ; Move a byte
si_dnext    dbra     d1,si_dmove       ; Do the rest
            move.w   (a2),d1           ; Refetch the inserted length
            adda.w   d1,a2             ; A2 nearly at the last char
            addq.w   #2,a2             ; One past the last character
            bra.s    si_inext          ; Skip dbra stuff
si_imove    move.b   -(a3),-(a4)       ; Insert a byte
si_inext    dbra     d1,si_imove       ; Insert the rest
            movem.l  (a7)+,d1/a1-a4    ; Restore those workers
            clr.l    d0                ; No errors
            rts
```

Listing 9.8: STR_INSERT

9.10 STR_COMP

```
*------------------------------------------------------------------------------
* NAME          STR_COMP
*------------------------------------------------------------------------------
```

```
* DEPENDENCY     None
* PURPOSE        To compare two strings for exact equality
* DESCRIPTION    Compare the strings at (A1) and (A2) for equality.
*                Numbers in the string are considered as well.
*                Equivalent to 'IF (A1$ = A2$)'
* INPUTS:
*                A1.L = First string
*                A2.L = Second string
* OUTPUTS:
*                D0 = Result of comparison.
*                     -1 = A1 string is < A2 string
*                      0 = A1 string = A2 string
*                     +1 = A1 string > A2 string
*                A1.L = First string (preserved)
*                A2.L = Second string (preserved)
*------------------------------------------------------------------------------
str_comp    movem.l a0-a2,-(a7)          ; Must preserve workers
            moveq   #2,d0                ; Case & numbers considered
sc_params   move.l  a1,a0                ; Uses different registers
            move.l  a2,a1                ; So swap them over
            move.w  UT_CSTR,a2           ; Fetch the vector address
            jsr     (a2)                 ; Compare strings using ROM
            movem.l (a7)+,a0-a2          ; Restore working registers
            tst.l   d0                   ; Make sure Z is set/unset
            rts
```

Listing 9.9: STR_COMP

9.11 STR_COMPI

```
*------------------------------------------------------------------------------
* NAME           STR_COMPI
*------------------------------------------------------------------------------
* DEPENDENCY     STR_COMP
* PURPOSE        To compare two strings for approximate equality
* DESCRIPTION    Compare the strings at (A1) and (A2) for equality with
*                numbers considered but not letter case.
*                Equivalent to 'IF (A1$ == A2$)'
* INPUTS:
*                A1.L = First string
*                A2.L = Second string
* OUTPUTS:
*                D0 = Result of comparison.
*                     -1 = A1 string is < A2 string
*                      0 = A1 string == A2 string
*                     +1 = A1 string > A2 string
*                A1.L = First string (preserved)
*                A2.L = Second string (preserved)
*------------------------------------------------------------------------------
str_compare movem.l a0-a2,-(a7)          ; Must preserve workers
            moveq   #3,d0                ; Numbers + Case insignificant
            bra     sc_params            ; Jump into STR_COMP
```

Listing 9.10: STR_COMPI

9.12 FILE_CLOSE

```
*------------------------------------------------------------------------------
* NAME            FILE_CLOSE
*------------------------------------------------------------------------------
* DEPENDENCY      None
* PURPOSE         Close the channel passed in A0
* DESCRIPTION     Close the file channel with QDOS ID in A0. To prevent
*                 any original QL systems from serious problems, checks
*                 for #0 being closed and ignores it.
* INPUTS:
*                 A0.L = Channel ID to be closed
* OUTPUTS:
*                 D0 is preserved as IO_CLOSE does not return errors
*                 except NOT OPEN and we ignore these here! The Z flag
*                 is indeterminate after this subroutine.
*                 A0.L is returned undefined to avoid channel reuse.
*------------------------------------------------------------------------------
file_close  cmpa.l  #0,a0               ; Test for SuperBasic #0
            beq.s   fc_exit             ; Ignore it
            move.l  d0,-(a7)            ; Preserve the worker
            moveq   #io_close,d0        ; Prepare to close it
            trap    #2                  ; Close it
            move.l  (a7)+,d0            ; Restore the worker
fc_exit     rts
```

Listing 9.11: FILE_CLOSE

9.13 FILE_OPEN

```
*------------------------------------------------------------------------------
* NAME            FILE_OPEN
*------------------------------------------------------------------------------
* DEPENDENCY      None
* PURPOSE         To open a file like 'OPEN #3,filename'
* DESCRIPTION     Opens a file in mode 0 (old exclusive device) The
*                 filename is passed in A0. The current job assumes
*                 ownership of the channel. May need a TRAP #4 before
*                 calling if the filename is relative A6 when called.
* INPUTS:
*                 A0.L = Pointer to filename
* OUTPUTS:
*                 A0.L = Channel id.
*                 D0 = Error code
*                 Z flag set if no errors, unset otherwise.
*------------------------------------------------------------------------------
file_open       movem.l d1-d3,-(a7)     ; Those workers need saving
                moveq   #0,d3           ; Old exclusing device mode
fo_params       moveq   #IO_OPEN,d0     ; Trap code
                moveq   -1,d1           ; Current job owns the channel
                trap  #2                ; Open it
                movem.l (a7)+,d1-d3     ; Restore workers
                tst.l   d0              ; Make sure Z is set/unset
                rts
```

Listing 9.12: FILE_OPEN

9.14 FILE_OPENIN

```
*------------------------------------------------------------------------------
* NAME            FILE_OPENIN
*------------------------------------------------------------------------------
* DEPENDENCY      FILE_OPEN
* PURPOSE         To open a file like 'OPEN_IN #3,filename'
* DESCRIPTION     Opens a file in mode 1 (old shared device) The
*                 filename is passed in A0. The current job assumes
*                 ownership of the channel. May need a TRAP #4 before
*                 calling if the filename is relative A6 when called.
* INPUTS:
*                 A0.L = Pointer to filename
* OUTPUTS:
*                 A0.L = Channel id.
*                 D0 = Error code
*                 Z flag set if no errors, unset otherwise.
*------------------------------------------------------------------------------
file_openin       movem.l d1-d3,-(a7)       ; Those workers need saving
                  moveq   #1,d3             ; Old shared device mode
                  bra     fo_params         ; Do the rest via FILE_OPEN
```

Listing 9.13: FILE_OPENIN

9.15 FILE_OPENNEW

```
*------------------------------------------------------------------------------
* NAME            FILE_OPENNEW
*------------------------------------------------------------------------------
* DEPENDENCY      FILE_OPEN
* PURPOSE         To open a file like 'OPEN_NEW #3,filename'
* DESCRIPTION     Opens a file in mode 2 (new exclusive device) The
*                 filename is passed in A0. The current job assumes
*                 ownership of the channel. May need a TRAP #4 before
*                 calling if the filename is relative A6 when called.
* INPUTS:
*                 A0.L = Pointer to filename
* OUTPUTS:
*                 A0.L = Channel id.
*                 D0 = Error code
*                 Z flag set if no errors, unset otherwise.
*------------------------------------------------------------------------------
file_opennew      movem.l d1-d3,-(a7)       ; Those workers need saving
                  moveq   #2,d3             ; New exclusive device mode
                  bra     fo_params         ; Do the rest via FILE_OPEN
```

Listing 9.14: FILE_OPENNEW

9.16 FILE_OPENOVER

```
*------------------------------------------------------------------------------
* NAME            FILE_OPENOVER
*------------------------------------------------------------------------------
* DEPENDENCY      FILE_OPEN
* PURPOSE         To open a file like 'OPEN_OVER #3,filename'
* DESCRIPTION     Opens a file in mode 3 (new overwrite device) The
*                 filename is passed in A0. The current job assumes
*                 ownership of the channel. May need a TRAP #4 before
*                 calling if the filename is relative A6 when called.
```

```
* INPUTS:
*               A0.L = Pointer to filename
* OUTPUTS:
*               A0.L = Channel id.
*               D0 = Error code
*               Z flag set if no errors, unset otherwise.
*------------------------------------------------------------------------
file_openover   movem.l d1-d3,-(a7)      ; Those workers need saving
                moveq   #3,d3            ; New overwrite device mode
                bra     fo_params        ; Do the rest via FILE_OPEN
```

Listing 9.15: FILE_OPENOVER

9.17 FILE_OPENDIR

```
*------------------------------------------------------------------------
* NAME          FILE_OPENDIR
*------------------------------------------------------------------------
* DEPENDENCY    FILE_OPEN
* PURPOSE       To open a file like 'OPEN_DIR #3,devicename'
* DESCRIPTION   Opens a file in mode 4 (directory) The filename is
*               passed in A0. The current job assumes ownership of
*               the channel. May need a TRAP #4 before calling if the
*               filename is relative A6 when called.
* INPUTS:
*               A0.L = Pointer to filename
* OUTPUTS:
*               A0.L = Channel id.
*               D0 = Error code
*               Z flag set if no errors, unset otherwise.
*------------------------------------------------------------------------
file_opendir    movem.l d1-d3,-(a7)      ; Those workers need saving
                moveq   #4,d3            ; Directory mode
                bra     fo_params        ; Do the rest via FILE_OPEN
```

Listing 9.16: FILE_OPENDIR

9.18 FILE_GET_HEAD

```
*------------------------------------------------------------------------
* NAME          FILE_GET_HEAD
*------------------------------------------------------------------------
* DEPENDENCY    None
* PURPOSE       To read the 64 bytes header for a file. (already open)
* DESCRIPTION   Reads a 64 byte file header for the open file with ID
*               in A0.L into the buffer whose address is passed
*               in A1.L. This buffer must be at least 64 bytes long!
* INPUTS:
*               A0.L = Channel ID
*               A1.L = Address of 64 byte buffer to put header into
* OUTPUTS:
*               D0 = Error code
*               D1 = Size of header read into buffer
*               A0 = Channel id (preserved)
*               A1 = Address of buffer (preserved)
*               Z flag set if no errors, unset otherwise.
*------------------------------------------------------------------------
```

```
file_get_head   movem.l d2-d3/a0-a1,-(a7) ; Save working registers
                moveq   #FS_HEADR,d0      ; Get trap code
                moveq   #64,d2            ; Buffer size
fgh_rest        moveq   #-1,d3            ; Infinity is a big thing
                trap    #3                ; Do it
                movem.l (a7)+,d2-d3/a0-a1 ; Restore workers
                tst.l   d0                ; Set flags
                rts                       ; Return to caller
```

Listing 9.17: FILE_GET_HEAD

9.19 FILE_SET_HEAD

```
*------------------------------------------------------------------------------
* NAME          FILE_SET_HEAD
*------------------------------------------------------------------------------
* DEPENDENCY    FILE_GET_HEAD
* PURPOSE       To write a 64 bytes header for a file. (already open)
* DESCRIPTION   Writes a 64 byte file header for the open file with ID
*               in A0.L from the buffer whose address is passed in
*               A1.L. This buffer must be at least 64 bytes long!
* INPUTS:
*               A0.L = Channel ID
*               A1.L = Address of 64 byte buffer holding the header
* OUTPUTS:
*               D0 = Error code
*               D1 = Size of header written from buffer
*               A0 = Channel id (preserved)
*               A1 = Address of buffer (preserved)
*               Z flag set if no errors, unset otherwise.
*------------------------------------------------------------------------------
file_set_head   movem.l d2-d3/a0-a1,-(a7) ; Save working registers
                moveq   #FS_HEADS,d0      ; Get trap code
                bra     fgh_rest          ; Do rest via FILE_GET_HEAD
```

Listing 9.18: FILE_SET_HEAD

9.20 PRINT

```
*------------------------------------------------------------------------------
* NAME          PRINT
*------------------------------------------------------------------------------
* DEPENDENCY    None
* PURPOSE       To send the string at (A1) to the channel in A0.
* DESCRIPTION   This routine prints a QDOS string (word then bytes) to
*               the channel ID passed in A0. The string starts at A1.
* INPUTS:
*               A0.L = Channel ID
*               A1.L = Address of a QDOS format string to be printed.
* OUTPUTS:
*               D0 = Error code
*               A0 = Channel id (preserved)
*               A1 = Address of buffer (preserved)
*               Z flag set if no errors, unset otherwise.
*------------------------------------------------------------------------------
print           move.l a1, -(a7)          ; Preserve the buffer address
                movea.w ut_mtext,a2       ; Print a string utility
```

```
                jsr      (a2)              ; Print it
                move.l   (a7)+,a1          ; Restore the buffer address
                tst.l    d0                ; Check for errors
                rts
```

Listing 9.19: PRINT

9.21 LINE_FEED

```
*------------------------------------------------------------------------------
* NAME          LINE_FEED
*------------------------------------------------------------------------------
* DEPENDENCY    None
* PURPOSE       To send a linefeed character to the channel in A0.
* DESCRIPTION   This routine prints a linefeed character to the channel
*               ID passed in A0.
* INPUTS:
*               A0.L = Channel ID
* OUTPUTS:
*               D0 = Error code
*               A0 = Channel id (preserved)
*               Z flag set if no errors, unset otherwise.
*------------------------------------------------------------------------------
line_feed       movem.l  d1/a1,-(a7)       ; Preserve registers
                moveq    #io_sbyte,d0      ; Send one byte to channel
                moveq    #linefeed,d1      ; Byte to send = linefeed
                moveq    #infinite,d3      ; Timeout
                trap     #3                ; Do it
                movem.l  (a7)+,d1/a1       ; Restore
                tst.l    d0                ; Set Z if errors
                rts
```

Listing 9.20: LINE_FEED

9.22 INPUT

```
*------------------------------------------------------------------------------
* NAME          INPUT
*------------------------------------------------------------------------------
* DEPENDENCY    None
* PURPOSE       To obtain input from the user via the channel ID in A0.
* DESCRIPTION   This routine allows the user to type into a buffer
*               (which is part of this subroutine) up to a maximum of
*               256 bytes. A channel ID in A0 is used.
* INPUTS:
*               A0.L = Channel ID
* OUTPUTS:
*               D0 = Error code
*               D1.W = Number of characters typed EXCLUDING ENTER
*                      if D1.W = 0, user simply pressed ENTER.
*               A0 = Channel id (preserved)
*               A1 = Start of buffer (word count of string user typed)
*               Z flag set if no errors, unset otherwise.
*------------------------------------------------------------------------------
input       movem.l  d2-d3,-(a7)           ; Preserve working registers
            lea      i_buffer+2,a1         ; Our buffer address plus 2
            move.l   a1,-(a7)              ; Save it on the stack
```

```
                moveq   #io_fline,d0            ; Input some bytes (inc LF)
                moveq   #256,d2                 ; Buffer size maximum
                moveq   #infinite,d3            ; Inifinite timeout
                trap    #3

                move.l  (a7)+,a1                ; Restore buffer pointer
                subq.w  #1,d1                   ; Subtract LF character
                move.w  d1,-(a1)                ; Save length and set A1
                movem.l (a7)+,d2-d3             ; Restore those workers
                tst.l   d0                      ; Did it all work?
                rts

i_buffer        ds.w    128+1                   ; 256 chars + 1 word
```

Listing 9.21: INPUT

9.23 JOB_HEADER

```
*------------------------------------------------------------------------------
* NAME            JOB_HEADER
*------------------------------------------------------------------------------
* DEPENDENCY      None
* PURPOSE         Code required to define a QDOSMSQ job header.
* DESCRIPTION     Defines a job header ready to be filled in by the user.
*                 The user will fill in his/her own jobname between the
*                 quotes and the assembler will do the rest.
* INPUTS:         None.
* OUTPUTS:        None.
*------------------------------------------------------------------------------
start           bra.s   prog_start              ; Short jump to program start
                dc.l    0                       ; Spare.
                dc.w    $4afb                   ; Job id must be at location 6
prog_name       dc.w    prog_start-prog_name-2  ; Length of job name
                dc.b    ''                      ; YOUR JOBNAME HERE

prog_start      PUT YOUR CODE HERE
```

Listing 9.22: JOB_HEADER

9.24 MEM_ALLOC

```
*------------------------------------------------------------------------------
* NAME            MEM_ALLOC
*------------------------------------------------------------------------------
* DEPENDENCY      None
* PURPOSE         Allocate an area of memory on the heap.
* DESCRIPTION     Allocate an area of memory, size as per D0.L, and
*                 return the address of the allocated area in A0.L. D0
*                 is set to any error code and the Z flag will be set if
*                 no errors occurred, reset otherwise.
* INPUTS:
*                 D0.L = Size, in bytes, of memory area to be allocated
* OUTPUTS:
*                 A0.L = Base address of the memory area allocated
*                 D0 = Error code
*                 Z flag set if no errors, unset otherwise.
*------------------------------------------------------------------------------
```

```
mem_alloc       movem.l  d1-d3/a1-a3,-(a7)   ; Save working registers
                move.l   d0,d1               ; Space required in D1
                moveq    #MT_ALCHP,d0        ; Set the trap
                moveq    #-1,d2              ; For the current job
                trap     #1                  ; Do it
                movem.l  (a7)+,d1-d3/a1-a3   ; Restore working registers
                tst.l    d0                  ; Set flags
                rts
```

Listing 9.23: MEM_ALLOC

9.25 MEM_DEALLOC

```
*-------------------------------------------------------------------------
* NAME          MEM_DEALLOC
*-------------------------------------------------------------------------
* DEPENDENCY    None
* PURPOSE       Deallocate an already allocated area of memory
* DESCRIPTION   Deallocate a previously allocated area of memory, the
*               address of which is passed in A0.L.
* INPUTS:
*               A0.L = Address of area to deallocate
* OUTPUTS:
*               A0.L = zero to avoid using the memory again!
*               D0 = Error code
*               Z flag set if no errors, unset otherwise.
*-------------------------------------------------------------------------
mem_dealloc     movem.l  d1-d3/a1-a3,-(a7)   ; Save working registers
                moveq    #MT_RECHP,d0        ; Set the trap
                trap     #1                  ; Do it
                movem.l  (a7)+,d1-d3/a1-a3   ; Restore registers
                suba.l   a0,a0               ; Blank the memory address
                tst.l    d0                  ; Set flags
                rts
```

Listing 9.24: MEM_DEALLOC

9.26 SCR_MODE

```
*-------------------------------------------------------------------------
* NAME          SCR_MODE
*-------------------------------------------------------------------------
* DEPENDENCY    None
* PURPOSE       Check the mode & set if required
* DESCRIPTION   Checks for the mode passed in D0 and if not correct,
*               change to that mode.
* INPUTS:
*               D0.B = 4 or 8 for required mode
* OUTPUTS:
*               D0 = Error code
*               Z flag set if no errors, unset otherwise.
*-------------------------------------------------------------------------
scr_mode        move.l    d1-d2/d7/a3,-(a7)  ; Save working registers
                move.b    d0,d7              ; Save required mode
                cmpi.w    #4,d7              ; Is mode 4 required?
                bne.s     scrm_8             ; Nope.
                clr.b     d7                 ; Mode 4 requires 0
```

```
scrm_8          moveq    #mt_dmode,d0
                moveq    #-1,d1              ; Read current mode
                moveq    #-1,d2              ; Read current display type
                trap     #1                  ; Do it
                tst.l    d0                  ; Did it work?
                bne.s    scrm_exit           ; No, bale out
                cmp.b    d1,d7               ; Correct mode?
                beq.s    scrm_exit           ; Don't set mode if ok
                moveq    #mt_dmode,d0        ; Else, set it
                move.b   d7,d1               ; Get the mode from D7
                trap     #1                  ; Set mode
                move.l   (a7)+,d1-d2/d7/a3   ; Restore registers
                tst.l    d0                  ; Set Z flag if ok
scrm_exit       rts
```

Listing 9.25: SCR_MODE

9.27 CLS

```
*------------------------------------------------------------------------------
* NAME            CLS
*------------------------------------------------------------------------------
* DEPENDENCY      None
* PURPOSE         To clear a screen/console channel.
* DESCRIPTION     Clears the screen channel whose ID is supplied in A0.
* INPUTS:
*                 A0.L = channel ID
* OUTPUTS:
*                 D0 = Error code
*                 A0.L = channel ID (preserved)
*                 Z flag set if no errors, unset otherwise.
*------------------------------------------------------------------------------
cls             move.l   d1/d3/a1,-(a7) ; These are corrupted
                moveq    #sd_clear,d0   ; CLS
                moveq    #-1,d3         ; Infinite timeout
                trap     #3             ; CLS the window
                move.l   (a7)+,d1/d3/a1 ; Restore corrupted registers
                tst.l    d0             ; Set Z flag if ok
                rts
```

Listing 9.26: CLS

9.28 SCR_PAPER

```
*------------------------------------------------------------------------------
* NAME            SCR_PAPER
*------------------------------------------------------------------------------
* DEPENDENCY      None
* PURPOSE         Set the PAPER colour for the given channel ID.
* DESCRIPTION     Sets the paper colour for the screen channel whose ID
*                 is passed in A0, to the colour code supplied in D0.W.
* INPUTS:
*                 D0.W = colour code for paper colour
*                 A0.L = Channel ID.
* OUTPUTS:
*                 D0 = Error code
*                 A0.L = channel ID (preserved)
```

```
*               Z flag set if no errors, unset otherwise.
*------------------------------------------------------------------------
scr_paper       move.l    d1/d3/a1,-(a7)  ; These will be corrupted
                move.w    d0,d1           ; Get the paper colour
                moveq     #sd_clear,d0    ; CLS
                moveq     #-1,d3          ; Infinite timeout
                trap      #3              ; Set PAPER colour (not STRIP)
                move.l    (a7)+,d1/d3/a1  ; Restore corrupted registers
                tst.l     d0              ; Set Z flag if all ok
                rts
```

Listing 9.27: SCR_PAPER

9.29 SCR_PAPER_SB

```
*------------------------------------------------------------------------
* NAME          SCR_PAPER_SB
*------------------------------------------------------------------------
* DEPENDENCY    SCR_PAPER
* DEPENDENCY    SCR_STRIP
* PURPOSE       Set the PAPER & STRIP colour for the given channel ID.
* DESCRIPTION   Sets the paper & strip colour for the screen channel
*               whose ID is passed in A0, to the colour code supplied
*               in D0.W. Works like SuperBasic's PAPER command.
* INPUTS:
*               D0.W = colour code for paper & strip colour
*               A0.L = Channel ID.
* OUTPUTS:
*               D0 = Error code
*               A0.L = channel ID (preserved)
*               Z flag set if no errors, unset otherwise.
*------------------------------------------------------------------------
scr_paper_sb    move.w   d0,d1            ; Save the colour
                bsr      scr_paper        ; Set the paper colour
                bne.s    spsb_exit        ; Tets for errors
                move.w   d1,d0            ; Get the colour code again
                bsr      scr_strip        ; Set the strip colour
scsb_exit       rts
```

Listing 9.28: SCR_PAPER_SB

9.30 SCR_INK

```
*------------------------------------------------------------------------
* NAME          SCR_INK
*------------------------------------------------------------------------
* DEPENDENCY    None
* PURPOSE       Set the INK colour for the given channel ID.
* DESCRIPTION   Sets the ink colour for the screen channel whose ID is
*               passed in A0, to the colour code supplied in D0.W.
* INPUTS:
*               D0.W = colour code for ink colour
*               A0.L = Channel ID.
* OUTPUTS:
*               D0 = Error code
*               A0.L = channel ID (preserved)
*               Z flag set if no errors, unset otherwise.
```

```
*------------------------------------------------------------------------------
scr_ink         move.l    d1/d3/a1,-(a7) ; These will be corrupted
                move.w    d0,d1          ; Get the ink colour
                moveq     #sd_clear,d0   ; CLS
                moveq     #-1,d3         ; Infinite timeout
                trap      #3             ; Set INK colour
                move.l    (a7)+,d1/d3/a1 ; Restore registers
                tst.l     d0             ; Set Z flag if all ok
                rts
```

Listing 9.29: SCR_INK

9.31 SCR_STRIP

```
*------------------------------------------------------------------------------
* NAME            SCR_STRIP
*------------------------------------------------------------------------------
* DEPENDENCY      None
* PURPOSE         Set the STRIP colour for the given channel ID.
* DESCRIPTION     Sets the strip colour for the screen channel whose ID
*                 is passed in A0, to the colour code supplied in D0.W.
* INPUTS:
*                 D0.W = colour code for strip colour
*                 A0.L = Channel ID.
* OUTPUTS:
*                 D0 = Error code
*                 A0.L = channel ID (preserved)
*                 Z flag set if no errors, unset otherwise.
*------------------------------------------------------------------------------
scr_strip       move.l    d1/d3/a1,-(a7)  ; These will be corrupted
                move.w    d0,d1           ; Get the paper colour
                moveq     #sd_clear,d0    ; CLS
                moveq     #-1,d3          ; Infinite timeout
                trap      #3              ; Set STRIP colour (not PAPER)
                move.l    (a7)+,d1/d3/a1  ; Restore corrupted registers
                tst.l     d0              ; Set Z flag if all ok
                rts
```

Listing 9.30: SCR_STRIP

9.32 COLOURS

```
*------------------------------------------------------------------------------
* NAME            COLOURS
*------------------------------------------------------------------------------
* DEPENDENCY      None
* PURPOSE         Define names for the various QDOSMSQ colours
* DESCRIPTION     Not really a subroutine, a set of equates which define
*                 names for the 8 colours available on a 'standard'
*                 QDOSMSQ machine.
* INPUTS:         None
* OUTPUTS:        None
*------------------------------------------------------------------------------
black           equ       0
blue            equ       1
red             equ       2
magenta         equ       3
```

```
green           equ     4
cyan            equ     5
yellow          equ     6
white           equ     7
```

Listing 9.31: COLOURS

9.33 The Librarian

Ok, so there you have a few of my favourite routines, all you need now is a librarian to sort them out for you. Ok, I give up, here is one for you - this is very basic and not super at all (sorry for that pun) it is up to you to expand on this if you want.

Some suggestions would be:

- Make PE aware?
- Add better/some error trapping.
- Save the dependencies so that the user need not enter them manually.
- Check what has just been entered with what has already been entered to avoid duplications.
- Reduce the number of file open/closes etc (On the Library file)
- Convert to assembler - Ha, now you're quaking in your boots !

I have omitted line numbers from the following listing.

```
CLS
Output = 3
INPUT 'Library name: ' LibraryName$
INPUT 'Output file name: '; Output$
OPEN_NEW #Output, Output$
:
REPeat main_loop
    INPUT 'Routine name (ENTER to quit): '; Name$
    IF (Name$ = '')
        EXIT MainLoop
    END IF
    ExtractName Name$
END REPeat MainLoop
:
CLOSE #Output
PRINT "Done."
STOP
:
:
DEF PROCedure ExtractName(ReqdName$)
    LOCal A$, Library, FoundName$
    Library = Output + 1
    OPEN_IN #Library, LibraryName$
    REPeat LibLoop
        IF (EOF #Library)
            EXIT LibLoop
        END IF
        INPUT #Library, A$
        IF (A$(1 TO 6) == "* NAME")
            FoundName$ = A$(17 TO)
            IF (FoundName$ == ReqdName$)
                PRINT "Found subroutine: " & ReqdName$
                GetDependencies(Library)
```

```
                ExtractCode(Library)
                CLOSE #Library
                RETurn
            ENDIF
        END IF
    END REPeat LibLoop
    PRINT "Cannot find: " & ReqdName$
END DEFine ExtractName
:
:
DEF PROCedure GetDependencies(Channel)
    LOCal A$
    REPeat DependLoop
        IF (EOF #Channel)
            RETurn
        END IF
        INPUT #Channel, A$
        IF (A$(1 TO 12) == "* DEPENDENCY")
            IF (A$(17 TO 20) == "None")
                PRINT "No dependencies"
                Return
            END IF
            PRINT "Dependency required: " & A$(17 TO)
        END IF
        IF (A$(1 TO 9) == "* PURPOSE"
            RETurn
        END IF
    END REPeat DependLoop
END DEFine GetDependencies
:
:
DEF PROC ExtractCode(Channel)
    LOCal A$
    REPeat FindCodeLoop
        IF (EOF #Channel)
            RETurn
        END IF
        INPUT #Channel, A$
        IF (A$(1 TO 5) == "*----"
            EXIT FindCodeLoop
        END IF
    END REPeat FindCodeLoop
    REPeat WriteCodeLoop
        IF (EOF #Channel)
            RETurn
        END IF
        INPUT #Channel, A$
        IF (A$(1 TO 5) == "*----"
            EXIT WriteCodeLoop
        END IF
        PRINT #Output, A$
    END REPeat WriteCodeLoop
    PRINT "Extracted."\\
END DEFine ExtractCode
```

9.33.1 So how does this lot work?

After asking for your details etc, it simply enters a loop asking you for the next routine to be extracted. This name is passed to ExtractName which opens the library file and scans it looking for all those lines which start '* NAME'. Once it finds one, it tests to see if this line includes the name you are looking for.

Note that this version of the program assumes you are using *exactly* the same format in your comments as I am above and as per the following description:

- Column 1 = An asterisk, the comment marker for most assemblers I have used.
- Column 2 = A space.
- Columns 3 to 16 = Parameter name, eg NAME, DEPENDENCY etc.
- Columns 17 onwards = Parameter details etc.

If the name found is the same as the one you requested, the dependencies are extracted and listed. You are advised to note these dependencies and enter them as the next routine to extract. Try not to duplicate names etc as the program doesn't test for duplicates.

Once all dependencies have been listed, The code is extracted and written to the output file.

A sample session follows:

```
Library name: Win1_GWASL_Library_lib
Output file name: Win1_source_MyNextProject_asm
Routine name (ENTER to quit): Colours
Found subroutine: COLOURS
No dependencies
Extracted.

Routine name (ENTER to quit): Scr_paper_sb
Found subroutine: SCR_PAPER_SB
Dependency required: SCR_PAPER
Dependency required: SCR_STRIP
Extracted.

Routine name (ENTER to quit): Scr_paper
Found subroutine: SCR_PAPER
No dependencies
Extracted.

Routine name (ENTER to quit): SCR_STRIP
Found subroutine: SCR_STRIP
No dependencies
Extracted.

Routine name (ENTER to quit):
Done.
```

So there you have it and now you can enhance it as required to suit your own purposes. Remember, my version expects the comments to be in the format given above. Additionally, no comments are written to the output file but you can easily amend the code in ExtractCode to do the needful. Enjoy.

9.34 Coming Up...

In the next chapter we shall be looking at the thorny subject of coding single and doubly linked lists in assembler.

IV SuperBasic, QDOS and Other Interesting Stuff. Part 2

10. Linked Lists

10.1 Introduction

This chapter introduces you to linked lists. In the QDOSMSQ operating system, linked lists are used in many places - and you can use them in your own code as well. This chapter tells you how.

10.2 Linked Lists

Linked lists are used within QDOSMSQ to hold details of the directory devices installed on the system, interrupt routines and so on, but what are they exactly?

Imagine that you are writing a program, and you decide that you need some storage for some data, let's say a list of people's names and addresses. So, how about an array? Well, the problem with that is how many entries are you going to allow? If you don't allow enough entries, you won't have much of an address book. If you have too many entries then you are wasting space. If you sell the program, or give it away, then you need to consider the needs of people other than yourself - some will need a few entries and others, much more. How do you cope?

Well, a linked list could be the answer. You start off with no storage defined at all, except for a single, maybe two, variables which hold the address in memory of the beginning (and maybe the end) of your list of addresses. As you add new contacts to your address book, each one is created at some 'random' location in memory and linked into your existing list of contacts. Hence, you have a linked list.

In a linked list, each entry is called a node, and the pointer to the very first entry in the list is known as the root node.

In memory an array, of 10 entries of 100 byte long strings, is consecutive. Don't forget the strings have a word at the start defining their length, so each entry is actually 102 bytes long. If the first entry is located at address 1000 then the next entry is at address 1102, the next at 1204 and so on. There are no gaps between entries and you can quickly calculate the start address of any particular entry as

$1000 + (INDEX * 104)$

where INDEX is the entry you are looking for, starting at zero.

In a linked list, the nodes are potentially all over the place, the first might be at address 1000, the second at 2000, the third at 1200 and so on. There is no logical order to the locations and you cannot calculate the address of a particular node using any formula as you can with arrays.

What you can do, however, is store away the address of the first node in a special node known as the root node, and from that, you can navigate along the list from start to finish by finding the address of the next node from the data stored in the individual nodes. Our 100 byte long strings would be 106 bytes allowing 4 bytes to store the memory address of the 'next' entry in the list and the obligatory 2 byte length word. However, think about that 102 bytes in each entry of the array - you might not need all 102 bytes. In our linked list, each node will have 4 bytes for the pointer and only as much space as is required by the data, so each node need not be $102 + 4$ bytes long. Another saving over the array.

A linked list can be thought of like an old program on UK TV, *Treasure Hunt*, where Aneka Rice used to zoom around the country in a helicopter picking up clues in one location which told her where to go for the next clue and so on, until she found the 'treasure' at the end of the list of clues. This is exactly what a linked list is.

If we have a node in our list defined as follows, then we can see how it looks in memory below. Each node in the list will look like Figure 10.1 with a 4 byte pointer at the start holding the address of the next node in the list, and everything from byte 4 onwards holding some form of desired data.

Figure 10.1: Linked List Node Structure.

The root node, as mentioned above, is special. It has no data part, only the pointer part, although it is not necessary for it not to be a full node, the data part will be empty in such cases.

The conceptual layout in memory is a bit like Figure 10.2 (using the addresses mentioned above and assuming the root node lives at address $ABCD):

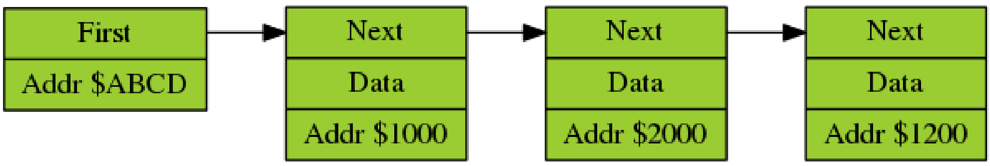


Figure 10.2: A Simple Linked List.

The lowest section of each node above simply shows an example address in memory where that particular node lives. It is not part of the node itself.

In physical terms there are, of course, no handy arrows. Using real values as described above in the pointer locations, it would look like Figure 10.3:

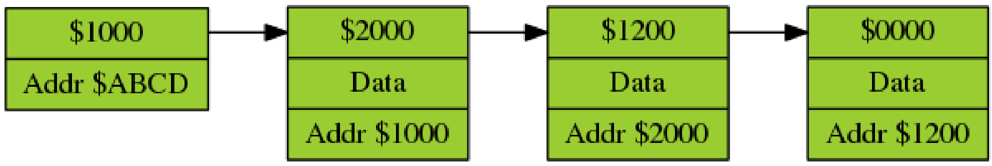


Figure 10.3: Memory Organisation of a Simple Linked List.

You can see from Figure 10.3 that the address of the following node is held in each node's first 4 bytes. The address of FIRST is actually somewhere in your program and your program only needs to allocate storage for the 4 bytes it takes to hold the address of the initial node in the list. FIRST is, of course, the root node of the list.

You must store the value zero in there before you go off adding nodes, you'll see this reason why below in the code to add a node.

10.2.1 Adding Nodes.

Adding a new node is simple, you allocate it on the heap, fill in the data part and add it to the front of the list. It is far easier to add a node at the start - address 1000 in the above example - than to have to work through the entire list to find the current end, and then add it there. This method takes longer and longer to carry out as you add extra entries to the list. Adding at the start of the list takes the same time regardless of how many entries are in the list.

As you add each node to the list, you copy the value in FIRST into the new node's NEXT pointer and put the address of the new node into FIRST. Sounds complicated, but here it is in code. If we assume that A0.L has the address of FIRST and that A1.L has the address of the node to be inserted into the list, as contrivingly demonstrated by these two lines:

```
Prelude     lea  FIRST,a0           ; Pointer to storage of first node
            lea  NewEntry,a1        ; Address of new node
```

Listing 10.1: Adding a Node - Prelude

Then adding a new node to a list is as simple as this:

```
AddNode     move.l  (a0),(a1)       ; Save first node in new node's NEXT
            move.l  a1,(a0)         ; Store new node in FIRST
            rts
```

Listing 10.2: Adding a Node

Nothing to it. The new node is always added at the start of the list, so the value in FIRST always points to the most recently added node. As you need to have zero in the NEXT pointer of the final node in the list, you can see why it was important to initialise the value in your programs FIRST variable to zero before adding any nodes. If you didn't have zero, you'd never know when the list was finished.

One thing, you don't want to allow the user to add the root node to its own list at any time, so best change the above code to prevent this from happening.

```
AddNode    cmpa.l   a0,a1        ; Don't allow the root node to be added
           beq.s    AddExit      ; Bale out quietly if attempted
           move.l   (a0),(a1)    ; Save first node in new node's NEXT
           move.l   a1,(a0)      ; Store new node in FIRST
AddExit    rts
```

Listing 10.3: A Better Way of Adding a Node

Another problem is when you try to add a node that is already there. So to be really careful, you could call the FindNode routine (coming soon - have patience!) prior to adding it in. However, as this scans the entire list until it finds or doesn't find the new node, it could add quite a lot of time to the simple exercise of adding a new node.

If you wrote the program, and you are allocating nodes on the heap each time, then don't bother attempting to find the node in the list before you add it.

10.2.2 Deleting Nodes.

Deleting a node is slightly more difficult. The node to be deleted could be anywhere in the list, or not even in the list. How to find the correct node is the main problem. However, for the same of argument, assume that we have the node address to be removed in A1.L and the address of FIRST in A0.L after a successful 'find' operation, then removing the node at A1.L requires that we must navigate the list, as in the following explanation.

We must navigate the list because we don't know where in memory the node prior to the one we wish to delete is. We need to find it, because it has a NEXT pointer holding the address of the 'deleted' node and this has to be changed or we lose everything in the list after the deleted node.

As ever, the value in the NEXT area of the very last node in the list is always zero. That way, we know when we have hit the end of the list. Here's the pseudo code to delete the node at A1.L from the list beginning at (A0.L)

- If the node to be deleted is the root node (the list pointer in A0) then don't allow it to be deleted.
- Start of the main loop.
- If the value stored in the address that A0.L points to is equal to zero, we have been passed an incorect node address to delete. Exit from the loop with an error.
- If the value stored in the address that A0.L points to is not the same as the value in A1.L then copy the value in the address that A0.L points to into A0 and restart the main loop. Basically we have replaced the address in A0 with the NEXT address from the node we were just looking at.
- If the value stored in the address that A0.L points to is equal to the value in A1.L then we have found the node PRIOR to the node we wish to delete and so the node we are looking at has to have the NEXT address updated to bypass the node we wish to delete so that it now points to the NEXT address which is currently stored in the node we are deleting. Exit from the loop with no errors.
- End of main loop.

That's the pseudo code, here's the actual code. Using the same preliminary stuff as above to sort out initial values of A0.L and A1.L and a little bit extra to show whether errors have been detected or not, we begin with this:

```
Prelude    lea FIRST,a0       ; Pointer to root node
           lea OldNode,a1     ; Address of node to delete
```

```
            moveq  #ERR_EF,d0      ; End of file = node not found = error
```

Listing 10.4: Deleting a Node - Prelude

Now, here's the actual code to find and remove the requested node.

```
DelNode     cmpa.l  a0,a1          ; Don't allow the root node deletion
            beq.s  DelExit         ; Bale out with error if attempted

DelLoop     cmp.l  #0,(a0)         ; Reached the end yet?
            beq.s  DelExit         ; Yes, node not found, exit with error

            cmp.l  (a0),a1         ; Found the PRIOR node yet?
            bne.s  DelNext         ; No, skip deletion code & try again

DelFound    move.l  (a1),(a0)      ; PRIOR node NEXT = deleted node's NEXT
            moveq   #0,d0          ; Node found and deleted ok
            bra.s   DelExit        ; Bale out with no errors

DelNext     move.l  (a0),a0        ; A0 now holds the NEXT node in the list
            bra.s   DelLoop        ; Go around again

DelExit     tst.l  d0              ; Set zero flag for success
            rts
```

Listing 10.5: Deleting a Node

The above code returns with the Z flag set if the node was deleted from the list, and unset if the node was not in the list. This allows the calling code to handle and errors correctly.

10.2.3 Finding Nodes.

The first thing you must do when deleting a node is to actually find it. The code above assumes that A1 holds a valid node address in the list defined by A0. Having said that, the code is robust enough to know that programmers make errors and it can handle the problem of a node address being passed which is not in the list by virtue of the fact that it scans the list until it finds the node prior to the one we wish to delete. It has to work that way because we need to adjust the NEXT pointer in the prior node to point past the deleted node to its NEXT node - if you catch my drift?

The code to find an node in a list is dependant on the sort of data stored in each node. If you store strings, the some form of string comparison routine needs to be built in - does it compare on an equality basis ('AAA' = 'AAA') or nearly equal basis ('AAA' == 'aaa') and so on. You can use the built in QDOSMSQ routines to do the comparisons.

If the data in the nodes are numbers (integers of word or long length) then you can compare them directly. If they are QDOSMSQ floating point format numbers, you can use the built in arithmetic routines to compare them. Regardless of which method is used, you need to write your own code to compare two nodes, or a node and a value so that the find routine knows when it has found the correct entry.

Of course, it is quite simple to build a FindNode routine which doesn't know or care what sort of data the individual nodes contain, provided it is passed the address of a routine which does know and care. If the specification for said routine requires the Z flag to be set for found and unset for not found, it could look something like the following peseudo code.

Assume that A0.L holds the address of FIRST, A1.L holds a pointer to a routine which compares the node with a given value and A2.L holds a pointer to that value. The data that A2.L points to can be anything, the routine at (A1.L) does the working out, our FindNode simply calls the routine once for each node in the list until such time as it gets a set Z flag on return. The comparison routine gets passed a node address in A3.L.

- Start of the main loop.
- If the value stored in the address that A0.L points to is equal to zero, we have not found a node with the desired value. Exit the main loop with a NOT FOUND error.
- Copy the address at (A0.L) into A3 and call the routine to compare data. If it returns with the Z flag set, the address in (A0.L) is the address of the node prior to the node we were looking for, however, the address in A3.L is the address of our required node as it is taken from the NEXT pointer. Remember, we passed the NEXT address (A0.L) over to the routine, not the address of THIS node - A0.L. Exit from the loop with the Z flag set to indicate a found node.
- Copy the NEXT address from the node we are looking at into A0.L and go back to the start of the loop.
- End of main loop.

And here's the real code to do the finding for use. As ever, we start off with some contrived values.

```
Prelude     lea   FIRST,a0          ; Pointer to root node address
            lea   Compare,a1        ; Address of node comparison routine
            lea   Required,a2       ; Address of the data we are looking for
            moveq #ERR_NF,d0        ; Node not found = error
```

Listing 10.6: Finding a Node - Prelude

Now, here's the actual code to find a node in the list which holds the required value.

```
FindNode    cmp.l  #0,(a0)          ; Reached the end yet?
            beq.s  DelExit          ; Yes, node not found, exit with error

            move.l (a0),a3          ; Fetch the NEXT node address into A3.L
            jsr    (a1)             ; And jump into the comparison routine
            beq.s  FindFound        ; Looks like we found our node

FindNext    move.l (a0),a0          ; A0 now holds the NEXT node in the list
            bra.s  FindNode         ; Go around again

FindFound   moveq  #0,d0            ; Clear the error flag

FindExit    tst.l  d0               ; Set zero flag for success
            rts
```

Listing 10.7: Finding a Node

The following is an example of a compare routine to look at a long word of data in the node at (A3.L) and see if it is equal to the long word of data stored at (A2.L). Don't forget, the comparison routine must preserve A0, A1, A2 and D0 or it will all go horribly wrong. The following routine does exactly that, by the simple method of not actually using those registers at all!

```
NData    equ 4                     ; Offset to node's data part

Compare     cmp.l NData(A3),(A2)   ; Is the data = the value we want?
            rts                    ; Exit with Z set if so
```

Listing 10.8: Finding a Node - Data Comparison

If an attempt is made to find the root node, then it will fail.

So there you have three short but extremely powerful routines which make linked lists possible. At this point I have to mention that there are actually routines built into QDOSMSQ to do exactly the same work as the AddNode and DelNode routines above, but there is nothing like FindNode - which is a shame. However, you now know how to build linked lists and add and delete nodes. You also know how to find an entry in a linked list so that you can process it in some way.

10.2.4 The Code Wrapper.

Putting all of the above together and tying in some extras to allocate nodes etc, here is a small, but perfectly formed program to create a linked list. The following is a wrapper that we shall use to demonstrate first the single linked lists as explained above. Later on, when other types of linked list are explained, we shall drop in only the code we need for the demo

```
* ===================================================================
* A test harness 'job' for our linked lists code. What's the point of
* all the explanations if you can't test the code?
*
* This code is simply a wrapper to allow different demos to be slotted
* in to demonstrate the real code in the chapter, as opposed to the job
* code.
*
* The code being demonstrated is located at DEMO below. As new demos
* are required, only that bit should (!) need changing.
* ===================================================================

* -------------------------------------------------------------------
* These are offsets from the start of the job's dataspace where working
* variable are stored. The dataspace is held at (A4) in the job's code.
* -------------------------------------------------------------------
con_id      equ     0                        ; Id for title channel
con_id2     equ     4                        ; Id for main output

* -------------------------------------------------------------------
* These are simply user friendly names instead of numbers for various
* bits and bobs, colours etc.
* -------------------------------------------------------------------
black       equ     0                        ; Colour code for mode 4 black
red         equ     2                        ; Red
green       equ     4                        ; Green
white       equ     7                        ; White
linefeed    equ    10                        ; Linefeed character
oops        equ    -1                        ; General error code
err_nc      equ    -1                        ; NOT COMPLETE error code

* -------------------------------------------------------------------
* Constants for use with job control commands. (It doesn't matter if I
* have two names with the same value! )
* -------------------------------------------------------------------
infinite    equ     -1                       ; Infinite timeout
me          equ     -1                       ; Id for 'this' job


* -------------------------------------------------------------------
```

```
* Code starts here.
*------------------------------------------------------------------------------
start       bra.s     LinkList              ; 2 bytes short jump
            dc.l      0                     ; 4 bytes padding
            dc.w      $4afb
            dc.w      11                    ; Bytes in job's name
            dc.b      'LinkedLists',0       ; Bytes of job's name + padding

LinkList    adda.l    a6,a4                 ; A4.L = start of dataspace
            bsr       Mode4                 ; Set the screen mode
            bsr       Title                 ; Open the title window
            bsr       Output                ; Open the output window
            bsr       Headings              ; Display headings
            bsr       Demo                  ; Do the demo code
            bsr       Finished              ; Advise user that we are done

*------------------------------------------------------------------------------
* Code ends here.
*------------------------------------------------------------------------------
all_done    moveq     #mt_frjob,d0          ; Force Remove a job
            moveq     #me,d1                ; The current job
            move.l    d0,d3                 ; Error code for SuperBasic
            trap      #1                    ; Kill this job

            bra.s     all_done              ; Should never get here
```

Listing 10.9: Linked Lists - Wrapper - Part 1

Note The following code will be replaced by either the singly linked list or the doubly linked list demo code which follows at the end of this chapter. For now, however, it is a place holder.

```
*------------------------------------------------------------------------------
* The DEMO code starts here.
*------------------------------------------------------------------------------
Demo        rts

*------------------------------------------------------------------------------
* The DEMO code ends here.
*------------------------------------------------------------------------------
```

Listing 10.10: Linked Lists - Wrapper - Demo Placeholder

Note The following code is common to both demos.

```
*------------------------------------------------------------------------------
* Set mode 4 if not already set. Do not change from TV to monitor or
* vice versa. We must preserve the display type if we reset the mode.
*------------------------------------------------------------------------------
Mode4       moveq     #mt_dmode,d0
            moveq     #-1,d1                ; Read current mode
            moveq     #-1,d2                ; Read current display type
            trap      #1                    ; Do it
            tst.l     d0                    ; Did it work?
```

```
              bne       all_done            ; No, bale out, cannot continue

              tst.b     d1                  ; 0 in D1.B = Mode 4
              beq.s     ModeExit            ; No need to set mode 4
              moveq     #mt_dmode,d0
              clr.l     d1                  ; We need mode 4
              trap      #1                  ; Set mode 4 (d2 = disp type)
              tst.l     d0                  ; Check it
              bne       all_done            ; Bale out if errors detected
ModeExit      rts                           ; Done.

* ------------------------------------------------------------------------
* Mode 4 is in use. Open the title window at the top of the screen.
* ------------------------------------------------------------------------
Title         lea       con_def,a1          ; Window definition
              movea.w   ut_con,a2           ; Utility to define a window
              jsr       (a2)                ; Do it
              tst.l     d0                  ; Did it work ok?
              bne       all_done            ; No, exit program
              move.l    a0,con_id(a4)       ; Store title channel id
              rts                           ; Done

*-------------------------------------------------------------------------
* Definition for title window channel
*-------------------------------------------------------------------------
con_def       dc.b      red                 ; Border colour
              dc.b      1                   ; Border width
              dc.b      white               ; Paper/strip colour
              dc.b      black               ; Ink colour
              dc.w      448                 ; Width
              dc.w      24                  ; Height
              dc.w      32                  ; Start position x
              dc.w      16                  ; Start position y

* ------------------------------------------------------------------------
* Open the output window underneath the title one.
* ------------------------------------------------------------------------
Output        lea       con_def2,a1         ; Output window definition
              movea.w   ut_con,a2           ; Utility again
              jsr       (a2)                ; Do it
              tst.l     d0                  ; Did it work?
              bne       all_done            ; No, exit routine
              move.l    a0,con_id2(a4)      ; Store output channel id

              moveq     #0,d0               ; No errors detected
              rts

*-------------------------------------------------------------------------
* Definition for output window channel
*-------------------------------------------------------------------------
con_def2      dc.b      red                 ; Border colour
              dc.b      1                   ; Border width
              dc.b      white               ; Paper/strip colour
              dc.b      black               ; Ink colour
              dc.w      448                 ; Width
              dc.w      200                 ; Height
```

```
                dc.w    32                  ; X org
                dc.w    40                  ; Y org

* ----------------------------------------------------------------------
* Print the headings
* ----------------------------------------------------------------------
headings        movea.l con_id(a4),a0       ; Title channel id
                bsr.s   cls                 ; Clear screen
                lea     mes_title,a1        ; Title string
                bsr.s   prompt              ; Print title string
                rts

mes_title       dc.w    mes_end-mes_title-2
                dc.b    'Single Linked Lists'
mes_end         equ     *

* ----------------------------------------------------------------------
* Sign off message
* ----------------------------------------------------------------------
Finished        movea.l con_id2(a4),a0      ; Title channel id
                lea     end_title,a1        ; Title string
                bsr.s   prompt              ; Print title string
                bsr.s   input               ; Wait for ENTER
                rts

end_title       dc.w    end_end-end_title-2
                dc.b    linefeed,linefeed,'Press ENTER to quit: '
end_end         equ     *

* ======================================================================
* CLS:
* ======================================================================
* 1. Clear the (screen) channel whose id is in A0.
* ======================================================================
cls             moveq   #sd_clear,d0        ; CLS
                moveq   #infinite,d3        ; Infinite timeout
                trap    #3                  ; CLS title window
                rts

* ======================================================================
* Prompt:
* ======================================================================
* 1. Print the string at (A1) to the channel in A0.
*
* Z set if all ok, unset if not.
* ======================================================================
prompt          movea.w ut_mtext,a2         ; Print a string utility
                jsr     (a2)                ; Print it
                tst.l   d0                  ; Check for errors
                rts

* ======================================================================
* Input:
* ======================================================================
* Wait for user input from the channel id in A0.
*
```

```
* Returns the input length (not counting the ENTER character) in D1.W
* Returns the address of the first character in the buffer in A1.L
* Preserves the channel id in A0.L
* Z set if all ok, unset if not.
* =====================================================================
input       lea      buffer+2,a1          ; Our buffer address plus 2
            move.l   a1,-(a7)             ; Save it on the stack
            moveq    #io_fline,d0         ; Input some bytes (+ linefeed)
            moveq    #60,d2               ; Buffer size maximum
            moveq    #infinite,d3         ; Inifinite timeout
            trap     #3

            move.l   (a7)+,a1             ; Restore buffer pointer
            subq.w   #1,d1                ; Subtract the linefeed
            move.w   d1,-2(a1)            ; Store length in buffer
            tst.l    d0                   ; Did it all work?
            rts

buffer      ds.w     31                   ; 60 chars for input + 1 word

* =====================================================================
* hex_l:
* =====================================================================
* Convert a 4 byte value in D4.L to Hex in a buffer. Use the input
* buffer for the output and DOES NOT store the length word!
*
* Expects D4.L to hold the value.
* =====================================================================
hex_l       swap     d4              ; $ABCD -> $CDAB in D4
            bsr.s    hex_w           ; Convert the $AB part first
            swap     d4              ; $CDAB -> $ABCD again
*           drop into hex_w to convert the $CD part

* =====================================================================
* hex_w:
* =====================================================================
* Convert a 2 byte value in D4.W to Hex in a buffer.
*
* Expects D4.W to hold the value.
* Expects A1.L to point at the buffer.
* =====================================================================
hex_w       ror.w    #8,d4           ; $DE -> $ED in D4
            bsr.s    hex_b           ; Convert the $D part first
            rol.w    #8,d4           ; $ED -> $DE again
*           drop into hex_b to convert the $E part

* =====================================================================
* hex_b:
* =====================================================================
* Convert a 1 byte value in D4.B to Hex in a buffer.
*
* Expects D4.B to hold the value.
* Expects A1.L to point at the buffer.
* =====================================================================
hex_b       ror.b    #4,d4           ; Swap lower and higer nibbles
            bsr.s    hex_nibble      ; Print high nibble first
```

```
            rol.b   #4,d4           ; Swap back again
*           drop into hex_nibble to print the lower nibble

* ===================================================================
* hex_nibble:
* ===================================================================
* Convert a 4 bit value in D4.B to Hex in a buffer.
*
* Expects D4.B to hold the value.
* Expects A1.L to point at the buffer.
* ===================================================================
hex_nibble  move.b  d4,-(a7)        ; Save value in both nibbles
            andi.b  #$0f,d4         ; D4.B now = 0 to 15
            addi.b  #'0',d4         ; Now = '0' to '?' (ascii only)
            cmpi.b  #'9',d4         ; Is this a digit?
            bls.s   nib_digit       ; Yes
            addi.b  #7,d4           ; Add offset to UPPERCASE letters

nib_digit   move.b  d4,(a1)+        ; Store in buffer
            move.b  (a7)+,d4        ; Restore original value
            rts

* ===================================================================
* print_hex:
* ===================================================================
* Convert D4 into 8 hex characters, and print it to the channel in A0.L
*
* Expects D4.L to hold the value.
* Expects A0.L to hold the channel id.
* ===================================================================
print_hex   lea     buffer,a1       ; Output buffer for address
            move.w  #8,(a1)+        ; We know the result is 8 bytes
            bsr     hex_l           ; Convert 4 bytes to text
            lea     buffer,a1       ; Text to print
            bsr     prompt          ; Print it
            rts                     ; All done. (Error code in D0)

* ===================================================================
* End of test harness
* ===================================================================
```

Listing 10.11: Linked Lists - Wrapper - Part 2

10.2.5 Running The Wrapper Code.

The above code does absolutely nothing, but if you assemble it and exec the resulting file, you should see a pair of windows one with a message 'Single Linked Lists' and a prompt in the other to 'Press ENTER to quit'. Once you press the ENTER key, the job will finish. So far so good.

The reason that it does nothing is shown below:

```
* ===================================================================
* The DEMO code starts here.
* ===================================================================
Demo        rts

```

```
* ================================================================
* The DEMO code ends here.
* ================================================================
```

Listing 10.12: Linked Lists - Wrapper - Demo Placeholder

The code at Demo, does nothing but return to the caller. Our linked list code will be slotted in to replace the lines of code shown above.

To demonstrate linked lists, we need only add some code to replace the lines above. In the following two chapters we do just that, and code to demonstrate single and doubly linked lists follows there.

10.2.6 Problem Areas.

The above description, and code, is for a Single Linked List, so called because there is a single link in each node which points to the next entry in the list. This is simple to code up - as we have seen - and is fairly simple to understand, at least it is if I've done my job correctly.

The problem with a linked list created in the above fashion is that you always have to scan the list from start to some undetermined entry when you want to delete a node. And this can add serious delays to the processing of your application when a lot of nodes have to be traversed each time you need to delete one.

There is an answer, Doubly Linked Lists.

10.3 Doubly Linked Lists.

If we change the structure of our nodes and add a PRIOR pointer to each node and to the root node as well, we can store the address of both nodes neighbouring our current one, as shown in Figure 10.4 which shows the node structure.

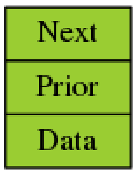


Figure 10.4: Structure of a Doubly Linked List Node.

Our conceptual model of the doubly linked list is shown in Figure 10.5.

10.3.1 Adding Nodes.

Adding a new node is still simple. Having allocated a node on the heap, you set it's PRIOR pointer to zero and it's NEXT to the current address held in the FIRST pointer - almost identical to the single linked list code above.

```
Prior       equ 4                ; Offset to PRIOR pointer in a node

Prelude     lea FIRST,a0         ; Pointer to root node
```

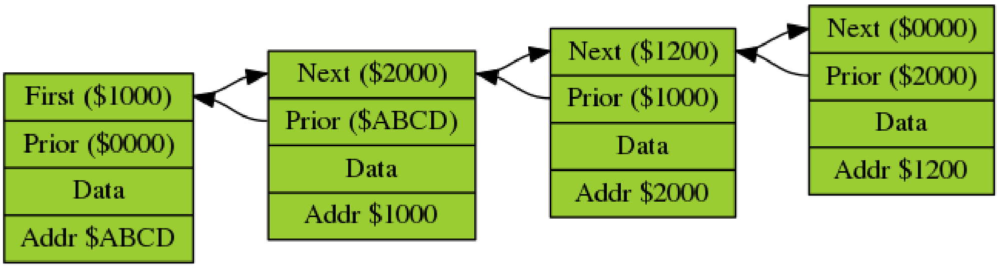


Figure 10.5: Conceptual Model of a Doubly Linked List.

```
            lea  NewEntry,a1        ; Address of new node
```

Listing 10.13: Adding a Node - Prelude

Then adding a new node to a doubly linked list is as simple as this:

```
AddNode     cmpa.l   a0,a1          ; Don't add the root node again
            beq.s    AddExit        ; Bale out quietly if attempted
            move.l  (a0),(a1)       ; Save first node in new node's NEXT
            move.l   a0,Prior(a1)   ; Set the PRIOR node for this new node
            move.l   a1,(a0)        ; Store new node in FIRST
            move.l   (a1),a0        ; A0 = address of original FIRST node
            cmpa.l   #0,a0          ; Nothing to do if A0 is zero
            beq.s    AddExit        ; Z set = First node added
            move.l   a1,prior(a0)   ; Store new FIRST node
AddExit     rts
```

Listing 10.14: Adding a Node

As with single linked lists there is nothing to it. The new node is always added at the start of the list, so the value in FIRST always points to the latest node added. The first non root node in a doubly linked list has no real PRIOR node, so that part of the newly added node is simply set to point back at the root node.

Building up the linked list above, in Figure 10.5, in stages, we would start with the root node located, most likely, somewhere in our code itself. Initially both the next and prior pointers would be set to zero.

The nodes are address at the start of the list, so the first node to be added would be the one on the far right of the diagram, node $1200. That node would be created and added to the list by setting the root node's next pointer to address $1200 and the new node's prior pointer to address $ABCD. Because it is the final node in the list so far, its next pointer is set to zero.

After adding the node at address $2000, we would change the next pointer of the root node to this new address, $2000, and the prior pointer of the node at $1200 to $2000. The new node would have its prior pointer set to the address that was in the $1200 node's prior pointer, which is $ABCD.

This process is repeated as we create each new node and add it to the list. Eventually, we end up with the structure shown in Figure 10.5 above.

You can see how each node points onward to the NEXT one and also backwards to the PRIOR one. The last node has no NEXT nodes, so it has its NEXT pointers set to zero to indicate the end of the list.

10.3.2 Deleting Nodes.

Deleting a node is much simpler. There is no need to scan the entire list from the start looking for the node prior to the one you want to delete because you already know it's address by following the PRIOR pointer backwards from the node to be deleted.

Here's the pseudo code to delete a node. We assume, as before, that A0.L is the root node pointer and A1.L is the node to be deleted.

- If the two addresses are equal, we cannot allow the root node to be deleted, exit with an error.
- If the address in the root node's NEXT pointer is zero then we still have an empty list so the value in A1 must be illegal. Exit with an error.
- Fetch the deleted node's PRIOR pointer. Every real node in a list will have a valid PRIOR pointer, only the root node has no prior pointer and we don't allow that to be deleted.
- Store the NEXT pointer from the deleted node into the NEXT pointer of the prior node.
- Fetch the deleted node's NEXT pointer, which might be zero if we are deleting the final node in the list.
- If it is not zero, store the deleted node's PRIOR pointer in the next node's PRIOR pointer.
- Exit without error.

That's the pseudo code, here's the real code to do all of the above.

```
Prior       equ 4                   ; Offset to PRIOR pointer in each node

Prelude     lea FIRST,a0            ; Pointer to root node
            lea OldNode,a1          ; Address of node to delete
            moveq #ERR_BP,d0        ; Trying to delete the root node?
```

Listing 10.15: Deleting a Node - Prelude

Now, here's the actual code to find and remove the requested node.

```
DelNode     cmpa.l a0,a1            ; Don't delete the root node
            beq.s DelExit           ; Bale out with error if attempted

            cmp.l #0,(a0)           ; Do we actually have a list?
            beq.s DelExit           ; Yes, node not found, exit with error

            move.l Prior(a1),a0     ; Fetch the deleted node's PRIOR pointer
            move.l (a1),(a0)        ; Store its NEXT pointer in the NEXT
                                    ; pointer of the PRIOR node

            move.l (a1),a0          ; Fetch the deleted node's NEXT pointer
            move.l  Prior(a1),Prior(a0) ; Store its PRIOR pointer in the
                                        ; next node's PRIOR pointer.

DelFound    move.l (a1),(a0)        ; PRIOR's NEXT = the deleted node's NEXT
            moveq  #0,d0            ; Node deleted ok
            bra.s  DelExit          ; Bale out with no errors

DelNext     move.l (a0),a0          ; A0 now holds the NEXT node in the list
            bra.s  DelNode          ; Go around again

DelExit     tst.l d0                ; Set zero flag for success
            rts
```

Listing 10.16: Deleting a Node

10.3.3 Finding Nodes.

As with single linked lists, you may have a need to locate a specific node by its contents, so you need a generic FindNode routine again. The fact that the list has two pointers this time around is the only difference, so the code is basically the same as for the single linked list above.

The only difference is the offset to the data part of the node needs to be set to 8 bytes instead of 4, so while the code for the FindNode remains the same, the code for the Compare routine needs to be changed to the following to account for the extra pointer.

As before, the comparison routine must preserve A0, A1, A2 and D0 or it will all go horribly wrong.

```
NData    equ 8                      ; Offset from start of node to the data

Compare     cmp.l  NData(a3),(a2)   ; Is the data = the value we want?
            rts                     ; Exit with Z set if so
```

Listing 10.17: Finding a Node - Data Comparison

Again, if an attempt is made to 'find' the root node, then it will fail.

10.3.4 A Better Mousetrap.

Because the code for the linked list find routine is identical except for the offset in the compare routine you can use the same code. If you modify it so that it passes the offset over to the compare routine in a spare register, say D1.W for example, then you can even have the same compare routine for both single and doubly linked lists, as shown below.

```
Compare     cmp.l  0(a3,d1.w),(a2)  ; Is the data = the value we want?
            rts                     ; Exit with Z set if so
```

Listing 10.18: Finding a Node - Data Comparison

Another method, much loved in the internals of Microsoft Windows, is to store a word holding the offset to the data at the start of each node. This would remove the need for the D1.W register to be passed into the comparison routine as a parameter as it could easily extract the data from the node itself, as follows:

```
Compare     move.w (a3),d1          ; Fetch the offset to the data
            cmp.l  0(a3,d1.w),(a2)  ; Is the data = the value we want?
            rts                     ; Exit with Z set if so
```

Listing 10.19: Finding a Node - Alternative Data Comparison

The drawback to this method is the redundancy of the data - each and every node has to have the first two bytes set to the offset to the data plus 2 for the size of the offset word itself. Two extra bytes per node may be the difference between getting all the data in memory or not. It is, of course, always up to you. If you decide to go down this route, don't forget to amend the code to add, find and delete nodes to take the extra two bytes into consideration when manipulating the pointers to NEXT and PRIOR. Your root node must also reflect these changes and have an offset word added to its own structure.

You might see a need to build a couple of comparison routines to compare two nodes rather than the example above where a node is being compared with a value. On the other hand, you could simply write one routine to compare two nodes and when looking for a value, create a dummy node and use that in the comparison routine. That way, you don't need separate routines to compare values and nodes.

10.3.5 Double Trouble.

The problem with a doubly linked list is that while adding nodes is just as simple as before, but deleting them could be problematical. If you are passed the address of a node which is not in the list, how do you tell that it is or is not a valid node address? You can end up trashing bytes of memory almost at random as you start changing the NEXT and PRIOR pointers for two areas of memory which may not be in your list.

My solution is to use a flag word or long word after the two list pointers in each node and when passed in a node address to delete, compare this value in the flag to see if it is correct before attempting to delete the node. As ever, I leave this 'as an exercise for the reader' to modify the code above to carry out said checks.

10.3.6 Sorting Lists.

The best way to sort a list is not to have to sort it at all. When you store a node in the list, store it in the correct place according to its value. A doubly linked list is used here again as you will need to go NEXT until you hit a value greater than the one you want to insert, then you might need to go PRIOR to insert it in the correct location. I'll leave you to figure out that little exercise.

There is an another way, which involves TREES of nodes rather than lists. A tree is simply a linked list which has a LEFT, RIGHT and UP pointer in each node.

With a tree, the nodes are not in a long line, but they are off to the LEFT and RIGHT of the root node. Each node may itself have children to the LEFT and RIGHT as well as a parent found by following the UP pointer.

Unfortunately, trees are a bit beyond my skills at the moment. I remember doing them in college and learning all the different ways to navigate them, but I cannot remember much about them nowadays - it's been over 30 years since I last considered them.

10.4 Remember those arrays?

Way back at the start of this chapter, I mentioned arrays and their problems. Well, combining an array with a linked list could be useful - but remember, the array is limited by the fact that you have to pre-define the number of entries.

Bearing this in mind, you could allocate an array of, say 1000 entries of 4 bytes each. Each entry in the array holds the address of an individual node, not the actual data stored there. Our address book system of 100 byte strings (not much of an address book I admit!) will now only need about 4Kb plus 102 bytes per used node - including the string length word for each entry. Using a plain array it would need almost 102Kb even for a blank address book.

Now you have compromised on memory needs as you don't allocate the space required to store your data until you need to, and you do allocate a much smaller amount to hold the 'contents table'. As you create new nodes, add their address to the array. You can still use the single or double linked lists if you wish, but there is no need. The array holds all the locations of each node in the order that they were created and you can navigate forwards, backwards and even access nodes at random using this method because the formula to find a given node is once more usable.

Have fun trying that out!

10.5 Coming Up...

Coming up next, the real code for the single linked list demo, and following that, the code for the doubly linked list demo.

11. Single Linked Lists Demo Code

11.1 Introduction

The following code demonstrates the use of single linked lists. It should be slotted into the test harness code wrapper in Chapter 10 at the appropriate place. It cannot be assembled as it stands - it needs to be part of the test harness.

11.2 How Does The Code Work?

The code is a small example of creating and navigating a linked list. The demo starts by creating a list of 5 nodes, each holding one long word of data being simply the node number 0 to 4.

The list contents are then printed on the screen shoing the node address, the next pointer and the data stored in that node. Once this is done, the node with data contents of 3 is located and deleted prior to the new list being printed out again.

Finally, each node in the list is deleted.

The first part of the code simply controls the whole demo by calling various sub-routines to do the hard work, display messages etc on screen.

```
* ====================================================================
* The DEMO code starts here.
*
* This demo does the following:
*
* Creates a number of nodes and stores a LONG value in each one.
* Lists each node address, it's NEXT pointer and data value on screen.
* Deletes a node.
* Lists each node address, it's NEXT pointer and data value on screen.
* Finds a node based on its data and displays its details on screen.
* Deletes all the nodes from the list.
```

```
* ==============================================================
Demo        bsr     BuildList           ; Build a linked list
            bsr     Before              ; Display 'BEFORE:'
            bsr     ShowList            ; Display list details
            bsr     FindNode            ; Locate a single node
            bne.s   DemoAfter           ; Failed to find node data = 3
            bsr     DeleteNode          ; Delete a single node

DemoAfter   bsr     After               ; Display 'AFTER: '
            bsr     ShowList            ; Show details again
            bsr     KillList            ; Delete entire list
            rts                         ; Done
```

Listing 11.1: Single Linked List - Demo Code

Following on from the main control section of the demo, we have our much beloved root node which must be initialised to zero as outlined in the original article. This is the pointer we will be loading into A0 quite often in the demo and it holds the address of the first node in the list. At present, there is no list, so the contents are set to zero to indicate the very end of the list.

```
* --------------------------------------------------------------
* A location to hold a single long word pointing to the first 'real'
* node in our linked list. This must be initialized to zero.
* --------------------------------------------------------------
RootNode    dc.l    0                   ; This is our root node.
```

Listing 11.2: Single Linked List - Demo Code - Root Node

The first of our sub-routines follows on. This part builds a list of 5 nodes in the most simple manner possible - it runs a loop which calls the sub-routine to create a single node and link it into the list. If you want a bigger list, change the counter loaded into D7 to one less than the number of node you want. Don't forget to adjust the height of your window as wll if you want to see all the results on screen at the same time.

```
* --------------------------------------------------------------
* Build a list of 5 nodes each holding a long word of data.
* --------------------------------------------------------------
BuildList   lea     RootNode,a0         ; Pointer to root node address
            moveq   #4,d7               ; How many nodes in D7 = 5
            moveq   #8,d1               ; Each node is 8 bytes long

BuildLoop   bsr.s   BuildNode           ; Create a node, address in A1
            bne     all_done            ; Just die on errors
            move.l  d7,4(a1)            ; Store data value
            bsr.s   AddNode             ; Add to list
            dbra    d7,BuildLoop        ; Do the rest
            rts                         ; Done
```

Listing 11.3: Single Linked List - Demo Code - Build List

Here's the first of the real list routines. We add a new node into the list in the manner outlined in the article. We reject attempts to add the rot node into the list - but without flagging any errors - and, as explained, we don't attempt to check if the new node already exists in the list because we are creating the node on the heap, so the chances of the new node being present already are pretty slim to say the least.

```
* --------------------------------------------------------------
```

```
* AddNode - Adds a new node to a list. See text for details.
* Preserves all regsiters.
* No errors returned.
* ------------------------------------------------------------------------
AddNode     cmpa.l   a0,a1               ; Don't add the root node again
            beq.s    AddExit             ; Bale out quietly if attempted
            move.l   (a0),(a1)           ; Save first node in new NEXT
            move.l   a1,(a0)             ; Store new node in root node
AddExit     rts
```

Listing 11.4: Single Linked List - Demo Code - Add Node

A new node is built by allocating some space on the common heap but we must preserve the working registers, the following code does this for us.

```
* ------------------------------------------------------------------------
* Allocate a single new node
* On entry, D1.L is amount of memory required.
* On exit, A1 holds the address of the new node, with D0 holding errors.
* All registers preserved - unless otherwise stated.
* ------------------------------------------------------------------------
BuildNode   movem.l  d1-d3/a0/a2-a3,-(a7) ; Save working registers
            moveq    #MT_ALCHP,d0        ; Set the trap
            moveq    #me,d2              ; I want it for me
            trap     #1                  ; Do it
            move.l   a0,a1               ; Node address into A1
            movem.l  (a7)+,d1-d3/a0/a2-a3 ; Restore working registers
            tst.l    d0                  ; Set flags
            rts                          ; Exit
```

Listing 11.5: Single Linked List - Demo Code - Build Node

The following sub-routine is called once to display the linked list before we do the deletions and again after we have deleting a node. The code simply walks through the entire list and prints out the node address, the next pointer and the data value by calling other sub-routines.

```
* ------------------------------------------------------------------------
* Walk through a linked list displaying the details of each node as we
* go.
* On entry, A0 = root node of the list.
* ------------------------------------------------------------------------
ShowList    lea      RootNode,a0         ; Root node address

ShowLoop    move.l   (a0),a0             ; Get address of the next node
            cmpa.l   #0,a0               ; Done?
            beq.s    ShowExit            ; Yes
            move.l   a0,-(a7)            ; Preserve A0 - it's our node
            bsr.s    ShowNode            ; Display that node's details
            move.l   (a7)+,a0            ; Restore the node pointer
            bra.s    ShowLoop            ; Do the rest of the list

ShowExit    rts                          ; Done
```

Listing 11.6: Single Linked List - Demo Code - Show List

This next short routine is called with the address of a node in A0.L and prints the details of that node to the screen.

```
*------------------------------------------------------------------
* Display the details of a single node in the linked list.
* On entry A0 = the node address.
*------------------------------------------------------------------
ShowNode    move.l  a0,a5               ; The node address
            move.l  con_id2(a4),a0      ; The channel address
            move.l  a5,d4               ; The node address
            bsr.s   ShowAddr            ; Print node address
            move.l  (a5),d4             ; The NEXT pointer
            bsr.s   ShowNext            ; Print NEXT pointer
            move.l  4(a5),d4            ; The node data
            bsr     ShowData            ; Print the data
            rts
```

Listing 11.7: Single Linked List - Demo Code - Show Node

Obviously, just displaying the list contents isn't very user firiendly, so the next couple of routines display a title which informs the user if the list being displyed is 'before' or 'after' the deletion of a node.

```
*------------------------------------------------------------------
* Display 'BEFORE:' in the output channel.
*------------------------------------------------------------------
Before      lea     BeforeAddr,a1       ; The prompt
            movea.l con_id2(a4),a0      ; Needs channel id
            bsr     Prompt              ; Print it
            rts

BeforeAddr  dc.w    B4End-BeforeAddr-2
            dc.b    'BEFORE:',linefeed
B4End       equ     *

*------------------------------------------------------------------
* Display 'AFTER:' in the output channel
*------------------------------------------------------------------
After       lea     AfterAddr,a1        ; The prompt
            movea.l con_id2(a4),a0      ; Needs channel id
            bsr     Prompt              ; Print it
            rts

AfterAddr   dc.w    AftEnd-AfterAddr-2
            dc.b    linefeed,linefeed,'AFTER:',linefeed
AftEnd      equ     *
```

Listing 11.8: Single Linked List - Demo Code - Show Before and After States

There now follows one of three separate but short routines to display on screen, the various parts of a list node. This one simply displays the node's address in memory. Following after this routine is a number of small sub-routines which assist in the displaying of node data by converting the contents of D4 to hex and printing it to the screen.

```
*------------------------------------------------------------------
* Display the node's actual address in memory.
* On entry D4 = the node address.
*------------------------------------------------------------------
ShowAddr    lea     MsgAddr,a1          ; Our prompt

```

```
ShowPrompt bsr      Prompt              ; Print it
           bsr.s    D4ToHex             ; Convert D4.L to hex
           bsr.s    PrintHex            ; Print it and a linefeed
           rts

MsgAddr    dc.w     AddrEnd-MsgAddr-2
           dc.b     linefeed,'Node address: '
AddrEnd    equ      *

* ------------------------------------------------------------------
* Print the contents of the buffer to screen.
* ------------------------------------------------------------------
PrintHex   lea      Buffer,a1           ; What to print
           move.l   con_id2(a4),a0      ; Channel to print to
           bsr      Prompt              ; Do it
           rts

* ------------------------------------------------------------------
* Convert the long word in D4 to hex ready for printing
* ------------------------------------------------------------------
D4ToHex    lea      buffer+2,a1         ; Buffer address
           bsr      hex_l               ; Do all 4 bytes = 8 characters
           lea      buffer,a1           ; Buffer again
           move.w   #8,(a1)             ; Store text length
           rts
```

Listing 11.9: Single Linked List - Demo Code - Show Addresses

The second and third routines to diplsy the details of a node now follow. Starting with the code to show the node's NEXT pointer address closely followed by the code to print the actual data stored in the node.

```
* ------------------------------------------------------------------
* Display the node's NEXT address in memory.
* On entry D4 = the node's NEXT pointer.
* ------------------------------------------------------------------
ShowNext   lea      MsgNext,a1          ; Our prompt
           bra.s    ShowPrompt          ; Print it

MsgNext    dc.w     NextEnd-MsgNext-2
           dc.b     '  NEXT pointer: '
NextEnd    equ      *

* ------------------------------------------------------------------
* Display the node's actual data content.
* On entry D4 = the data.
* ------------------------------------------------------------------
ShowData   lea      MsgData,a1          ; Our prompt
           bra.s    ShowPrompt          ; Print it

MsgData    dc.w     DataEnd-MsgData-2
           dc.b     '  Data value: '
DataEnd    equ      *
```

Listing 11.10: Single Linked List - Demo Code - Show Next Address

Next we have the code to locate a single node in the linked list based upon the data part of the node.

This part is simply the setup routine for the following code at FindANode which does the actual scanning of the node and calling the compare routine as described in the previous chapter.

```
*---------------------------------------------------------------------
* Locate a node in the list based on it's data value.
* On exit, A1 is the required node's address plus Z set - if found.
*          A1 is undefined plus Z clear - if not found.
*---------------------------------------------------------------------
FindNode    lea      RootNode,a0          ; Pointer to root node in list
            lea      Compare,a1           ; Node comparison routine
            moveq    #3,d1                ; We are looking for this data
            bsr.s    FindANode            ; Go find it, or not
            rts

*---------------------------------------------------------------------
* This routine expects the following input registers to be able to scan
* a linked list for the required data and return the address of the
* node holding that data with the Z flag set if found, or the Z flag
* cleared for not found.
*
* A0.L = Rootnode of the list.
* A1.L = Address of Compare routine.
* D1.L = Value to look for in list.
*---------------------------------------------------------------------
FindANode   moveq    #oops,d0             ; Assume not found (yet)

FindLoop    cmpa.l   #0,a0                ; Reached the end yet?
            beq.s    FindExit             ; Yes, node not found, exit

            move.l   (a0),a3              ; Get NEXT node into A3.L
            jsr      (a1)                 ; Call the comparison routine
            beq.s    FindFound            ; Looks like we found our node

FindNext    move.l   (a0),a0              ; A0 = NEXT node in the list
            bra.s    FindLoop             ; Go around again

FindFound   movea.l  a3,a1                ; This is the required node
            moveq    #0,d0                ; Clear error flag

FindExit    tst.l    d0                   ; Set zero flag for success
            rts

*---------------------------------------------------------------------
* This is the simple compare routine for our FindNode code. On entry,
* we have the following registers set:
*
* D1.L = The value we want to find in a node in the list.
* A3.L = The address of the node we are searching.
*
* We must preserve A0, A1 and D1.
*---------------------------------------------------------------------
NData       equ      4                    ; Offset to the data
Compare     cmp.l    NData(a3),d1         ; Found the data yet?
            rts                           ; Exit with Z set if so
```

Listing 11.11: Single Linked List - Demo Code - Find Node

This next routine is not really required on QDOSMSQ as a terminating job always has any allocated heap areas returned to the system by the job scheduler routines. Because I'm a lazy typist and in order that I reduce the large amounts of code in the magazine, I'm not writing any code here!

If you wish to carry out the list tidying explicitly for yourself as an exercise, feel free to do so. As a suggestion, start a loop which keep going around the list fetching the NEXT node pointer and deleting that from the list using the routines in this code. Once the node has been unlinked from the list, you may deallocate it's heap area - but don't forget to preserve those registers.

```
*------------------------------------------------------------------------
* QDOSMSQ tidies up rather nicely for us on exit - so we don't have to!
*------------------------------------------------------------------------
KillList    rts
```

Listing 11.12: Single Linked List - Demo Code - Kill List

The folowing code sets up the demo to delete the node that was just 'found' by searching for the node holding data 3. This code is called with the address of the '3' node in A1.L and it simply sets up the following routine which actually scans the list looking to make sure that the node we are deleting exists in the list.

```
*------------------------------------------------------------------------
* A demo routine to delete the node whose address is passed in A1.L.
* Sets Z if found & deleted, clears it otherwise.
*------------------------------------------------------------------------
DeleteNode  lea      rootnode,a0          ; Address of the root node
            bsr.s    DelANode
            rts
```

Listing 11.13: Single Linked List - Demo Code - Delete Node

This is the node deletion code itself. As described in the article, we must not delete the root node itself - as this isn't allocated on the heap. We must also check that the node is in the list by scanning from start to finish looking for the node in the list which has a NEXT pointer holding the address of the node we want to delete.

We remove a node from the list by copying the soon to be deleted node's NEXT pointer into the NEXT pointer of the node before it, thus bypassing the node we want to delete.

Warn

This code only deletes a node from the linked list. It does not deallocate the memory on the common heap that was allocated to create the node. QDOSMSQ will do this at the end of the demo, but in real life, you would need to carry out this task yourself - especially as you may not want a number of deleted heap areas hanging around in memory fragmenting your heap.

```
*------------------------------------------------------------------------
* Routine to delete a node with the address passed in A1.L from the
* list whose address is passed in A0.L. On exit, Z flag will be set if
* deleted or cleared if not.
*------------------------------------------------------------------------
DelANode    moveq    #oops,d0             ; Assume it's going to fail
            cmpa.l   a1,a0                ; Deleting the root node?
            beq.s    DelExit              ; Exit if so.

DelLoop     cmpi.l   #0,(a0)              ; Finished yet?
            beq.s    DelExit              ; Exit not found
```

```
                cmpa.l  (a0),a1             ; Found the previous node
*                                           ; to the one we want to delete?
                bne.s   DelNext             ; Not yet, try again

DelFound        move.l  (a1),(a0)           ; Delete the node - set NEXT
*                                           ; of the node BEFORE the one to
*                                           ; be deleted to NEXT of the
*                                           ; node that is being deleted.
                moveq   #0,d0               ; Indicate found and deleted
                bra.s   DelExit             ; Set Z flag on the way out

DelNext         move.l  (a0),a0             ; Get the next node in the list
                bra.s   DelLoop             ; And try again

DelExit         tst.l   d0                  ; Set or clear Z flag
                rts

* =====================================================================
* The DEMO code ends here.
* =====================================================================
```

Listing 11.14: Single Linked List - Demo Code - Deleting A Node

And that is all there is to it. The SingleList demo code should be assembled and run in the normal fashion. You'll be able to see that there are indeed 5 nodes in the list (in the BEFORE section at the top of the screen) then under that, the AFTER section shows a missing node with data content 3 - we have deleted it from the list.

11.3 Coming Up...

In the next chapter the real working demo code for doubly linked lists will be shown and explained.

12. Doubly Linked Lists Demo Code

12.1 Introduction

The following code demonstrates the use of doubly linked lists. It should be slotted into the test harness code wrapper from Chapter 10 at the appropriate place. It cannot be assembled as it stands - it needs to be part of the test harness.

12.2 How Does The Code Work?

Much of the demo code is identical to last time, so I'll save some space and paper by only showing you routines that are changed or new ones that were added.

As with the SingleList demo, the code is a small example of creating and navigating a linked list. The demo starts by creating a list of 5 nodes, each holding one long word of data being simply the node number 0 to 4. Each node is linked to the one after it and to the one before it.

The list contents are then printed on the screen showing the node address, the prior pointer, the next pointer and the data stored in that node. Once this is done, the node with data contents of 3 is located and deleted prior to the new list being printed out again. Sounds very familiar doesn't it?

I've had to trim the informational part of the screen output for each node to accommodate the extra address in the PRIOR pointer and to make sure that it all fits on one screen line.

As with the demo code for singly linked lists, I'm not physically deleting the allocated heap areas used for each node. This reduces the amount of code that appears in the magazine and reduces the need to chop down a few more trees. However, bear in mind that if you create programs which don't delete the heap areas when a node is deleted, that your memory usage will remain high throughout the run of the program.

In my case, this is a small demo and QDOSMSQ does the tidying up for me at the end of the demo.

Note

In the following descriptions of changes to the existing demo code for single linked lists, all of the line numbers shown or mentioned, refer to the original line numbers in the demo code for single linked lists from the previous chapter.

Where code is being added, the first line number shown is where I have inserted it into my version of the demo code. Your mileage may vary - as *they* say!

The first part of the code which has changed is the definition of the root node at line 24. In the single linked list, this was a single long word initialised to zero. In this demo we have a pair of long words initialised to zero. To make life easier, we also define a number of equates for use throughout the remainder of the code.

The root node must now be initialised to zero in both its NEXT and PRIOR areas as outlined in the original article. This is the pointer we will be loading into A0 quite often in the demo and it holds the address of the first node in the list. At present, there is no list, so the contents are set to zero to indicate the very end of the list. The PRIOR pointer will always be zero because there is never a previous node to the root.

Note

So, effectively, I could have simply left the root node identical to that of the single linked list demo, if there's no need to hold a PRIOR address, we don't need a PRIOR pointer storage area.

However, read any book on linked lists in almost any language, and you will note that the root node is simply a normal node, without any use of it's PRIOR pointer. Some books do use the PRIOR pointer, to point at the final node in the list, but that can lead to runaway code if there's no way to detect the end node. Especially when the last node's NEXT pointer points to the root node!

```
24 *------------------------------------------------------------------------
25 * A location to hold a single long word pointing to the first 'real'
26 * node in our linked list. This must be initialized to zero.
27 *------------------------------------------------------------------------
28 RootNode    dc.l     0,0              ; Root node with 2 pointers.
29
30 NodeSize    equ      12               ; Node size in bytes
31 Next        equ      0                ; Offset to NEXT pointer
32 Prior       equ      4                ; Offset to PRIOR pointer
33 NData       equ      8                ; Offset to the node's data
```

Listing 12.1: Doubly Linked List - Demo Code - Root Node

The code in BuildList (line 29 onwards) has been changed slightly too. In the single list version, the offsets were hard coded as numbers. This isn't very clever - if you change the offsets at any future point, you have to find all the places where the numbers are hard coded. In the new version, I use equates instead of hard coded values. This way, if I change my node structure, I only have to change the equates once.

```
29 *------------------------------------------------------------------------
30 * Build a list of 5 nodes each holding a long word of data.
31 *------------------------------------------------------------------------
32 BuildList   lea      RootNode,a0      ; Root node address
33             moveq    #4,d7            ; 5 nodes to create
34             moveq    #NodeSize,d1     ; Each node is 12 bytes long
35
36 BuildLoop   bsr.s    BuildNode        ; Create node, address in A1.L
37             bne      all_done         ; Just die on errors
```

```
                move.l   d7,NData(a1)      ; Store data value
                bsr.s    AddNode           ; Add to list
                dbra     d7,BuildLoop      ; Do the rest
                rts                        ; Done
```

Listing 12.2: Doubly Linked List - Demo Code - Build List

AddNode has been changed to cater for the doubly linked list by initialising the NEXT and PRIOR pointers in the new node and in the root node, then checking if there was already any nodes in the list. If there were any nodes, the previous 'first' node in the list (ie, the most recent one added) needs to have its PRIOR pointer set to be our brand new node. This is done by loading A0 with the new node's NEXT pointer and testing that for zero, if the new node has no NEXT address, it can only be the very first node in the list.

We initialise as follows:

- NEXT(Root) copied to NEXT(NewNode)
- Root address copied to PRIOR(NewNode)
- NewNode address copied to NEXT(Root)

If this is not the very first node in the list then:

- Get address of NEXT(NewNode)
- NewNode address copied to PRIOR(NextNode).

```
*------------------------------------------------------------------------------
* AddNode - Adds a new node to a list. See text for details.
*
* Entry: A0.L = root node address, A1.L = New node address.
* Exit :Preserves all registers, no error codes returned.
*------------------------------------------------------------------------------
AddNode         cmpa.l   a0,a1             ; Don't add the root node again
                beq.s    AddExit           ; Bale out quietly if attempted
                move.l   (a0),(a1)         ; Save first node in node's NEXT
                move.l   a0,Prior(a1)      ; Set the new node's PRIOR
                move.l   a1,(a0)           ; Store address of node in root
                cmpa.l   #0,(a1)           ; First ever node?
                beq.s    AddExit           ; Yes. Exit with Z set
                move.l   a0,-(a7)          ; Preserve root node pointer
                move.l   (a1),a0           ; A0 = addr of previous first node
                move.l   a1,prior(a0)      ; Set PRIOR to our new node
                move.l   (a7)+,a0          ; Restore root node pointer
AddExit         rts
```

Listing 12.3: Doubly Linked List - Demo Code - Add Node

The ShowNode code is the next part that has changed. It has had a couple of lines added to call a new routine - ShowPrior - which, as its name suggests, displays the address of the PRIOR pointer for the node being displayed on screen.

Warn

The line BSR.S ShowNext must also be changed to remove the '.S' from the BSR instruction. We've slipped out of range of a short jump now, so you'll get an error message 'Number Too Big' if you don't.

```
*------------------------------------------------------------------------------
* Display the details of a single node in the linked list.
```

```
84 * On entry A0 = the node address.
85 *-----------------------------------------------------------------------
86 ShowNode    move.l  a0,a5            ; The node address
87             move.l  con_id2(a4),a0   ; The channel address
88             move.l  a5,d4            ; The node address
89             bsr.s   ShowAddr         ; Print node address
90             move.l  (a5),d4          ; The NEXT pointer
91             bsr     ShowNext         ; Print NEXT pointer
92             move.l  Prior(a5),d4     ; The PRIOR pointer
93             bsr.s   ShowPrior        ; Print PRIOR pointer
94             move.l  NData(a5),d4     ; The node data
95             bsr     ShowData         ; Print the data
96             rts
```

Listing 12.4: Doubly Linked List - Demo Code - Show Node

In addition, I've abbreviated the message printed by ShowAddr to the following:

```
129 MsgAddr     dc.w    AddrEnd-MsgAddr-2
130             dc.b    linefeed,'Node addr: '
131 AddrEnd     equ     *
```

Listing 12.5: Changes to MsgAddr Text Data

The following short routine should be added just above ShowNext (currently at line 149 onwards) in the original code. It is called by ShowNode to display the address in a node's PRIOR pointer.

```
149 *-----------------------------------------------------------------------
150 * Display the node's PRIOR address in memory.
151 * On entry D4 = the node's PRIOR pointer.
152 *-----------------------------------------------------------------------
153 ShowPrior   lea     MsgPrior,a1      ; Our prompt
154             bra.s   ShowPrompt       ; Print it
155 
156 MsgPrior    dc.w    PriorEnd-MsgPrior-2
157             dc.b    '   PRIOR: '
158 PriorEnd    equ     *
```

Listing 12.6: Doubly Linked List - Demo Code - Show Prior Address

The code in the ShowNext and ShowData routines hasn't changed, but the messages they display have. I needed to shorten the text to get everything on screen in one line per node. Please make the following changes at original lines 156 and 167:

```
156 MsgNext     dc.w    NextEnd-MsgNext-2
157             dc.b    '   NEXT: '
158 NextEnd     equ     *
```

Listing 12.7: Changes to MsgNext Text Data

```
167 MsgData     dc.w    DataEnd-MsgData-2
168             dc.b    '   Data: '
169 DataEnd     equ     *
```

Listing 12.8: Changes to MsgData Text Data

The code in DelANode has been reduced quite dramatically to the following. As before we don't allow deletion of the root node itself and exit quietly if any attempt is made to do so.

Next we check to ensure that we actually have a list to delete from. If the root node's NEXT pointer is still zero, we have no nodes in the list and again, we exit quietly. In both these exit situations, we clear the Z flag to indicate a node not deleted error.

Deleting the node from the list (but, as before, not from memory) is actually quite simple. As A1 points to the node to be deleted we can find the node before it from the PRIOR(a1) address, and the node after it by the NEXT(A1) address. All we do to delete the node from the list is set the prior node's NEXT address to the current value in the deleted node's NEXT address and then set the next node's PRIOR address to the PRIOR address of the deleted node.

However, if we are deleting the very last node in the list, we must not attempt to change the (non-existent) next node's pointers as we may well end up writing to random locations in memory. In the last node, the NEXT pointer is always zero.

```
*------------------------------------------------------------------
* Routine to delete a node with the address passed in A1.L from the
* list with the address passed in A0.L. On exit, Z flag will be set if
* the node was deleted, or cleared if not.
* Trashes A0 and D0 on exit.
*------------------------------------------------------------------
DelANode    moveq    #oops,d0          ; Assume it's going to fail
            cmpa.l   a1,a0             ; Trying to delete the rootnode?
            beq.s    DelExit           ; Exit if so.

DelList     cmpi.l   #0,(a0)           ; Empty list?
            beq.s    DelExit           ; Yes. Exit not found
            move.l   Prior(a1),a0      ; A0 = node before the 'deleted' one
            move.l   (a1),(a0)         ; Prior's NEXT = deleted node's NEXT
*                                      ; thus deleting the node from the
*                                      ; NEXT chain through the list.
            cmpi.l   #0,(a1)           ; Deleting final node in list?
            beq.s    DelDone           ; Yes, nothing more to do
            move.l   (a1),a0           ; A0 = deleted node's NEXT
            move.l   Prior(a1),Prior(a0) ; Next's PRIOR = deleted node's
*                                      ; PRIOR thus deleting the node
*                                      ; from the PRIOR chain.
DelDone     moveq    #0,d0             ; Indicate node deleted successfully

DelExit     tst.l    d0                ; Set or clear Z flag
            rts

* ==================================================================
* The DEMO code ends here.
* ==================================================================
```

Listing 12.9: Doubly Linked List - Demo Code - Deleting A Node

And that is all the changes you have to make. The DoubleList demo code should be assembled and run in the normal fashion. You'll be able to see that there are indeed 5 nodes in the list (in the BEFORE section at the top of the screen) then under that, the AFTER section shows a missing node with data content 3 - we have deleted it from the list.

12.3 Coming Up...

In the next chapter we'll stray a little into some territory that I have never seen demonstrated in assembly language programming for the QL, I'm talking of recursive routines. Until then, keep your stack untangled!

13. Recursion

13.1 Introduction

After the recent musings on single and double linked lists, this time I'm going to delve into the murky depths of a subject I've never seen before discussed for QDOSMSQ assembly language. The subject is recursion.

Recursion is a very simple concept, but for some people, it can be quite difficult to get their head around it, and it never comes clear. I suspect for those people, trying to do it in assembly language is equally difficult. Lets hope I can explain it in simple enough terms for even me to understand!

13.2 Recursion in Assembly Language

A recursive routine is simply a routine which calls itself until a certain exit condition is detected. The exit condition is very important, if you miss it out, your programs will loop around until such time as the stack fills up and the program crashes, or eats itself.

Here's a very simple example of a program that will recurse 'forever' until it dies.

```
Start      bsr.s       Recurse
           rts

Recurse    bsr.s       Recurse
           rts
```

Listing 13.1: Very Faulty Recursive Program

None of the `RTS` instructions will ever get executed. because all the program does is calls Recurse over and over again, but each call is nested inside the previous call, so the (A7) stack pointer keeps going down by 4 each time it is called as the `BSR` instruction stacks the return address and then branches off to the next iteration of Recurse.

Recursion educators are very fond of certain examples when teaching recursion. The towers of Hanoi, Factorials, Exponentiation, Fibonacci numbers etc. I'm no different, so here's a few small explanations and examples.

13.2.1 Factorials

The factorial of a number is that number, multiplied by the factorial of the number before it. The symbol for a factorial number is the exclamation mark (!) So, $4! = 4 * 3!$

There is no concept of a negative factorial, so $-3!$, doesn't exist. The smallest factorial number is $1!$, which has the value of 1.

Factorial numbers imply recursion and we have the following simple code.

However, just before we delve into the code be aware that factorials get very big very quickly, 12! ($1C8CFC00) is the largest that can fit into a single 32 bit (unsigned) register and 13! ($17328CC00) cannot fit. The largest factorial we can fit into a 16 bit unsigned word is only 8! ($9D80) and as the 68008 processor can only multiply 16 bit numbers, this means that 9! will be the biggest that the following routine can calculate without overflowing.

Note Other processors in the MC68000 family can multiply 32 bit numbers.

On entry to the code, D0.W is the number to calculate the factorial of and on exit, D0.L is the result. D0.W must in range 1 to 9 only but this routine does not perform error trapping - on your own head be it!

```
Start       bra.s     Start2        ; Skip over result area
Result      dc.l      0             ; Result area
FirstNumber dc.w      9             ; Assume 9! by default

Start2      lea       Result,a3     ; Where to shove the answer
            moveq     #0,d0         ; Clear all 32 bits of D0.L
            move.w    4(a3),d0      ; Fetch FirstNumber
            bsr.s     Factorial     ; Do it
FRET_1      move.l    d0,(a3)       ; Shove it!
            moveq     #0,d0         ; No errors
            rts                     ; Back to SuperBasic

Factorial   move.w    d0,-(a7)      ; Stack current number
            subq.w    #1,d0         ; Calculate previous number
            bne.s     Not_zero      ; Not done unless next number is 0
FPOP        move.w    (a7)+,d0      ; Retrieve the value 1 from stack
            rts                     ; Back to FRET_1 if FirstNumber
*                                   ; was 1 else FRET_2

Not_zero    bsr.s     Factorial     ; Lets go round again, and again
FRET_2      mulu      (a7)+,d0      ; Do the multiplication
            rts                     ; Exit
```

Listing 13.2: Recursive Factorial Program

For the simplest case, assume we start with a value of $0001. The stack will look like this at label 'Start':

```
Return address to SuperBasic.
```

As we drop through the code beginning at ‘Start2’ and execute our first branch to subroutine ‘Factorial’ the stack now looks like this :

```
Return address to SuperBasic.
Return address to FRET_1
```

Tracing through the ‘Factorial’ code, we stack the current value of D0.W (which is 1) so the stack now looks like this:

```
Return address to SuperBasic.
Return address to FRET_1
$0001
```

After the subtraction, D0 has become zero, so we exit out of the ‘Factorial’ code from label FPOP by popping the value $0001 off the stack into D0.W leaving the stack like this:

```
Return address to SuperBasic.
Return address to FRET_1
```

Then we execute the `RTS` instruction to return us to the label FRET_1 where we store the result of $00000001 from D0.L into the result area set aside for this very purpose.

So far so good, we haven’t actually done any recursion yet, but read on. If we start with the value of $0002 in ‘FirstNumber’ then the process is slightly different. We start, as ever with the SuperBasic return address on the stack when we are at label ‘Start’.

Dropping into ‘Start2’ and executing our first `BSR` to ‘Factorial’, the stack is as above at the same point. Nothing much has changed. However, we then stack the current value in D0.W to give a stack as follows:

```
Return address to SuperBasic.
Return address to FRET_1
$0002
```

This is slightly different. When we calculate the next number, we do not set D0.W to zero, so we skip out of the ‘Factorial’ code block to the code at ‘Not_zero’ which immediately causes another `BSR` to ‘Factorial’ leaving the stack as follows:

```
Return address to SuperBasic.
Return address to FRET_1
$0002
Return address to FRET_2
```

Once again, we stack D0.W and subtract one to find that we have now reached zero. The stack looks like this:

```
Return address to SuperBasic.
Return address to FRET_1
$0002
Return address to FRET_2
$0001
```

Once more, we pop the value $0001 off the stack back into D0.W and then execute the `RTS` instruction. This time, however, we do not return to FRET_1 but to FRET_2 where we end up with the following stack arrangement:

```
Return address to SuperBasic.
Return address to FRET_1
$0002
```

The instruction at ‘FRET_2’ causes the top word on the stack to be multiplued by D0.W the result store in D0.L. This leaves D0.L equal to $00000002 which just happens to be the correct value for 2! and we exit the code by returning to ‘FRET_1’ where we store the result again.

The process is similar for all the other numbers, so 5! will have a stack looking like this when we reach, but just before we execute the code at ‘FPOP’:

```
Return address to SuperBasic.
Return address to FRET_1
$0005
Return address to FRET_2
$0004
Return address to FRET_2
$0003
Return address to FRET_2
$0002
Return address to FRET_2
$0001
```

The stack will begin to unwind as we do the sequence of `MULU` and `RTS` instructions at FRET_2 as we first calculate 1!, then 2!, then 3!, then 4! and finally 5! which is the result we return to SuperBasic.

To run the above code, do this, or something like it:

```
1000 Start = ALCHP(128)
1010 LBYTES win1_source_factorial_bin , Start
1020:
1030 DEFine PROCedure Fact(n)
1040   IF n < 1 OR n > 9 THEN
1050      PRINT n; ' is slightly out of range , 1 to 9 only please.'
1060   END IF
1070   POKE_W Start + 6, n
1080   CALL Start
1090   PRINT n; '! = '; PEEK_L(Start + 2)
1100 END DEFine Fact
```

Now, at the SuperBasic prompt, run the above to load the code, you only need to do this once, then just type Fact(n) where 'n' is a value between 1 and 9 as described above in the text. The results will be 'interesting' if you use values outside of this range.

Actually, in the interests of experiment, I tried it out. Using values above 9 is fine, up to a point, but zero will trash SuperBasic as the stack wanders down through memory and corrupts data that it shouldn't be anywhere near. Larger values will no doubt have the same effect, but anything over 9 gives incorrect results as the 16 bit `MULU` instruction isn't using the additional bits of the number. Anyone got a good 32 bit `MULU` and/or `MULS` routine they want to share?

I don't have the numeric skills to write one, and while there are plenty on the Web, they are, of course, someone else's work and subject to copyright etc.

13.2.2 The Fibonacci Series

The Fibonacci series looks like this:

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ...
```

Apart from the first two numbers, each number in the series is the sum of the previous two numbers. This is written as

```
Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
```

The very explanation cries out for recursion, you can see it in the statement above. We need to cater for the first two terms in the series and test for a Fibonacci(0) or Fibonacci(1) and return the value 1 for both of these values. The slight difference between Fibonacci and Factorial is that we need to recurse twice for each number, once for $(N-1)$ and once for $(N-2)$. This makes the code slightly more interesting and the stack too.

Here's how it looks in plain and simple SuperBasic:

```
1000 DEFine FuNction Fibonacci(n)
1010    IF n = 0 OR n = 1 THEN RETurn 1
1020    RETurn Fibonacci(n-1) + Fibonacci(n-2)
1030 END DEFine
```

So how difficult could it be to convert the above (two) lines of working code into assembly language? It all depends on how easily you get your head around the recursion, I had to sit and stare at the screen for a while until I finally came up with the following code:

```
Start       bra.s    Start2          ; Skip over result storage
Result      dc.l     0               ; Space for result at Start + 2
FirstNumber dc.l     9               ; Fib(9) by default = Start + 6

Start2      lea      Result,a3       ; Where to shove the answer
            move.l   4(a3),d0        ; Fetch FirstNumber
            bsr.s    Fibonacci       ; Do it
FRET_1      move.l   d0,(a3)         ; Shove it!
            moveq    #0,d0           ; No errors
            rts                      ; Back to SuperBasic

```

```
Fibonacci   cmpi.l   #2,d0        ; Special cases 0 or 1?
            bcc.s    Fib_2        ; No, D0 is 2 or more. (Unsigned!)
            moveq    #1,d0        ; Return 1 for Fib(0) or Fib(1)
            rts                   ; That's our exit clause!

Fib_2       subq.l   #1,d0        ; Calculate N-1
            move.l   d0,-(a7)     ; Stack our 'N-1' value
            bsr.s    Fibonacci    ; Work out Fib(N-1)
FRET_2      move.l   d0,-(a7)     ; Save the result of Fib(N-1)
            move.l   4(a7),d0     ; Retrieve N-1
            subq.l   #1,d0        ; Calculate N-2
            bsr.s    Fibonacci    ; Work out Fib(N-2)
FRET_3      add.l    (a7)+,D0     ; Add Fib(N-1) to Fib(N-2)
            adda.l   #4,a7        ; Tidy original N-1 off of stack
            rts                   ; And return
```

Listing 13.3: Recursive Fibonacci Program

To run the above code, do this, or something like it:

```
1000 Start = ALCHP(128)
1010 LBYTES win1_source_fibonacci_bin, Start
1020:
1030 DEFine PROCedure Fib(n)
1040   IF n < 0 THEN
1050      PRINT n; ' is slightly out of range, 0 and over only please.'
1060   END IF
1070   POKE_L Start + 6, n
1080   CALL Start
1090   PRINT n; '! = '; PEEK_L(Start + 2)
1100 END DEFine Fib
```

This time we can use numbers larger than 9 as we are adding 32 bit values in the code, not multiplying. Of course, you can still pick a number big enough to trash the stack. Interestingly enough, Fib (30) executes in 1 second on my QPC setup, but the original SuperBasic version ran and ran and ran I CTRL-SPACE'd it after a while.

As an exercise, try to work out what the stack looks like for different values of N - it's an interesting lesson in mind numbing loops. Once you figure it out though, it gets easier.

When you are writing recursive code like the two examples above, you must remember two golden rules:

- you must always have a 'get out' clause to stop recursion
- never ever try to use other registers as storage - it just doesn't work!

In the above, we just stacked our working values and this is fine, but in other code, you might need to have a lot more values to stack, so how best to do this? The answer is quite simple, use the `LINK` and `UNLK` instructions which are designed to build stack frames that you can access using Address Register Indirect With Displacement - for example 4(a5) - addressing mode instructions.

Out of interest, has anyone spotted a potential problem with the above code?

The calculation of Fib(N-2) duplicates most of the work done by Fib(N-1). One solution to this problem is to have an array of values in memory and when calculating a new value, store it in the table if it has not been stored already, if it has been stored already, simply extract it from the table.

The last two paragraphs should have given you an inkling of some homework - which will not be marked - feel free to try out the implied exercises for yourself. The only problem with the array of values is that you never know how big to make the table and you need some method of determining if the table has been initialised (to all zeros) GWASL doesn't fill buffers with zeros, just with assorted random characters, unlike an array in SuperBasic.

The array could be set up as follows:

```
Answers    dc.l     0           ; Fib(0), 0 indicates uninitialised table
           ds.l     1000        ; Fib(1) through Fib(1000)
```

Listing 13.4: Improving the Fibonacci Code - Answers Array

Because GWASL won't initialise the entries for 2 to 1000 you have to do it yourself, as follows:

```
Init       lea      Answers,a3      ; Start of answer array
           cmpi.l   #0,(a3)         ; Has table been initialised yet?
           bne.s    Done            ; Yes, exit
           move.w   1000,d0         ; 1000 = 1001 entries
I_Loop     clr.l    (a3)+           ; Clear this entry, point at next
           dbra     d0,I_Loop       ; Do the rest
           lea      Answers,a3      ; Answer(0)
           move.l   #1,(a3)+        ; Initialised to Fib(0)
           move.l   #1,(a3)         ; An initialise Fib(1) as well.
```

Listing 13.5: Improving the Fibonacci Code - Array Initialisation

Code like the above should be called at the start of the file so that the initialisation is only ever performed once per session. Making multiple calls to the Fibonacci code will only require the table be initialised once.

When the code has calculated Fib(N-1) then it can store the result in D0.L into the table. As N-1 is on the stack, it can be retrieved into a spare register - say D1 - and converted to an offset by shifting it right twice (LSL.L #2,D1) and using that as an offset into the answer table.

Now, when asked to calculate a value, check the offset into the table and if it is not zero, return that as your answer - no recursion and much faster. You'll have to remember to limit the number of allowable values if using a table - you could end up corrupting some random bits of memory and the amount of space you need to ALCHP will go up as a result of the table - check it after assembly to see how big it is.

Have fun and if you feel brave, Dilwyn wrote a SuperBasic version of 'The Towers Of Hanoi' some time back, why not convert that to Assembly language:o)

13.3 Coming Up...

The next chapter takes a bit of a breather from all this hard work writing code. In it, I'll discuss *my* own personal methods of writing code.

14. Program Development

14.1 Introduction

In this chapter I'll be going through the way I tend to write my assembly language (and indeed, all my other languages too) programming from the initial thought to the 'final completed' program. I put 'final' and 'completed' in quotes because programs never ever reach that stage there are always bugs to fix and improvements to be made.

14.2 Program Development in Assembly Language

Program development is the art of starting from nothing more than an idea and progressing with various stages until the thought becomes reality, or is discarded as unworkable.

We don't all do things the same way, and assembly language programming is no different - we all do it in a manner that is comfortable for us. The following lets you have a brief glance into my own methods.

14.2.1 The Initial Thought.

The initial idea for a program comes at the most inopportune moments I have found. I've had 'great' ideas at three in the morning, at other times when I was in the bath reading a novel, while driving to work and so on. The fact is, you never know when an idea will suddenly appear, so be prepared and have a bit of paper and a pen handy - not while driving of course - to jot down your ideas before they vanish from memory forever.

14.2.2 Work It Out.

Sometimes, given a little thought, the initial idea is found to be not so good after all and the project is abandoned there and then. Those ideas that get through need to be fleshed out a little to see just how good they are.

If they get past this stage, we can start to jot down the basic structure of our program. I personally tend to start with 'the big idea' and break it down into stages before breaking these down into smaller stages and so on until I have a set of small (hopefully) self contained routines at the bottom. This is top down development and used to be quite popular.

14.2.3 Start Writing Code.

At this point, armed with your list of routines, you can begin to write down your initial thoughts for the code you want to write to make the 'big idea' come to fruition. Having all the routines broken down by the previous stage, you know where repeated code can be extracted to a sub-routine and so on.

I tend to use a pencil to write code at this stage and arm myself with a decent rubber (eraser for my American readers!) because mistakes will be made. I also arm myself with three books:

- Andy Pennell's QDOS manual.
- Andy Pennell's Assembly Language Programming book.
- My trusty copy of the Motorola MC68000/MC68008 Programmer's Guide.

I also have a cheap narrow feint ruled A4 sized notepad to do my coding on. I then let my brain run away with itself to see how many different mistakes I can make in as short a time as possible.

Even after all these years, I still write down assembly code that just isn't legal syntax and this is sometimes 'obvious' when I look over the code, but usually I notice when George's trusty assembler (GWASL) complains about something in my code.

As I produce code for one routine. I usually find myself needing another, so I note it down on my list and carry on. This 'stepwise refinement' of my rough draft usually produces code that will be typed in using my trusty PFE text editor. This isn't a QL program, it runs on Windows, but I've used it form many years to write code and I prefer it. It allows me to save code in Linux format - which just happens to be the same as the QL's format and I like it.[1]

Once I have the code typed into a file, it gets saved to my C:\ or D:\ drive ready for import into QPC. Within QPC, my code files are copied from the DOS_ device to my RAM_ disc and GWASL is called into action. It almost never assembles first time.

QED is fired up and I make my changes to the RAM_ version, saving the file to DOS_ as a backup. Once I have a code file that actually assembles, I save the whole lot to WIN1_SOURCE_ and get ready to test it all out.

14.2.4 Testing The Code.

I tend to look on the bright side of most things, and running my own code is always fun. I simply EX the binary file and see what happens. Usually, it's a crash or system lock-up and I have to reboot. At least rebooting QPC takes a lot less time that rebooting Windows.

So, I know that there is at least one bug in my code and so it's bug hunting time again. After reloading, I run my next test with a code listing and JMON/QMON to trace through the code.

I wrote an article recently about debugging with JMON/QMON so I wont go into great detail here. (See Appendix B Debugging with QMON2)

[1] Since that was originally written I've converted over to Linux.

Suffice to say, my initial trace starts off with me single stepping up to each sub-routine call, then let each sub-routine run as a single unit. This way, I tend to quickly find out where my major problem lies.

After another reboot - if required - I use the procedures outlined in my JMON/QMON article (See Appendix B Debugging with QMON2) to set a break point at the 'broken' sub-routine, and I run the code to that point. From there on, I trace the code one line at a time until I hit a sub-sub-routine and let that run as a unit again. Once more, I quickly narrow my search for the main problem down to a single (or a couple) of small bits of code.

This code is then breakpointed and tested again, but in single step mode all the way through.

Eventually I either find the offending line(s) and fix them, or I find out which conditional branch I've got the wrong way round - I have been known to `BCC` when I should have used `BCS` and so on.

The rest of the process is similar to the above. It may not be the best in the world, but it works for me and I can quickly get debugged code finished and start 'tarting' it all up.

To show how easy most of the above is, I am going to work through a full example of 'an idea' from initial rough draft onwards to the finished code. I'm still working on this code at the moment and will not be writing up the article until I'm finished. I shall be documenting the process as I go and will write the article up from that.

You will no doubt have read some of my rants and raves about the Disassembler I'm writing as a project for this series. It has been developed bit by bit without any of the above 'discipline' so it has suffered from an extremely large number of errors, some stupidity on my part and a couple of rewrites in places. As I've said before, this is not how I wanted to write the utility but I'm somewhat stuck with it now. It shows how much better things are when you do it properly.

14.3 Coming Up...

Coming up in the next chapter we have a discussion of problems with QDOSMSQ EXECable files being downloaded from the internet and how to best recreate the correct dataspace settings.

V

SuperBasic, QDOS and Other Interesting Stuff. Part 3

15. Dataspace Problems

15.1 Introduction

There has been much correspondence recently (today is the 3rd of April 2006[1])[2] on the ql-users email list since Dilwyn posted a messages about the seeming "Catch- 22" problems where someone downloads a QL application from the internet and has to extract the zip file on a Windows PC. The files thus extracted are then read into QPC (or similar) and subsequently, any executable files fail to work.

The problem is caused by the need for QL files to have a correct file header which has the file type byte set to be EXECable and a valid value in the data space part of the header.

Getting hold of a QL specific version of Zip/Unzip is quite simple, but it arrives in a zip file so we have a recursive problem here. Dilwyn's advice is quite simple, load the program into memory and SEXEC it with the correct data space. This is indeed a simple solution, but I wouldn't be writing this article if I didn't have an alternative would I?

Actually, there is a version of Zip for the QL which has been converted to run as an EXECable file that will extract itself, so this is the easiest solution overall. However, the utility I describe below can be used to make any file executable and to provide a data space value without having to read the file off disc, delete it and then SEXEC a new copy back to the disc - what happens if you have a crash just after the delete?

The code below is based on a utility I wrote many years ago (1991 actually) that was supplied with WinBack when it was still a commercial program. I have only had to make a slight change to it for this version.

In the old days, the utility checked to make sure that the file was already executable and failed if not, this version doesn't fail, it simply changes the file to be executable.

[1] Actually, today is the 15th November 2014!

[2] No it isn't! It's *much* later than that!

When you EXEC the dataspace_bin file, you will be prompted for a filename. Type one in and if it is not an executable already, the program will advise you of this - then convert it to an EXEC file.

Next you will be shown the current data space and asked for a new value. The value you type should be numeric (or numeric with a 'k' at the end) and even. If not even it will be rounded up to the next even number.

If the value you type is less than the minimum the program allows (default is 1024 bytes), then your value will be rounded up to the minimum value. If you don't like this, then set minimum to zero in the source code and it will be ignored.

To finish the program, simply press ENTER when prompted for a filename.

15.2 The Code

We shall begin the typing with the usual set of equates and constants, type the following into a file - I suggest you name it dataspace_asm, but this is not mandatory.

```
*=====================================================================*
* DATASPACE version 1.10                                              *
*                                                                     *
* Copyright Norman Dunbar February 1991/2006                          *
*                                                                     *
* Changes a task file's dataspace.                                    *
*=====================================================================*
*                                                                     *
* AMENDMENTS                                                          *
*                                                                     *
* 03/04/2006 - makes files executable if not and doesn't complain.    *
*=====================================================================*

*---------------------------------------------------------------------*
* EQUATES General                                                     *
*---------------------------------------------------------------------*
ftyp        equ     $05             Offset to file type
fdat        equ     $06             Offset to file dataspace
exec_file   equ     $01             Indicator file can be EXEC'd
minimum     equ     1024            Minimum dataspace allowed

me          equ     -$01            This job
linefeed    equ     $0A             Ascii line feed
timeout     equ     -$01            Infinite timeout
black       equ     $00             Black ink/paper code
red         equ     $02             Red ditto
green       equ     $04             Green ditto
white       equ     $07             White ditto
```

Listing 15.1: Dataspace Program - Equates etc

Following on from the above, we have the standard QDOSMSQ job header. From the code that follows below, you can see the header with the job name and version in the usual format for a QDOSMSQ job.

```
*=====================================================================*
* The code starts here with a standard QDOS job header.               *
*=====================================================================*
```

```
start       bra.s    dataspace           Jump over header
            dc.l     0                   Make sure $4AFB is at offset 6
            dc.w     $4AFB               ID word

name        dc.w     22                  Length of name
            dc.b     'Dataspace Version 1.10'

*-------------------------------------------------------------------*
* Open a console window                                             *
*-------------------------------------------------------------------*
dataspace   move.w   ut_con,a2
            lea      con_def,a1          Console definition block
            jsr      (a2)                Open a CON_ channel
            tst.l    d0                  Check for errors
            bne      job_end             And bale out if found
            move.l   a0,d7               Store console id

*-------------------------------------------------------------------*
* Console is open, sign on                                          *
*-------------------------------------------------------------------*
sign_on     lea      name,a1             Job name
            bsr      write_text          Print job name
            lea      copyright,a1        Copyright message
            bsr      write_text          And copyright message
```

Listing 15.2: Dataspace Program - Part 1 - Initialisation

After the job header, the first part of the program performs all the initialisation that is required. The job opens a new console channel and saves the channel number in D7.L. D7.L is not corrupted by any of the code that follows, so it is a good place to save the channel number. It will be used each time we pass through the main loop.

It is slightly more efficient to move data between two registers than it is to fetch from a memory location. In this utility, that's hardly going to be needed, but we might as well use all the registers before we have to start saving data in memory.

Once the main console channel is open, we print a small sign on message showing the job name (extracted from the standard job header area) and a copyright message and then drop into the main processing loop.

Please note that although this code is copyright, you have my express permission to use and abuse it as you see fit. If the code can be useful in programs that you write in future, please feel free to copy it directly with my blessings!

You will note that much of this program is written as simple subroutine calls. Once again, reusing existing and working code is always a good idea in my book.

```
*-------------------------------------------------------------------*
* Main loop, keep asking for a filename until ENTER only pressed    *
*                                                                   *
* First prompt for filename                                         *
*-------------------------------------------------------------------*
main_loop   move.l   d7,a0               Console id
            lea      mess_1,a1
            bsr      write_text          Enter filename message
```

```
                bsr       get_text              Get filename
```

Listing 15.3: Dataspace Program - Part 2 - Get Filename

The main loop starts off by getting the console channel number into A1 which is where most (if not all) the channel handling routines expect it to be. D7 is never corrupted by the code below so makes a good place to save it.

The user is prompted to ‘enter a filename or press ENTER only to quit’, the program waits for a response from the user and if the user simply pressed ENTER, the program will skip off to the code at label ‘any_key’ to terminate the program.

```
*------------------------------------------------------------------------------*
* Then check if it was only ENTER and if so exit the job                       *
*------------------------------------------------------------------------------*
check_end       tst.w     d1                    Finished?
                beq       any_key               Yes

*------------------------------------------------------------------------------*
* Otherwise attempt to open the file                                           *
*------------------------------------------------------------------------------*
open_file       moveq     #io_open,d0           Open file
                moveq     #me,d1                For this job
                moveq     #0,d3                 For input
                move.l    a1,a0                 File name is in buffer
                trap      #2
                tst.l     d0                    Check errors
                beq.s     read_head             Open was ok

*------------------------------------------------------------------------------*
* Cannot open the file, print its name and the error message                   *
*------------------------------------------------------------------------------*
cant_open       move.l    d0,-(a7)              Store error code
                lea       mess_2,a1             Cannot open message
                bsr       write_text            Print it
                lea       input,a1              File name
                bsr       write_text            Print filename
                lea       mess_6,a1             A colon
                bsr       write_text            Print it
                move.l    (a7)+,d0              Get error code
                move.w    ut_err,a2
                jsr       (a2)                  Print error message

*------------------------------------------------------------------------------*
* Note, D0 is preserved by UT_ERR, so cannot check for errors                  *
*------------------------------------------------------------------------------*

                bra.s     main_loop
```

Listing 15.4: Dataspace Program - Part 3 - Open the File

Assuming that we got a response back from the user, we assume that it is a filename which the user wishes to either make executable, or adjust the data space. The code at ‘open_file’ above attempts to open the filename supplied by the user for input. This means that it must already exist on a device somewhere.

If the file opened ok, the program skips to the code below which attempts to read the file header, otherwise the user is presented with a message detailing why the file couldn't be opened, and we skip back to the start of the main loop where we prompt for another filename.

You will note my comment that `UT_ERR` preserves the error code passed to it in D0, so we cannot make any checks to determine whether the error code in D0 is the one we passed in, or one generated by UT_ERR. Don't worry about it:o)

A quick tangent is required now, before we proceed. On a directory device such as 'flp1_', 'mdv1_' and so on, all files have two parts to them. First of all there is the file header - a 64 byte section of data which is stored in the directory (and allegedly at the start of each file too - you just can't get at it!) - and the contents of the file proper.

This header holds details of the file size, it's type, data space, various dates giving details of when the file was last changed, backed up etc. We are interested in only two parts of the header in this utility - the file type (byte) and the file's data space (long).

We defined a couple of equates way back at the start of the code - the 'ftyp' equate points at byte 5 of the 64 byte buffer and the 'fdat' points at the long word starting at byte 6 of the header. These are the two places we need to get data from and write data to.

On with the real code again. The following reads a file header and processes errors as required.

```
*------------------------------------------------------------------------------*
* File has been opened ok, read the file header                                *
*------------------------------------------------------------------------------*
read_head   move.l   a0,d6                Store file id
            moveq    #fs_headr,d0
            moveq    #64,d2               Size of buffer
            moveq    #timeout,d3          Timeout
            lea      buffer,a1            Put header here
            move.l   a1,a5                Store buffer
            trap     #3                   Go get the file header
            tst.l    d0                   Check for errors
            beq.s    check_exec           none

*------------------------------------------------------------------------------*
* Cannot read file header, say so and print the error message                  *
*------------------------------------------------------------------------------*
cant_read   lea      mess_3,a1            Cannot read header message
            move.l   d0,-(a7)             Store error code
            bsr      write_text           Print message
            move.l   (a7)+,d0             Get error code
            move.w   ut_err,a2
            jsr      (a2)                 Print error message
            bra      main_end             Skip the rest of the loop
```

Listing 15.5: Dataspace Program - Part 4 - Read File Header

The code above tries to read the 64 byte file header into a buffer - which must be big enough to hold all 64 bytes - and if an error is detected in the attempt, the user is told about the problem and the file is closed prior to the main loop starting all over again.

The buffer start address is saved in register A5.L for later use prior to the attempt to read the file header - this is because the real buffer address register, A1.L will be corrupted by the trap call.

```
*------------------------------------------------------------------------------*
* Header has been read ok, check if the file is EXECable                       *
*------------------------------------------------------------------------------*
```

```
check_exec cmpi.b   #exec_file,ftyp(a5) Check file is EXECable
           beq.s    current             It is

*-------------------------------------------------------------*
* File is not EXECable, print a warning message and convert it *
*-------------------------------------------------------------*
not_exec   lea      input,a1            Filename
           bsr      write_text          Print filename
           lea      mess_4,a1           Not an EXECable file message
           bsr      write_text          Print the message
           move.b   #exec_file,ftyp(a5) Make file EXECable

*-------------------------------------------------------------*
* File is EXECable, print its current dataspace                *
*-------------------------------------------------------------*
current    lea      mess_5,a1           Current dataspace is message
           bsr      write_text          Print it
```

Listing 15.6: Dataspace Program - Part 5 - Exec Check

The file is now open, its header has been read into our buffer. The code now checks to see if this file is executable already. If it is, nothing is said or done, however, if the file is not executable - and this will be the case for a file extracted by a Windows version of Zip - we display a warning message to the user and convert the file to be EXECable.

In either case, we print a message which will eventually inform the user how big the file's dataspace is at the moment. The first part of the message is easy - it is simple text but we also need to print out the current value. This is done by the code below.

```
*-------------------------------------------------------------*
* Now get the current dataspace & convert it to ASCII          *
*-------------------------------------------------------------*
           move.l   fdat(a5),d3         D3.L is dataspace
           lea      input+2,a1          A1.L is output buffer
           lea      tens_table,a2       Powers of 10
           moveq    #0,d1               D1.W is digit counter

next_digit move.l   (a2)+,d2            D2.L is current power of 10
           beq.s    all_done            But zero is end of table
           clr.b    d0                  D0.B is current digit

digit_loop sub.l    d2,d3               Subtract the current power of 10
           blt.s    buff_digit          Too far
           addq.b   #1,d0               Increase current digit
           bra.s    digit_loop          And try again

buff_digit add.l    d2,d3               Correct for the overflow
           tst.b    d0                  Is this a zero?
           bne.s    not_a_zero          No
           tst.w    d1                  Yes, is it a leading zero?
           beq.s    next_digit          Yes, ignore it

not_a_zero addi.b   #'0',d0             Convert to ASCII
           move.b   d0,(a1)+            Store in buffer
           addq.w   #1,d1               Increment total digits
           bra.s    next_digit          And do the rest

```

```
*------------------------------------------------------------------------*
* Check for a result of zero. In this case force a '0' to be printed    *
*------------------------------------------------------------------------*
all_done    tst.w    d1                    Any digits found?
            bne.s    not_zero              yes
            move.b   #'0',(a1)             Store a zero
            moveq    #1,d1                 And set the count

not_zero    lea      input,a1              The buffer
            move.w   d1,(a1)               Store character count

*------------------------------------------------------------------------*
* Dataspace is converted, print it out                                   *
*------------------------------------------------------------------------*

            bsr      write_text            Print old dataspace
```

Listing 15.7: Dataspace Program - Part 6 - Print Current Dataspace

The code above begins by reading the long word representing the file data space requirements from the file header into register D3.L.

D3.L is then converted to text read for printing by the fairly simple method of repeatedly subtracting assorted powers of 10 from the value until we get an overflow (underflow?) and saving the number of times we managed to successfully subtract the current power of 10. This count is our current digit and is held in D0.B as it cannot be greater than 9.

If the counter is non-zero, we convert it to ASCII by adding the ASCII code for '0' (zero) to the count and save it in the buffer located at A1. We only print out the full number when we have decoded it - we don't print each digit individually as we go along.

If the dataspace is still zero, even after processing all the digits, we simply print a zero.

```
*------------------------------------------------------------------------*
* Now prompt for, and read in the required new dataspace                 *
*------------------------------------------------------------------------*
new         lea      mess_8,a1             New dataspace message
            bsr      write_text            Print it
            bsr      get_text              Get new dataspace
            tst.w    d1                    No text?
            beq.s    new                   Try again
            move.w   d1,d0                 Get text length
            subq.w   #1,d0                 Adjust for dbra
            addq.l   #2,a1                 Adjust A1 past the word counter
```

Listing 15.8: Dataspace Program - Part 7 - Get New Dataspace

Having printed out the current data space for the file in question, we next prompt the user to enter a new value. If the user simply presses ENTER without entering a value, the program detects this, and simply loops around to ask for the new data space value again. To get out of this loop a valid numeric value must be entered.

The utility accepts pure digits or a number of digits suffixed by a 'K' (in upper or lower case) and this is used to specify a data space in Kilobytes rather than bytes. If there are spaces in the user input, they will simply be skipped.

```
*------------------------------------------------------------------------*
* Convert from ASCII into binary, ignore leading (any) spaces & stop *
```

```
* if a 'K' or 'k' is detected. Reject all other non-digit characters *
*----------------------------------------------------------------------*
convert     moveq    #0,d4              Needs to be a long word
            move.l   d4,d5              D5 is total so far

conv_next   move.b   (a1)+,d4           Get a byte
            cmpi.b   #' ',d4            Is it a space?
            beq.s    try_next           Yes, ignore it
            cmpi.b   #'k',d4            Is it 'k'
            bne.s    try_K              no

mul_1024    asl.l    #2,d5              Yes, multiply by 1024
            asl.l    #8,d5              Can't do it in one go
            bra.s    make_even          And exit

try_K       cmpi.b   #'K',d4            Try uppercase
            beq.s    mul_1024           Yes

            cmpi.b   #'0',d4            Is it a digit?
            bcs.s    invalid            No
            cmpi.b   #'9',d4            But it might be
            bls.s    mul_10             Yes it is

*----------------------------------------------------------------------*
* An invalid digit has been detected, print error message & try again *
*----------------------------------------------------------------------*
invalid     lea      mess_10,a1         Invalid digit message
            bsr.s    write_text         Print it
            bra.s    new                try again
```

Listing 15.9: Dataspace Program - Part 8 - ASCII Conversion

As we scan through the input supplied by the user we ignore any spaces. I could have written the code to detect when the first digit had been detected and processed spaces after that as errors, but I was obviously too lazy to do so back in 1991. (Not much has changed in 2006[3][4] then!)

Each character is checked and if it is a 'K' (in any letter case) it indicates the end of the input and the current value is multiplied by 1024 to get the correct number of bytes.

If an invalid character is detected, an error message is printed and the user restarts the inner loop of the program where s/he is prompted to type in a new data space value.

```
*----------------------------------------------------------------------*
* Multiply D5.L by 10 and add in the digit just read                   *
*----------------------------------------------------------------------*
mul_10      asl.l    #1,d5              D5 = D5 * 2
            move.l   d5,d3              Store for now
            asl.l    #2,d5              Now D5 = D5 * 8
            add.l    d3,d5              And finally D5 = D5 * 10
            subi.b   #'0',d4            Convert byte to (long) binary
            add.l    d4,d5              Total = (total * 10) + digit

```

[3] 2006? Have I been writing Assembly articles that long? It's 2014 at the moment and 2015 is rapidly approaching. I'll be getting a bus pass pretty soon at this rate!

[4] Ahem!

```
try_next    dbra    d0,conv_next        Do rest of digits
```

Listing 15.10: Dataspace Program - Part 9 - Multiply by 10

If we get to the code above, then the current character in the user input must be a digit. We multiply the running total so far by 10, convert the latest digit from an ASCII character down to a numeric value and add it to the running total in D5.L before skipping back to continue scanning the input area for another digit.

```
*-------------------------------------------------------------------*
* When finished, the value must be even                             *
*-------------------------------------------------------------------*
make_even   addq.l  #1,d5               Prepare to make even
            bclr    #0,d5               Make dataspace even

*-------------------------------------------------------------------*
* And, not less than minimum allowed                                *
*-------------------------------------------------------------------*
            cmpi.l  #minimum,d5         Check new size
            bcc.s   set_head            It's ok.
            move.l  #minimum,d5         Make sure it is = minimum size
```

Listing 15.11: Dataspace Program - Part 10 - Final Checks

Eventually, we exit from the loop scanning the user's input and arrive at the code above. This is a short but very important piece of code. The data space for a file must be even. If it is odd, then any attempt to EXEX (EX etc) the code will result in an address exception with the usual resulting lock up. Just say no to odd addresses!

The running total in D5.L is rounded up to the next even number, or left alone if it is already even.

If the running total is less than our minimum allowed value, it is set to that value.

```
*-------------------------------------------------------------------*
* Now load the header with the new dataspace and set it             *
*-------------------------------------------------------------------*
set_head    move.l  d5,fdat(a5)         Store in the header
            moveq   #fs_heads,d0        Set the file header
            moveq   #timeout,d3         Timeout
            move.l  d6,a0               File id
            move.l  a5,a1               File header
            trap    #3                  Go set it
            tst.l   d0                  Any errors?
            bne.s   not_set             Yes
```

Listing 15.12: Dataspace Program - Part 11 - Write Header

As we now have a new data space value in D5.L, we save it in the file header buffer in the correct location and call the QDOSMSQ trap call to write the file header back to the device. If this works ok, we drop into the following code to flush the changes to the device.

QDOSMSQ is happy to save changes in the file slave area until it has a moment to write them out. This is ok in most cases, but if a user wishes to remove a floppy disc, for example, that we must make sure that all changes are written down to the disc. The following code does that task, closes the file and starts the main loop all over again.

```
*-------------------------------------------------------------------*
* Now flush out the file buffers, so that if I change discs I have  *
```

```
* written the header to the current disc. Can't detect the QDOS error *
* READ/WRITE failed (try removing the disc and it won't fail or       *
* produce an error code). It might print a message if it can find an *
* open command channel which is not in use. I got it when testing via *
* a monitor but not while running on its own.                        *
*---------------------------------------------------------------------*
flush      moveq   #fs_flush,d0        Prepare to flush the buffer
           moveq   #timeout,d3         This could take all day
           trap    #3                  But do it anyway
           tst.l   d0                  And check for errors
           beq.s   main_end            None, do the next file
```

Listing 15.13: Dataspace Program - Part 12 - Flush Buffers

If the file header failed to be written, the following code will inform the user that there is a problem and display the QDOSMSQ error message.

```
*---------------------------------------------------------------------*
* Header not set or flush failed, print error message                 *
*---------------------------------------------------------------------*
not_set    move.l  d0,-(a7)            Store error code
           lea     mess_7,a1           Cannot set header message
           bsr.s   write_text          Print it
           move.l  (a7)+,d0            Get error code
           move.w  ut_err,a2
           jsr     (a2)                Print error message

*---------------------------------------------------------------------*
* Can't trap errors in UT_ERR as D0 is preserved.                     *
* Close the file & loop to the start                                  *
*---------------------------------------------------------------------*
main_end   move.l  d6,a0               File id for close
           moveq   #io_close,d0
           trap    #2                  Close the file
           bra     main_loop           And see if more to be done
```

Listing 15.14: Dataspace Program - Part 13 - Error Handling

We have now reached the end of the main loop. The code above retrieves the file's channel number from register D6.L, closes the file and skips back to the start of the main loop ready for the next file to be processed.

The code below is a collection of simple subroutines, some you will have seen before, which carry out various useful parts of the program. The comments above each should be sufficient to explain what is going on.

Also included below are some data input and header buffer areas.

```
*---------------------------------------------------------------------*
* Subroutine to print text to screen                                  *
*                                                                     *
* ENTRY                                                               *
*                                                                     *
* D7.L = Channel id                                                   *
* A1.L = Pointer to text to print (Word then bytes)                   *
*                                                                     *
* EXIT                                                                *
*                                                                     *
```

```
* A0.L = channel id                                                    *
*----------------------------------------------------------------------*
write_text  move.w   ut_mtext,a2          Print text vector
            move.l   d7,a0                Channel id
            jsr      (a2)                 Print it
            tst.l    d0                   Check errors
            bne.s    job_error            Oops kill job
            rts                           Otherwise exit

*----------------------------------------------------------------------*
* A fatal error has occurred, print it, wait for any key and kill job *
* wait for key allows WMAN & PTR_GEN users to see the message before  *
* WMAN restores the screen.                                            *
*----------------------------------------------------------------------*
job_error   move.l   d7,a0                Get console id
            move.w   ut_err0,a2           Print error text vector
            jsr      (a2)                 Print to #0

any_key     move.l   d7,a0                In case entry is here
            lea      mess_12,a1           Press any key message
            move.w   ut_mtext,a2          Don't use WRITE_TEXT
            jsr      (a2)                 Print it
            moveq    #io_fbyte,d0         Fetch one byte
            moveq    #timeout,d3          Take all day if you like
            trap     #3                   Go get it
            moveq    #io_close,d0
            trap     #2                   Close console channel

*----------------------------------------------------------------------*
* This job will self destruct in no time at all                        *
*----------------------------------------------------------------------*
job_end     moveq    #mt_frjob,d0         Job is about to die
            moveq    #me,d1               And it is this job
            trap     #1                   RIP (there is not return)

*----------------------------------------------------------------------*
* Subroutine to get some text from the user                            *
*                                                                      *
* ENTRY                                                                *
*                                                                      *
* D7.L = channel id                                                    *
*                                                                      *
* EXIT                                                                 *
*                                                                      *
* D1.W = number of bytes read                                          *
* A0.L = channel id                                                    *
* A1.L = start of buffer (word then bytes)                             *
*----------------------------------------------------------------------*
get_text    lea      input,a1             Buffer for the text
            move.l   a1,-(a7)             Store it
            addq.l   #2,a1                Leave room for the length word
            moveq    #io_fline,d0
            moveq    #42,d2               Maximum buffer size
            moveq    #timeout,d3          Take as long as you like
            trap     #3                   Get some text
            tst.l    d0                   Check for errors
```

```
              bne.s    job_error              Bale out (stack will be ok)
              move.l   (a7)+,a1               Get buffer start
              subq.w   #1,d1                  Remove the line feed
              move.w   d1,(a1)                Store text length
              rts                             Exit

*------------------------------------------------------------------------------*
* Definition block for my console channel                                      *
*------------------------------------------------------------------------------*
con_def       dc.b     black                  Border colour
              dc.b     $01                    Border width
              dc.b     white                  Paper & strip colour
              dc.b     black                  Ink colour
              dc.w     $01C0                  Width = 448
              dc.w     $0064                  Height = 100
              dc.w     $0020                  X position = 32
              dc.w     $0010                  Y position = 16

*------------------------------------------------------------------------------*
* Copyright message, so the world knows my name                                *
*------------------------------------------------------------------------------*
copyright     dc.w     copy_end-copyright-2
              dc.b     linefeed
              dc.b     'Copyright Norman Dunbar, Jan 1991/April 2006.'
              dc.b     linefeed
copy_end      equ      *

*------------------------------------------------------------------------------*
* Various prompts & error messages                                             *
*------------------------------------------------------------------------------*
mess_1        dc.w     end_1-mess_1-2
              dc.b     linefeed
              dc.b     'Enter filename'
              dc.b     linefeed
              dc.b     'or ENTER only to finish: '
end_1         equ      *

mess_2        dc.w     end_2-mess_2-2
              dc.b     'Cannot open '
end_2         equ      *

mess_3        dc.w     end_3-mess_3-2
              dc.b     'Cannot read file header: '
end_3         equ      *

mess_4        dc.w     end_4-mess_4-2
              dc.b     ' is being converted to an EXECable file'
              dc.b     linefeed
end_4         equ      *

mess_5        dc.w     end_5-mess_5-2
              dc.b     'Current dataspace is: '
end_5         equ      *

mess_6        dc.w     end_6-mess_6-2
              dc.b     ': '
```

```
end_6       equ     *

mess_7      dc.w    end_7-mess_7-2
            dc.b    'Cannot set file header: '
end_7       equ     *

mess_8      dc.w    end_8-mess_8-2
            dc.b    linefeed
            dc.b    'Enter new dataspace in bytes, or'
            dc.b    linefeed
            dc.b    'end with "K" for kilobytes: '
end_8       equ     *

mess_9      dc.w    end_9-mess_9-2
            dc.b    ' bytes'
            dc.b    linefeed
end_9       equ     *

mess_10     dc.w    end_10-mess_10-2
            dc.b    'Invalid digit found in input'
            dc.b    linefeed
end_10      equ     *

mess_11     dc.w    end_11-mess_11-2
            dc.b    'Dataspace set.'
            dc.b    linefeed
end_11      equ     *

mess_12     dc.w    end_12-mess_12-2
            dc.b    linefeed
            dc.b    'Goodbye, press any key to kill job.....'
end_12      equ     *

*----------------------------------------------------------------------*
* Two buffer areas, one for the file header & one for user input      *
* note how sneaky I have been, by using DS.W I have forced them both  *
* to be word aligned. If I had used DC.B they might not have been, &  *
* I would be bound to get an address exception sometime. (it happened)*
*----------------------------------------------------------------------*

buffer      ds.w    32                  Buffer is 64 bytes maximum
input       ds.w    22                  Size = 41 + ENTER + word count

*----------------------------------------------------------------------*
* A table of all powers of ten, from 9 to 0. This corresponds to the  *
* values used when converting an UNSIGNED long word to ASCII.         *
* 2^31 = 2,147,483,648                                                 *
* 10^9 = 1,000,000,000 so is a big enough 'highest' power to use      *
*----------------------------------------------------------------------*
tens_table  dc.l    1000000000          10 ^ 9
            dc.l    100000000           10 ^ 8
            dc.l    10000000            10 ^ 7
            dc.l    1000000             10 ^ 6
            dc.l    100000              10 ^ 5
            dc.l    10000               10 ^ 4
            dc.l    1000                10 ^ 3
```

```
            dc.l    100                 10 ^ 2
            dc.l    10                  10 ^ 1
            dc.l    1                   10 ^ 0
            dc.l    0                   Table end marker
```

Listing 15.15: Dataspace Program - Part 14 - Various Subroutines

15.3 Coming Up...

The next chapter takes a look at the maths package supplied deep in the bowels of QDOSMSQ and shows a couple of examples of creating your own routines to perform lots of complicated arithmetic routines.

16. Using the Maths Package

16.1 Introduction

Don't worry, it's not as bad as it sounds. What I'm talking about is the internal package of routines, provided by the operating system, to allow various mathematical operations to be carried out. For example, multiplying two floating point numbers together, or finding the square root of a number and so on.

16.2 The Maths Package

The two entry points to this useful set of routines is known (in old QDOS format) as `RI_EXEC` - which carries out a single operation - and `RI_EXECB` - which carries out a stream of operations. For SMSQE users the names are `QA_OP` and `QA_MOP` respectively.

These are vectored routines which simply means that you can find where they are by loading the contents of a word in memory. If the actual location of the code moves around between versions of SMSQE (As QDOS is not being updated) then the vectors remain in the same place.

To call a vectored routine is quite simple. All you do is set up the entry registers as per the QDOSMSQ documentation, load an address register with the vector and JSR (An) as follows:

```
move.w   ca_gtfp,a2          ; Fetch floating point parameter(s)
jsr      (a2)                ; Do it
```

Listing 16.1: Example Code, Calling a Vectored Routine

On return, D0 will be set to an error code or zero - with the Z flag set accordingly - if it all worked ok. All current vectors are WORD sized by the way.

Without any further hesitation, lets jump straight in with some example code. The following short routine shows the `RI_EXEC` entry point to the maths package in use. It is a simple demonstration and creates a new function names ROOT which simply returns the square root of its single parameter.

```
*------------------------------------------------------------------------
* Equates as required.
*------------------------------------------------------------------------
err_bp      equ     -15                 ; Bad parameter error
bv_rip      equ     $58                 ; Maths stack pointer
ri_sqrt     equ     $28                 ; Op code for square root

*------------------------------------------------------------------------
* Usual start block for PROCedure and FuNction extensions.
*------------------------------------------------------------------------
start       lea     define,a1           ; Pointer to definition table
            move.w  BP_INIT,a2          ;
            jsr     (a2)                ; Call BP_INIT
            rts                         ; Exit to SuperBasic

*------------------------------------------------------------------------
* Definition block for our new function.
*------------------------------------------------------------------------
define      dc.w    0                   ; 0 new procedures
            dc.w    0                   ; End of procedures

            dc.w    1                   ; There is 1 function

            dc.w    root-*              ; First function
            dc.b    4,'ROOT'

            dc.w    0                   ; End of functions

*------------------------------------------------------------------------
* The actual start of the ROOT code is next.
*------------------------------------------------------------------------
root        move.l  a5,d7               ; End of parameters
            sub.l   a3,d7               ; Minus start of parameters
            cmpi.w  #8,d7               ; One parameter?
            beq.s   get_1               ; Yes
bad_param   moveq   #err_bp,d0          ; Bad Parameter error
quit        rts                         ; Exit

*------------------------------------------------------------------------
* The single floating point parameter is fetched next.
*------------------------------------------------------------------------
get_1       move.w  ca_gtfp,a2          ; We want a float variable
            jsr     (a2)                ; Fetch it
            beq.s   got_ok              ; Yes it did
            rts                         ; Bale out with error

*------------------------------------------------------------------------
* Check that it all worked.
*------------------------------------------------------------------------
got_ok      cmpi.w  #1,d3               ; One parameter?
            bne.s   bad_param           ; Oops!

*------------------------------------------------------------------------
* The value on the arithmetic stack is ready to be SQRTed.
*------------------------------------------------------------------------
do_it       moveq   #ri_sqrt,d0         ; Take square root
```

```
              moveq   #0,d7             ; Must be zero or crash!
              move.w  RI_EXEC,a2        ; Get vector
              jsr     (a2)              ; Do it
              bne.s   quit              ; Oops!

*------------------------------------------------------------------------
* If all went well, return the new value on the maths stack as a float.
*------------------------------------------------------------------------
ret_fp        moveq   #2,d4             ; Return FP number
              rts                       ; Exit with result
```

Listing 16.2: The Maths Package - Calculate Square Roots

Note You will see in the next chapter a conversation between George Gwilt and myself. The code above has been corrected from that in the original article according to George's comments.

Save the above code to a file (mine is called square_root_asm) and assemble it. Once done, LRESPR the resulting bin file (square_root_bin) and try it out as follows:

```
PRINT ROOT(9)
PRINT ROOT(100)
PRINT ROOT(25)
```

You can make sure that it is working properly by comparing the result from ROOT with the corresponding result for SQRT.

There is nothing complicated in the code. Most of the above is checking that we expect a single parameter and checking that everything worked on and so on. It is the last 8 lines of code that do the actual work and return the result to SuperBasic.

The example above shows how a single operation is carried out. What do you have to do if the mathematical operation you want to perform takes more than a single step?

The answer is simple, you build a list of steps as byte values and terminate them with a zero byte, then call `RI_EXECB` to execute the steps in order.

Here is another example which uses a relatively simple set of commands to work out the Nth root of any number. Sounds complicated but it is quite simply done using about the only bit of maths 'trickery' that I can remember from my time at school.

The following simple SuperBasic code will demonstrate:

```
1000 DEFine FuNction AnyRoot(m, n)
1010:
1020 REMark Returns the Nth root of the number M
1030:
1040 LOCal ln_m
1050:
1060 ln_m = LN(m)
1070 ln_m = ln_m / n
1080 RETurn EXP(ln_m)
1090 END DEFine
```

If you type the above into SuperBasic and call it as follows, you can calculate all the roots you want:

```
PRINT AnyRoot(100, 3): REMark calculate the cube root of 100
```

And so on. The code works and works quite well, however, as this is an Assembly Language tutorial series, I can't let you off the hook that easily! Here's the Assembly version.

```
*------------------------------------------------------------------------------
* Equates as required.
*------------------------------------------------------------------------------
err_bp      equ     -15                 ; Bad parameter error
bv_rip      equ     $58                 ; Maths stack pointer
ri_ln       equ     $2a                 ; Take LN of a number
ri_div      equ     $10                 ; Divide TOS into NOS
ri_exp      equ     $2e                 ; EXP of a number
ri_end      equ     $00                 ; End of opcodes list

*------------------------------------------------------------------------------
* Usual start block for PROCedure and FuNction extensions.
*------------------------------------------------------------------------------
start       lea     define,a1           ; Definition table
            move.w  BP_INIT,a2          ;
            jsr     (a2)                ; Call BP_INIT
            rts                         ; Back to SuperBasic

*------------------------------------------------------------------------------
* Definition block for our new function.
*------------------------------------------------------------------------------
define      dc.w    0                   ; 0 new procedures
            dc.w    0                   ; End of procedures

            dc.w    1                   ; There is 1 function

            dc.w    anyroot-*           ; First function
            dc.b    7,'ANYROOT'

            dc.w    0                   ; End of functions

*------------------------------------------------------------------------------
* The actual start of the ANYROOT code is next.
*------------------------------------------------------------------------------
anyroot     move.l  a5,d7               ; End of parameters
            sub.l   a3,d7               ; Minus start of parameters
            cmpi.w  #16,d7              ; Do we have two parameters?
            beq.s   get_2               ; Yes
bad_param   moveq   #err_bp,d0          ; Bad Parameter error
quit        rts                         ; Exit

*------------------------------------------------------------------------------
* The two floating point parameters are fetched next.
*------------------------------------------------------------------------------
get_2       move.w  ca_gtfp,a2          ; We want float variables
            jsr     (a2)                ; Fetch
            beq.s   got_ok              ; All ok
```

```
              rts                                ; Bale out on error

*------------------------------------------------------------------------
* Check that it all worked.
*------------------------------------------------------------------------
got_ok        cmpi.w    #2,d3                    ; Two parameters?
              bne.s     bad_param                ; Oops!
              bra.s     do_it                    ; skip over the op-codes

*------------------------------------------------------------------------
* A list of op codes to calculate the Nth root of M.
*------------------------------------------------------------------------
op_codes      dc.b      ri_div                    ; Divide TOS into NOS
              dc.b      ri_exp                    ; Take EXP of TOS
              dc.b      ri_end                    ; End of op codes

*------------------------------------------------------------------------
* At this point there are two values on the stack:
*
* 0(A6,A1.L) = M = Big value
* 6(A6,A1.l) = N = Root to find
*
* To work out our Nth root of M, we need to do the following:
*
* Take the LN of M.
* Divide it by N.
* Take the EXP of the result.
* Return it to SuperBasic.
*
* Of course, it's never as easy as it seems!
*------------------------------------------------------------------------
do_it         moveq     #ri_ln,d0                ; LN op code
              moveq     #0,d7                    ; Must be zero or crash!
              move.w    RI_EXEC,a2               ; Get vector for one op
              jsr       (a2)                     ; Do it
              bne.s     quit                     ; Oops!

*------------------------------------------------------------------------
* Now the stack is holding the following:
*
* 0(A6,A1.L) = LN(M)
* 6(A6,A1.L) = N = Root to find.
*
* They are the wrong way around :o(
*------------------------------------------------------------------------
swap_tos      move.l    0(a6,a1.l),d7            ; Get a long word
              move.l    6(a6,a1.l),d6            ; And another
              move.l    d6,0(a6,a1.l)            ; Store
              move.l    d7,6(a6,a1.l)            ; Store
              move.w    4(a6,a1.l),d7
              move.w    10(a6,a1.l),d6
              move.w    d6,4(a6,a1.l)
              move.w    d7,10(a6,a1.l)           ; Now we have N and LN(M) swapped

*------------------------------------------------------------------------
* The stack is how we want it to be, so we can continue.
```

```
*------------------------------------------------------------------------------
do_more     moveq     #0,d7               ; Or a crash will probably result
            move.w    RI_EXECB,a2         ; Perform lots of ops
            lea       op_codes,a3         ; Op codes to perform
            jsr       (a2)                ; Do the op list
            bne.s     exit                ; Oops!

*------------------------------------------------------------------------------
* If all went well, return the result on the arithmetic stack as float.
* Note that the maths stack is 6 bytes shorter now, so we have to save
*  the top in BV_RIP before we exit.
*------------------------------------------------------------------------------
ret_fp      move.l    a1,bv_rip(a6)       ; Make sure maths stack is set
            moveq     #2,d4               ; Return FP number
exit        rts                           ; Exit with result
```

Listing 16.3: The Maths Package - Calculate Any Root

Note The code above has been corrected from that in the original article according to George's comments that can be read in the next chapter.

Save the above code to a file (mine is called any_root_asm) and assemble it. Once done, LRESPR the resulting bin file (any_root_bin) and try it out as follows:

```
PRINT ANYROOT(9, 2)
PRINT ANYROOT(100, 3)
PRINT ANYROOT(25, 4)
```

There's not much of real interest in the above code. As ever we validate our parameters to make sure we only expect two then fetch them as floating point values onto the maths stack. After a check to see that we really did get two parameters, we have the values M and N on the stack with M being at the 'top' (TOS = top of stack) and N being underneath it (NOS = next on stack).

We start by running a single op code to calculate $\ln(M)$ which leaves the stack with a new TOS which is simply $\ln(M)$.

We next want to divide $\ln(M)$ by N but unfortunately, they are the wrong way around so we swap over the 6 bytes at 0(A6,A1.L) with the 6 bytes at 6(A6,A1.L) and then run a sequence of op codes to:

- Divide $\ln(M)$ by N leaving the result as the TOS.
- Calculate $\exp(\ln(M))$ as the new TOS

Once this has been done, we store the new value of A1 at BV_RIP(A6) as required, set the result to be a floating point number and exit to SuperBasic with the result.

As you may have noticed, the text above mentions that the math stack pointer (A1.L) can be changed by the various op codes that we execute. The following table gives you details on what op codes are available and how they manipulate the maths stack.

So, there you have it, a pile of ingredients all set for you to make up your own numerical recipes. Have fun.

Now, one thing that I have not mentioned above, or even used in the code examples is temporary storage. However, before I delve into that, it's best if you familiarise yourself with Ta-

Value	**OpCode**	**A1.L**	**Description**
$00	RI_END	=	End of op code list (RI_EXECB)
$02	RI_NINT	+4	Convert FP to Word INT
$04	RI_INT	+4	Truncate FP to Word INT
$06	RI_NLINT	+2	Convert FP to Long INT
$08	RI_FLOAT	-4	Convert Word INT to FP
$0A	RI_ADD	+6	Add TOS to NOS, remove TOS from stack
$0C	RI_SUB	+6	Subtract TOS from NOS, remove TOS from stack
$0E	RI_MULT	+6	Multiply NOS by TOS, remove TOS from stack
$10	RI_DIV	+6	Divide TOS into NOS, remove TOS from stack
$12	RI_ABS	=	Make TOS positive
$14	RI_NEG	=	Negate TOS
$16	RI_DUP	-6	Copy TOS and create a new TOS above current TOS
$17	??	=	Swap TOS and NOS. Available in Minerva ROMs and SMSQ only.
$18	RI_COS	=	Cosine of TOS
$1A	RI_SIN	=	Sine of TOS
$1C	RI_TAN	=	Tangent of TOS
$1E	RI_COT	=	Cotangent of TOS
$20	RI_ASIN	=	Arcsine of TOS
$22	RI_ACOS	=	ArcCosine of TOS
$24	RI_ATAN	=	ArcTangent of TOS
$26	RI_ACOT	=	ArcCotangent of TOS
$28	RI_SQRT	=	Sqare root of TOS
$2A	RI_LN	=	Natural log of TOS
$2C	RI_LOG10	=	Log base 10 of TOS
$2E	RI_EXP	=	Exponential of TOS
$30	RI_POWFP	+6	Raise NOS to power TOS, remove TOS from stack
$32	RI_PI	-6	Put PI on the stack as the new TOS (SMSQ/E)
$31-$FF			Are the 'save' and 'load' op codes

Table 16.1: Arithmetic Package Operations

bles 16.2, 16.3, 16.4 and 16.5 which detail exactly which input and output registers are required for the `RI_EXEC` and `RI_ECXECB` vector calls.

Register	Usage
D0.B	Op code. The high word of D0 should be zero
D7.L	Should be zero
A1.L	Arithmetic stack pointer (relative to A6)
A4.L	Pointer to variable storage (relative to A6)

Table 16.2: RI_EXEC Entry Registers

Register	Usage
D0	Error code
D1-D3	Preserved
A0	Preserved
A1.L	Updated to new arithmetic stack pointer
A2-A4	Preserved

Table 16.3: RI_EXEC Exit Registers

Register	Usage
D0.B	Not used
D7.L	Should be zero
A1.L	Arithmetic stack pointer (relative to A6)
A3.L	Pointer to list of op codes.(NOT relative to A6)
A4.L	Pointer to variable storage (relative to A6)

Table 16.4: RI_EXECB Entry Registers

You will notice that A4 was never used in my two examples. This is a pointer to the top of an area of memory where you wish to save floating point values to, and load them back from. A4 is relative to A6 (as ever).

The op codes from $31 through $FF can be used to save and load 6 byte floating point values from the stack to and from the variables area.

Op codes that are even allow numbers to be loaded from storage onto the stack creating a new TOS and setting A1 to A1-6.

Op codes that are odd cause the number at TOS to be removed from the stack and saved in the variables area. This causes A1 to change to A1+6. The corresponding load routine is the op code minus 1. (If I call this routine with $33 then the opposite routine is $32 and so on.)

The actual start address of the variables area where your number will be stored is calculated as:

$$A6.L + A4.L + ((D0.B \text{ AND } \$FE) - \$100)$$

or, in another way:

$$((D0.B \text{ AND } \$FE) - \$100)(A6, A4.L)$$

Each load or save operation uses 6 bytes starting at the above address and working UP in memory. This means that you cannot use all of the load/save op codes for the following reason.

Register	Usage
D0	Error code
D1-D3	Preserved
A0	Preserved
A1.L	Updated to new arithmetic stack pointer
A2-A4	Preserved

Table 16.5: RI_EXECB Exit Registers

Assume you want to save two numbers from the stack. You might be tempted (as I was) to assume that you could save the first using $FF and the second using $FD. OK, try it out. Remember saves are odd, loads are even.

Assume also that the absolute address (ie $A6 + A4$) of your variables area is $1000 0000.

So, where do our two values end up at?

For $FF it works out as:

$$\begin{aligned}&\$10000000 + (\$FF \text{ AND } \$FE)\\&= \$10000000 + \$FE\\&= \$100000FE\end{aligned}$$

For $FD it is:

$$\begin{aligned}&\$10000000 + (\$FD \text{ AND } \$FE)\\&= \$10000000 + \$FC\\&= \$100000FC\end{aligned}$$

Because each save uses 6 bytes, the ranges covered are:

- For code $FF, we use the bytes from $1000 00FE to $1000 0103
- For code $FD, we use the bytes from $1000 00FC to $1000 0101

This has two pretty major problems in my opinion. The first is that we have overwritten some bytes above the top of our variables area and the second is that we have managed to overwrite a few bytes of our first saved number with the second one!

The maximum range of bytes available for saving data to and loading it back from is between -208(A6,A4.L) for op code $31 to -2(A6,A4.L) for op code $FF, however, it seems that you are best to use only certain values (see below) to avoid trashing your saved values and avoid using the top two values $FF and $FD for saves and loads or you will partially overwrite other data above your variables area.

I would advise using the save codes as follows:

- $FB (-5) as the absolute minimum value; then
- $F5 (-11)
- $EF (-17)
- $E9 (-23)
- $E3 (-29)
- $DD (-35)
- $D7 (-41)
- ...

And so on subtracting 6 from the op code each time. To load these values back onto the arithmetic stack, use the following codes:

- $FA (-6) as the absolute minimum value; then
- $F4 (-12)
- $EE (-18)
- $E8 (-24)
- $E2 (-30)
- $DC (-36)
- $D6 (-42)
- ...

George has documented the values and offsets to use in saving and loading floating point values on my Wiki at `http://www.qdosmsq.dunbar-it.co.uk/doku.php?id=qdosmsq:vectors:op` where there is an example of saving and reloading floating point variables from the maths stack into a programmer defined variables storage area.

16.3 Coming Up...

Well, just when I thought everything was ok, George Gwilt hammered me silly by email (in the nicest possible way of course) about this current chapter and the previous one.

In the next chapter, George and I have a conversation about what I did wrong or could have done better.

Much of the code above has been changed to match George's comments. Some of his explanations are hinted at in the above, but have not yet been changed. Read on for the full, gory details!

17. Much Ado About Previous Chapters

17.1 Introduction

George Gwilt, my faithful reader, has brought me to task on my last two articles. Part 15 where I wrote (ok, updated a very old 1991 utility which I had written) and again after Part 16 where I delved into the Arithmetic Package in QDOSMSQ.

I shall attempt to answer Georges concerns in this chapter.

17.2 Chapter 15 - Dataspace Utility Problems

George makes a number of interesting points about this article and all I can say is, ‘he is absolutely correct’.

As for my small routine to convert an ASCII string into a number in a long word, George asks why it is not itself a sub-routine when I make such a ‘fuss’ (my word) of reusable code.

I can only plead guilty as charged and state, for the record, that this is the only time I’ve ever written anything in assembly language which required me to do that conversion. To that end, and nothing else, the code was in-lined in 1991 and remained so in 2006.

However, I’m sure a general purpose ASCII->Long could be easily written as a subroutine. I’m certain that there is one lurking somewhere inside QDOSMSQ which correctly (I hope) handles invalid characters, errors, overflow and so on.

I shall be creating just such a beast in the next chapter.

I feel rather unable to comment on George’s own conversion routine - I never did very well at maths at school and I’m not sure exactly how George’s code works (yet!).

17.3 Chapter 16 - Artithmetic Package Problems

Shortly after that article appeared, George contacted me with a whole host of problems. I shall attempt to answer George's concerns below, although George and I have conversed in an email exchange on this subject, I think it is proper to publicise the results especially as they concern my previous chapter. Corrections are due!

George ... I have only one comment on ROOT. It is that, as explained by Dickens (QL Advanced User Guide), you do not need to test D0 for errors after a call to a vector since the condition codes are set on return. Indeed Norman does not make the test after calling the vector `BP_INIT` near the top of page 21. The two later tests on that page can thus be deleted.

Norman I agree, however, it has been my observation in the past that only sometimes are the condition codes actually set on exit from QDOSMSQ. To this end I tend to always test D0 on exit from a QDOSMSQ call - just to be safe. This does mean that where I neglected to do this after the call to `BP_INIT` (on page 21) is where *my* error was. I should have had a test there.

George points out that I don't need the two tests later on that same page. While technically correct, I would be inclined to leave them present and add in the one I missed out rather than removing the latter two. I like to make sure that the condition codes are correctly set by testing them explicitly as this saves me trying to remember which calls do set them and which calls don't.

George ANYROOT is more interesting as it uses `RI_EXECB` to perform a string of operations. Once again the testing of D0 in do_it and do_more is not really needed. Also, I should point out that the three lines of op_codes should not be between got_ok and do_it otherwise do_it will never be done.

Norman This is absolutely correct. I have no idea what went on here, but the code should be as follows[1]:

```
got_ok      cmpi.w   #2,d3            ; One parameter?
            bne.s    bad_param        ; Oops!
            bra.s    do_it            ; skip over the op-codes.
```

Listing 17.1: Corrections to ANYROOT Code in Previous chapter

With a short branch over the op-codes added. I suspect that I have inadvertently fixed the code while running under QPC but forgotten to save the corrected version back to one of my DOS_ drives prior to importing the code into the article. Quite honestly, the original code without the branch would most likely have hung the system.

I have checked my source code system and found that the same 'broken' version is present there too, so it does look like I forgot to save a change back to DOS. 'Mea Culpa' as they say.

George Also I wonder how Norman expects it to work given that the address in A3, set in do_more, is not relative to A6 as he suggests is necessary in his definition of `RI_EXECB` on page 26. (But see later.)

Norman I'm afraid that this was a 'copy and paste' error. I copied the A1 line above it and pasted it in. While I remembered to change the A1 to an A3, I neglected to remove the part about it being relative to A6. That is incorrect as A3.L is the pointer to the string of bytes and is not relative to A6 at all.

George It is annoying that immediately after the first operation the operands are in the wrong order on the stack. Norman has produced swap_tos to switch the order. The code works well, but, since I started my programming life on machines with limited space and slow speed, I always try to

[1] And, if you read it again, it actually is now.

compress and speed up any program. I might suggest here that you eliminate the two occurrences of exg d6,d7 [2]and instead swap the d7 and d6 in the following two lines in both cases.

Norman I started on a ZX-81 with 1KB of RAM and I'm mostly self-taught - hence all the errors!

George suggests changing my code at SWAP_TOS from this:

```
swap_tos    move.l   0(a6,a1.l),d7     ; Get a long word
            move.l   6(a6,a1.l),d6     ; And another
            exg      d6,d7             ; Swap them around
            move.l   d7,0(a6,a1.l)     ; Store
            move.l   d6,6(a6,a1.l)     ; Store
            move.w   4(a6,a1.l),d7
            move.w   10(a6,a1.l),d6
            exg      d6,d7             ; Swap again
            move.w   d7,4(a6,a1.l)
            move.w   d6,10(a6,a1.l)    ; Now we have N and LN(M) swapped
```

Listing 17.2: ANYROOT - Swap_Tos - Original Code

to the following to save a couple of instructions and hence, valuable time and space:

```
swap_tos    move.l   0(a6,a1.l),d7     ; Get a long word
            move.l   6(a6,a1.l),d6     ; And another
            move.l   d6,0(a6,a1.l)     ; Store
            move.l   d7,6(a6,a1.l)     ; Store
            move.w   4(a6,a1.l),d7
            move.w   10(a6,a1.l),d6
            move.w   d6,4(a6,a1.l)
            move.w   d7,10(a6,a1.l)    ; Now we have N and LN(M) swapped
```

Listing 17.3: ANYROOT - Swap_Tos - Original Code

Once again, George is correct - I must have run out of caffeine at that point. The two `EXG` instructions are completely unnecessary when written as above.

George I have, however, a more radical suggestion. It is that you eliminate both do_it and swap_tos and increase the size of op_codes so that the whole procedure is carried out with just one set of operations using `RI_EXECB`. The easy way of doing this is possible if you have SMSQ or Minerva both of which have additional operations one of which swaps TOS and NOS. The code is $17.

Norman I try to keep things as close to the original QL as possible so this option may not have been available to some of my other readers. I was, however, completely unaware of it until George sent me his email. I am completely surprised in finding myself to be the first person since QDOS was originally written to need a SWAP_TOS_NOS routine :o)

When I wrote the code originally, I was almost certain that I could do it an one single `RI_EXECB` operation. That was when I discovered the need for a swap operation and hence the break up into a single RI_EXEC, manual swap and the `RI_EXECB` call. Not as elegant as I would have liked.

George The second method is to go through the business of copying TOS and NOS somewhere and then returning them to NOS and TOS. The codes for this are in the group referred to by Norman as $FF31 to $FFFF. The place I would use for temporary storage is the Basic buffer. First, op_codes would become:

```
op_codes  dc.b        RI_LN,-5,-11,-6,-12,RI_DIV,RI_EXP,RI_END
```

Listing 17.4: ANYROOT - Swap_Tos - Suggested Op Codes

[2]Consider them eliminated.

do_it and swap_tos would be deleted and do_more would have added to it:

```
        movea.l     bv_bfbas(a6),a4
        lea         12(a4),a4          ; point to the end of 12 bytes
```

If you really think that there may not be as much as 12 bytes available in the Basic buffer you can add:

```
        cmpa.l      bv_tkbas(a6),a4    ; A4 beyond end of buffer?
        bhi         bufful             ; oops
```

bv_bfbas is 0 and bv_tkbas is 8.

The codes -5 and -6 save and load 6 bytes to and from -6(A6,A4.L) and the codes -11 and -12 do the same with address -12(A6,A4.L).

Norman Again, when I wrote the article, I was having a few difficulties with the save and load op-codes and was enjoying much discussion on the QL-USERS email list at the time. I avoided them until I could better understand them. As it turned out, the explanations simply confused the matter (for me) and I decided to leave them alone and simply document them at the end of my article.

Interestingly, George raises something that I was always confused by when Simon N Goodwin was writing in the old QL World assembly language series, using the Basic Buffer as a workspace. I'm sure that there could be a couple of pages for an article here on this very subject - if someone who knows it was prepared to write one (hint hint). :o)

George ... However, I do have severe objections to the later part of the article. This mainly relates to the op codes, especially those used for loading and saving numbers onto and from the stack. The table at the top of page 26 lists the op codes. If used in `RI_EXECB` these must fit into bytes. It is thus plain wrong to list the codes other than those from 0 to $30 as $FF31-$FFFF. But why should Norman do this?

Norman To answer the last question, I did it because I was advised by Marcel Kilgus, in an email, that *I differed from the documentation* and that the load and save op-codes were indeed negative words and not bytes. And indeed, I quote:

" ... the opcodes $FF31 to $FFFF are for load/save, not $32 to $FF. Yes, the latter DO work, but it seems that's more an undocumented side-effect."

I took the advice of the man who wrote QPC and probably has forgotten more about Assembly Language programming that I have ever known!

George To find out I examined the definitions of `RI_EXEC/RI_EXECB` in five different publications:

Title	Author	Errors
QL Technical Guide	David Karlin & Tony Tebby	A and B
QL Advanced User Guide	Adrian Dickens	C
The Sinclair QDOS Companion	Andrew Pennell	None
QDOS Reference Manual	Jochen Merz	A

Table 17.1: QDOS Documentation and RI_EXEC/RI_EXECB Errors

Norman

George has given a pretty thorough explanation of the differences between the above 5 sets of documentation and the code in a JS ROM and SMSQ/E - I can only state that I wish I had stuck with Pennell rather than trying to find out more! See table 17.1.

I now skip directly to George's closing points.

George ... Having explained how the operations codes are used by `RI_EXEC` and `RI_EXECB` I return to Norman's article on page 26[3].

1. In the description of the OpCodes he lists $FF31 to $FFFF as the 'save' and 'load' bytes. Clearly these should be $31 to $FF (or 49 to 255).

Norman This is correct. Obviously, had I been paying more attention in class, I would have questioned Marcel's *word* information as the `RI_EXECB` call executes a string of *bytes*. Setting a *word* in amongst the bytes would have resulted in an $FF op-code being carried out followed by a separate and incorrect byte code, rather than a store or load operation.

Basically, when I say 'a negative word' I really mean 'a negative byte'.

George 2. In the definition of `RI_EXEC` it is bits 8 to 15 of D0 which should be zero and not the high word, which can in fact be anything.

Norman Correct. `RI_EXEC` expects a byte sized op-code.

George 3. In the definition of `RI_EXECB` A3.L is the absolute pointer to the list of op codes. It is not relative to A6 as stated. (So ANYROOT will work after all.)

Norman Yes, as admitted above, this was a type on my part. A3.L is an non- relative address.

George 4.1 The byte op codes from $33 to $FF can be used to save and load numbers. The effect of codes $31 and $32 depend on the operating system. JS ROM and SMSQ differ here. Both operating systems contain oddities. In the JS ROM $ 31 will be treated as a save to -208(A4,A6.L). However, $30 will be treated as the operation NOSˆTOS so that you can't bring back the saved item!

SMSQE has a different, though similar quirk. The code $31 gives a "not implemented" error. Code $32 puts PI on the stack. Code $33 saves the number on the stack to -206(A4,A6.L). As with the JS ROM this number cannot be reloaded, since the code to do so just puts PI on the stack!

4.2 Norman mentions calling "this routine with $FF33". This is of course impossible with `RI_EXECB`, the op code is just $33. You can call `RI_EXEC` with D0.W equal to $FF33, or $1C33, or $0033 and each will have the same effect. He then says that the actual address used for storage is:

$$A6.L + A4.L + (D0.W \text{ AND } \$FFFE)$$

Again, I'm afraid this is not quite true. The address is actually:

$$((opcode \text{ AND } 254) - 256)(A4, A6.L)$$

Norman I agree with George's point 4.1. As for 4.2, the address calculation I gave is the one I found in Jochen's QDOS Documentation.

George 4.3 Pennell is not wrong in the way Norman suggests in his first WARNING on page 28. First, as I have tried to explain, the op codes are really and truly byte sized numbers (2 to 255). Second, Pennell gives the range for loading (not saving) a number. His range of offsets, -206 to

[3] In the original QL Today publication.

-2 is absolutely correct for QDOS. It is when Pennell says that the op codes with odd values $31 to $FF give the same range he is wrong. That range is -208 to -2, but the value -208 is effectively useless. I don't think this very minor error will bother anyone and it certainly does not warrant a WARNING.

Norman I'm not doing very well am I? Once again, I stand corrected.

George 4.4 Norman suggests that in fact you can use a byte for the codes $31 to $FF when using `RI_EXEC`, but that it is an undocumented feature. This is not true since Pennell (page 133) does document it.

Norman I blame Marcel, it's all his fault. (Only kidding.) As I mentioned above, I was basing my article on the latest information that I had been given as a result of asking for clarification on the QL-USERS mailing list.

George 4.5 I would like to add real WARNING. It is that you can crash the program by using an odd op code below 50 in QDOS. This is because the code is used directly as an index into the programs performing the operations.

Norman This is indeed true.

George I hope Norman will forgive me for attempting to set so many things straight. The errors are not wholly his fault!

Norman Phew, I'm glad that's over. I took a severe beating at the hands of George and I promise to do better in future!

Seriously, I'm always happy to be corrected in anything I say or write - so, if you spot anything that you disagree with, let me know.

And as for forgiveness, I have no problems there either.

17.4 Coming Up...

So, that's a slightly different article this time. I hope to be back in deepest, darkest code again next time, especially as I have promised to provide a useful ASCII to long word conversion routine. I think I know just where I can find one.....

18. Ascii To Long Converter

18.1 Introduction

In the last exciting instalment of the series, I mentioned that I would be looking into the bowels of QDOSMSQ to see if I can find a useful sub-routine to convert a string of ASCII characters into a long value in a register. This was suggested by comments from George Gwilt when he mentioned that he was surprised that I didn't have a reusable routine to do this conversion. This chapter is a result of that enquiry.

18.2 How QDOSMSQ Does It

As ever, I like to take the lazy approach to writing code. If someone else has done it for me, that's a bonus. Inside QDOSMSQ there is a vectored routine called `CN_DTOF` which reads a string of characters and converts those to a floating point value on the maths stack. This routine can be entered with D7.L holding the address of the first byte of memory *after* the final character of the string, or with D7 set to zero.

In the latter case, the `CN_DTOF` routine simply keeps reading until it comes across any character which is not a valid digit, decimal point or 'e' in the buffer. In the former case, the routine stops when it reaches the address in register D7.L or if it hits an invalid character before then.

On exit, the buffer pointer is pointing at the character after the buffer or at the invalid character, unless an error occurred, in which case A0.L and A1.L are restored to their values on entry.

So far so good, we have a floating point value on the maths stack at 0(A6,A1.L) but we wanted a long value from our routine. This too is easy. Thinking back to the article on using the arithmetic package, we can use the `RI_NLINT` operation to convert a floating point value down to a long word. Once this is done, it is a simple job to copy it off the maths stack into our data register and we are done.

All conversion 'problems' for the character data have been dealt with by QDOSMSQ as have problems of overflow and so on when we convert from FP to LONG. How easy can it get?

18.3 Rules And Regulations

Obviously, we might have problems. Isn't the maths stack provided for use by SuperBasic routines only? Well, the code in this article shows that this is not the case, provided a couple of simple rules are followed.

- Rule number one is that A1.L has to point at the byte just above the top of the maths stack - at the highest address in other words.
- Rule number two is that you must have enough space on the maths stack for the operation(s) to be carried out. It is possible that some routines will need working space on the maths stack. This must be catered for or you may find that the maths operations corrupt data below your maths stack.

Note According to Dickens, the `CN_DTOF` vector uses about 30 bytes of space on the maths stack. So, for this conversion routine to work, you should set up a maths stack with *at least* 30 bytes - although it wouldn't break the system to use a bit more for safety. I'm using 15 long words, which should be ample.

The maths stack, while looking special, has to be considered for what it is, it is just a chunk of memory somewhere in the system, relative to A6 of course.

18.4 The Code

The following is our conversion routine in all it's glory. As you can see, there is not much to it.

```
*-------------------------------------------------------------------------------
; Useful routine to convert an ASCII string to a LONG word.
;
; Entry Registers:
;
; A0.L - Pointer to first character in buffer (not the size word).
; A1.L - Pointer to an area of AT LEAST 30 bytes for a maths stack.
;
; Exit Registers:
;
; D0.L - Error code, or zero if no errors. (Z flag set for no errors).
; D1.L - Value of converted ASCII string.
; A0.L - Updated pointer. First char after all valid numerics (and 'e')
;        or first character after end of input in nothing was invalid.
; Rest preserved
;
; Error Exit Registers:
;
; D0.L - Error code, or zero if no errors. (Z flag set for no errors).
; D1.L - unknown.
; A0 - Preserved = pointer of start of buffer on entry.
; Rest preserver.
*-------------------------------------------------------------------------------
ri_nlint equ      6              ; Code to convert FP to LONG

```

```
convert  movem.l   d2/d7/a1-a3,-(a7)   ; Save workers
         suba.l    a6,a0               ; Relativise buffer address
         suba.l    a6,a1               ; And the maths stack
         moveq     #0,d0               ; Assume no errors
         moveq     #0,d1               ; Zero result
         moveq     #0,d7               ; For CN_DTOF
         move.w    cn_dtof,a2          ; Convert ASCII to an FP number
         jsr       (a2)                ; Do conversion
         tst.l     d0                  ; OK?
         bne.s     restore             ; No, bale out.
```

Listing 18.1: ASCII to LONG Converter - Part 1

The entry point to our routine is at the 'convert' label above. We start off by saving all the registers that we are going to use, or that will be trashed by the various QDOSMSQ code.

Once that has been done, we subtract the current value of A6 from the two pointer registers as these addresses have to be A6 relative for the maths package code to work.

Next, and the most complicated part of the code is to convert our buffer load of characters into a floating point number on the maths stack. If there were conversion errors then we abandon ship and bale out.

Conversion errors occur when there are illegal characters in the buffer - more than one decimal point, two or more 'e' characters etc. Note however, that conversion will stop when a non-valid (but non-error causing) character is found. So '1024K' will result in the value 1024 being created and then conversion would stop.

```
*------------------------------------------------------------------------------
; We now have a floating point value on the maths stack at 0(a6,a1.l).
; Convert that down to a long word.
*------------------------------------------------------------------------------

         moveq     #ri_nlint,d0        ; FP to LONG
         moveq     #0,d7               ; For maths package
         move.w    ri_exec,a2          ; Execute one maths operation
         jsr       (a2)                ; Do it.
         tst.l     d0                  ; OK?
         bne.s     restore             ; No, bale out
```

Listing 18.2: ASCII to LONG Converter - Part 2

The second part of the code above, is where we convert the floating point value on the maths stack into a long integer. This uses the afore mentioned maths package to do the conversion. Any errors such as overflow will be trapped and returned in D0. We test for this on return from the `RI_EXEC` and if we have a problem in conversion, we bale out.

```
*------------------------------------------------------------------------------
; We now have a long word on the maths stack 0(a6,a1.l).
*------------------------------------------------------------------------------

         move.l    (a6,a1.l),d1        ; This is our value
         adda.l    #4,a1               ; Tidy maths stack pointer

restore  movem.l   (a7)+,d2/d7/a1-a3   ; Restore workers
         adda.l    a6,a0               ; Unrelative the buffer again
         tst.l     d0                  ; Set flags
```

```
            rts
```

Listing 18.3: ASCII to LONG Converter - Part 3

The above simply copies the long word from the maths stack into the D1 register ready to return it to the caller, tidies up the stack and restores the working registers. We exit with the Z flag set if no errors occurred and unset otherwise.

On exit, the address in A0.L points at the first character after the string of digits that were converted - in an input buffer, for example, this would be the linefeed.

The QDOSMSQ routines to convert the ASCII into an FP number have 'interesting' register settings on exit. If no errors occurred then we exit with A0.L set to point at the character after the end of the buffer, or, at the invalid character that caused conversion to end. If there was a conversion error, the value in A0.L is reset to that on entry - the pointer to the first character in the buffer.

My code exits with the registers set as described in the code header above.

As a quick example of testing the above, and just to prove that it does work, here is a small test harness. Save the following as a new file named 'test_asm'.

```
test    bra.s   test2

result  ds.l    1               ; One long word for the result
        ds.b    1               ; One byte for the terminator

fp      dc.b    '1234567.89x'   ; The fp number in Ascii plus
*                               ; an invalid character
        dc.l    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0   ; 15 Long words
msp     equ     *               ; STACK TOP for the maths stack

test2   lea     fp,a0           ; Buffer holding Ascii
        lea     msp,a1          ; Top of maths stack
        bsr.s   convert         ; Convert from ascii to long
        lea     result,a1
        move.l  d1,(a1)+        ; Save result
        move.b  (a0),(a1)       ; Terminator
        rts

        in win1_source_convert_asm   ; Load in the utility code
```

Listing 18.4: ASCII to LONG Converter - Test Harness

Save the file and assemble it. To test it all out, the following is all that is required:

```
ADDR = alchp(1024)
LBYTES win1_source_test_bin,addr
CALL ADDR
PRINT 'Result = '; PEEK_L(ADDR+2)
PRINT 'Terminator = '; CHR$(PEEK(ADDR+6))
```

Which in my case gives me a nice long value of 1234568 for Result and a terminator of 'x'. In the event of an illegal FP number being converted, say one with two decimal points or two 'e' characters or whatever, an invalid number error will result. If the FP value cannot be converted to a LONG without overflowing, an overflow error will result.

So, there is is, a small piece of code (around 156 bytes in my 'test_bin' file) to convert a string of ASCII characters into a LONG word. How easy was that then?

18.5 Code Improvements

In the code above, the 'convert' routine assumes that a buffer, pointed to by A0.L, holds a string of ASCII characters *without* a leading QDOS string's length word. Unfortunately, most of QDOS relies on there being a length word at the start, so we really should allow for this in the convert code as well.

Well, I've been thinking (a rare thing for me - ask my wife!) and I realised that, internally, QDOSMSQ allows D7.L to be zero or the address of the first byte in memory AFTER the last character of the ASCII to be converted to a floating point value. We can use this in our favour. The conversion stops when the address in D7 is reached as QDOSMSQ loops around converting each character from the buffer.

With a slight modification the the code, we can cater for both formats of buffers - one without a leading size, and one with. The changes required are simple.

Add the following code just before the code at convert, line 26 in my source file:

```
26 convertq move.w   (a0)+,d7          ; Get the length word
27          ext.l    d7                ; Sign extend to a long word
28          add.l    a0,d7             ; D7.L correctly set, A0 also.
```

Listing 18.5: Better ASCII to LONG Converter - Converq

Then, remove the following line from near the start of the convert code, it's just above the call to `CN_DTOF` which is at line 31 in my file:

```
31          moveq    #0,d7             ; For CN_DTOF
```

So, your codefile should now look like this:

```
26 convertq move.w   (a0)+,d7          ; Get the length word
27          ext.l    d7                ; Sign extend to a long word
28          add.l    a0,d7             ; D7.L correctly set, A0 also.
29
30 convert  movem.l  d2/d7/a1-a3,-(a7)   ; Save workers
31          suba.l   a6,a0             ; Relativise buffer address
32          suba.l   a6,a1             ; And the maths stack
33          moveq    #0,d0             ; Assume no errors
34          moveq    #0,d1             ; Zero result
35          move.w   cn_dtof,a2        ; Convert ASCII to an FP number
36          jsr      (a2)              ; Do conversion
37          tst.l    d0                ; OK?
38          bne.s    restore           ; No, bale out.
```

Listing 18.6: Better ASCII to LONG Converter - Part 1

And that's all there is to it. You can now call the 'convert' code with A0.L pointing at a buffer of ASCII characters and no QDOS length word as long as the buffer has an 'invalid' digit at the end, a linefeed perhaps, or, you can point A0.L at a proper QDOSMSQ string's length word and call the code at 'convertq' instead - with D7 set to zero first of course.

A small test harness for the new version would be as follows:

```
test        bra.s       test2

result      ds.l        1                   ; One long word for the result
            ds.b        1                   ; One byte for the terminator

fp          dc.w         10                 ; How long is the text?
            dc.b        '1234567.89'        ; The fp number in Ascii plus
*                                           ; an invalid character
            dc.l        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0     ; 15 Long words
msp         equ         *                   ; STACK TOP for the maths stack

test2       lea         fp,a0               ; Buffer holding Ascii
            lea         msp,a1              ; Top of maths stack
            bsr.s       convertq            ; Convert from ascii to long
            lea         result,a1
            move.l      d1,(a1)+            ; Save result
            move.b      (a0),(a1)           ; Terminator
            rts

            in win1_source_convert_asm    ; Load in the utility code
```

Listing 18.7: Better ASCII to LONG Converter - Test Harness

The above is remarkably similar to the test harness I provided above. The only difference is that the ASCII buffer at label 'fp' has been converted to a properly formatted QDOSMSQ string with a leading length word added and the 'x' has been removed from the end of the original ASCII buffer.

Note the call to 'convertq' rather than 'convert'.

18.6 Coming Up...

Now, just this week[1] I have sold my house and so my wife Alison and I are in the process of looking for a new home. This means that I might not have email etc for much longer so I cannot guarantee whether I shall be writing in the next issue or not. Hopefully I will be, but just in case, I apologise in advance for my absence!

See you soon for more exciting code!

[1] Ahem, that was written 8 years ago, in what must have been 2007. We are still in the 'new' house though - we haven't moved again, yet.....

19. Assorted Revisions And Ramblings!

19.1 Introduction

Greetings from the basement!

We have moved house and are getting settled.[1] We have still got a lot of boxes to unpack and things to find, but we are getting there. I have a new ‘office’ deep down in the basement where it is nice and cool. This is the first in the Assembler series to come from the basement.

With all the upheaval of getting moved and unpacked etc, I have not got a lot of code for you this time, hopefully, you won’t be too bored by this episode in which I go over bits and pieces of assembly language programming that causes me grief.

It all started when I was having a think the other day about life in general and assembly language in particular. I was pondering on the bits of programming in assembler that I always get wrong, or have to really think about - and still get wrong.

19.2 SIGNED And UNSIGNED Tests

I don’t know about you, but I seem to have severe difficulties in remembering which are the signed and which are the unsigned tests. I have to confess that I always have a list of them written down (or printed out) and stuck to my work area - wherever that happens to be.

Table 19.1 is a reminder of the ‘cc’ code to use in a `Bcc` or whatever for signed and unsigned comparisons:

So, if D0.B contains the value $FF it represents either 255 (unsigned) or -1 (signed). You, as the programmer should know whether the value is considered signed or not and can make the correct comparison checks.

The EQ and NE tests are interesting in that they either mean ‘two values are [not] the same’ when

[1] That was written originally in 2007. Treat with a pinch of salt now!

Desired Test	Signed	Unsigned
Greater Equal	GE	CC
Greater Than	GT	HI
Equal/Zero	EQ	EQ
Not Equal/Zero	NE	NE
Less Equal	LE	LS
Less Than	LT	CS
Negative	MI	n/a
Positive	PL	n/a

Table 19.1: Signed and Unsigned Tests

comparing things such as memory and registers, or two registers etc, or, when having just loaded a register with a value, they mean 'the value just loaded into a data register is [not] zero'.

The following code examples are identical in result, one is just quicker than the other:

```
    MOVE.W (A1),D0
    BEQ.S  D0Zero
     ....
```

and

```
    MOVE.W (A1),D0
    CMPI.W #0,D0
    BEQ.S  D0Zero
     ...
```

19.3 Which Way Round Is The 'Subtraction' In CMP?

If I see CMPI.W #1234,D0 then it is obvious, I am comparing D0.W with the value 1234. That's easy. However, when I see CMP.W D0,D1 I lose the plot.

What am I comparing here is it D0 with D1 or the other way around. My brain hurts already.

Is the value of 1234 subtracted from D0 or is the value in D0 subtracted from 1234. Which way round is the subtraction and the resulting setting of flags?

The answer, I note from part 2 of this series is that the source register is subtracted from the destination register exactly as a `SUB` instruction would do, the result is simply discarded. So in the instruction CMP.W D0,D1 the flags are set according to D1.W minus D0.W.

It is assumed that after this pseudo-subtraction, some `Bcc`, `Scc` or `DBcc` instruction will no doubt check the flags and do something useful with the result.

19.4 Which CC Code To Use After CMP

Leading on from the above, I never remember which 'cc' code to use after a `CMP` - although, having written out the above it is becoming clearer. The following code gives me the willies time after time:

```
    ...
    CMP.L  D0,D1
    BHI     somewhere
    ...
```

This fragment has everything that confuses me, almost. It has a `CMP` followed by a 'cc' instruction - so I have to think about the two 'problem areas' I mention above. Signed or unsigned and which register is causing the HI to be true or false.

Well, the HI is, from my table above, unsigned and using my new found knowledge of the `CMP` instruction I know (for a short while at least) that the flag are set to the result of (D1.L - D0.L) but which way around does it go again?

The `BHI` should be read as 'branch if destination register HI source register' in the preceding `CMP` or `SUB` or whatever was used to set the flags. So, using this explanation, I now know that the code above branches if D1.L is higher (in an unsigned manner) than D0.L.

This leads me to surmise that the following pseudo-code:

```
IF unsigned (D1.L > D0.L) THEN
...
ELSE
...
END IF
```

Becomes:

```
IF      CMP.L D0,D1      Flags = result of D1.L - D0.L
        BHI.s THEN      D1 is indeed (unsigned) greater than than D0
ELSE    ...             D0 is less or equal to D1 (the ELSE bit)
        BRA.s ENDIF      Skip over the THEN clause
THEN    ...             Do the THEN stuff
ENDIF   ...              Together again.
```

Alternatively, reverse the jumps to look more like the pseudo code:

```
IF      CMP.L D0,D1      Flags = result of (D1.L - D0.L)
        BLS.s ELSE      D1 is not HI (unsigned) than D0
THEN    ...             D1 is indeed HI than D0, (the THEN bit)
        BRA.s ENDIF      Skip over the ELSE clause
ELSE    ...             Do the ELSE stuff
ENDIF   ...              Together again.
```

Maybe we should think about writing our assembly language in pseudo code and having a pre-processor convert it into the real assembler code ....

19.5 Loops With Conditions

The instruction format for decrement and branch on condition instructions is `DBcc` where 'cc' is one of the many condition codes noted above.

So, you have an area of RAM full of data and you go looking through it for the first occurrence of a specific byte value, let's say $00, and you know that the leading word of the data defines the length in bytes. So, the following fragment would do the job - assuming A0.L points to the data and D1.W holds a valid data length.

```
LOOP      CMP.B #$00,(A0)+
          DBcc D1,LOOP
ENDLOOP ...
```

What we need to figure out is which 'cc' we require and also, what is result when we get to the ENDLOOP label if we found a zero byte or if we didn't.

One way we will end up at ENDLOOP is when our counter in D1 expires - reaches minus 1 - that indicates that we ran out of data before finding what we wanted. But, what happens if we find a zero byte - and which 'cc' do we need.

If we remember that `DBcc` really means 'test condition and decrement if false and branch' then we should be ok. Alternatively:

```
    IF 'cc' is FALSE THEN
        D1 = D1 - 1
        IF D1 <> -1 Then
          GOTO LOOP
        ELSE
          GOTO ENDLOOP
        END IF
    ELSE
        GOTO ENDLOOP
    END IF
```

So, we want to check for a zero byte, we can use the 'EQ' test - remember EQ means we have hit a zero or two values are not equal - and our code now becomes:

```
LOOP      CMP.B #$00,(A0)+
          DBEQ D1,LOOP
ENDLOOP ...
```

So, we have reached ENDLOOP and we need to know if we hit a zero byte or if we ran out of data. How to tell?

Well the good news is that the `DBcc` instructions do not alter the flags. So on exit from a `DBcc` loop, if the 'cc' is still true, then the condition was met and the loop terminated before the counter ran out. All we have to do is retest with the same condition as follows:

```
LOOP      CMP.B #$00,(A0)+
          DBEQ D1,LOOP
ENDLOOP BEQ.S FoundZeroByte
```

In this case, we check for the EQ condition which tells us that the loop terminated early. We can test the inverse condition as well to see if the loop expired without hitting the required condition:

```
LOOP      CMP.B  #$00,(A0)+
          DBEQ  D1,LOOP
ENDLOOP BNE.S  NotFound
```

Which we see makes the branch if the loop expired when the counter in D1.W hit minus 1. I propose that we rename this family of instructions to 'Decrement and Branch UNLESS condition'. That makes more sense to me.

19.6 Do I TST.L D0 After TRAPs And Vectors?

I always get corrected on this one, either by George or Simon. For years I have always done this:

```
    ...
    TRAP  #1
    TST.L  D0
    BNE  HandleError
    ...
```

Which is fine for a TRAP call - it *has* to be done this way. However, for a vector call it is different:

```
    ...
    MOVE.W  UT_GTSTR,A2
    JSR  (A2)
    TST.L  D0
    BNE  HandleEror
    ...
```

This is *wrong* - I do not need to test D0 after a *vectored* utility call. The reason I do after a TRAP and don't after a vector is quite subtle and was only recently pointed out to me by Simon when it all became very clear indeed.

A TRAP call is treated as an exception and to return from an exception handler, you use the RTE instruction. To return from a vectored call, it is an RTS instruction. The difference between the two is that the RTE restores the status register as well as the program counter. RTS simply restores the program counter.

So, all these years where I've been testing D0 on return from vectors I've been wasting clock cycles when I need not have done. The status register is correctly set on exit from a vectored utility but has only D0 is set on return from a TRAP.

Simple, but it has caught me out for years. I now need to unlearn my habit of coding a TST.L D0 every time I use a vectored utility.

Happy coding.

19.7 Coming Up...

After many years of studious avoidance, I'm biting the bullet and attempting to learn to code programs under the Pointer Environment.

VI The Pointer Environment - Introduction

20. The Pointer Environment

20.1 Introduction

Well for many years now I've been avoiding this moment, but it has finally arrived. I am starting to learn all about programming the PE in assembly language. You are coming along for the ride! I'm sure that along the way I'll be making as many, if not more, errors than I usually do and someone out there who knows far more than me, will correct me as we go along. As I always say, you should learn from your mistakes![1]

20.2 The Pointer Environment

There are two parts to the PE, PTR_GEN and WMAN - the Pointer environment and the Window Manager. To slip easily into programming, we shall start off with a small and perfectly useless pointer only program, so the Window Manager stuff will not be used.

In order to run the program you should have loaded. If you are running SMSQ/E then it is built in to the operating system. If you are running a 'normal' QL then you will need to load it. I suspect most - if not all - users still with us will have a copy, however, if not, Dilwyn's Web Site will have a copy to download.

Ok, lets dive straight in with our next to useless program, beginning with a host of equates.

```
me          equ     -1          ; The current job id
timeout     equ     -1          ; Infinite timeout
openOld     equ     0           ; Open Old Exclusive device
iop_pinf    equ     $70         ; Get PE information
iop_outl    equ     $7a         ; Outline a primary window
iop_rptr    equ     $71         ; Read the pointer
```

[1] Or mine!

```
termVec      equ     $01                  ; When to return from IOP_RPTR
```

Listing 20.1: Simple PE Program - Part 1

The final equate above tells the call to IOP_RPTR when to return back to the code in our program. It is 8 bits long and each bit has special meaning, as follows:

Bit	Description
0	Return when a key or button is pressed in the window. Also, request window resize
1	Return when a key or button is pressed (subject to auto repeat). Also request window move
2	Return when a key or button is released in the window
3	Return when the pointer moves from the given co-ordinates in the window
4	Return when the pointer moves out of the window
5	Return when the pointer is inside the window
6	Pointer hit the window edge
7	Window Request

Table 20.1: The termination vector

Bits 0 and 1 look a bit funny and are used in conjunction with bit 7 to indicate a window request. If bit 7 is set then the rest should be zero except bit 0 or 1 - one of these should be set. If bit 0 is set the pointer is the resize one and if bit 1 is set it is the window move pointer. If both are unset, the Window Required pointer will be displayed. With bit 7, the topmost window of the pile if hit and selected. More on this later in the series.

Our termination vector is set to have bit zero turned on only. This means when a mouse button is pressed or any key is pressed while the pointer is in the window we create later, the call to IOP_RPTR will return.

As this is a job, we need a standard job header and as we have seen this so many times before, I shall not insult your intelligence by explaining it yet again!

```
             bra.s   start
             dc.l    0
             dc.w    $4afb
             dc.w    15
             dc.b    'PTR_GEN Test v1'
```

Listing 20.2: Simple PE Program - Part 2

Next we have a few definitions of channel names, window sizes and space for the Pointer Record that we get back from a call to IOP_RPTR. First of all, a channel to a console is required.

```
conChan      dc.w    4
             dc.b    'con_'
```

Listing 20.3: Simple PE Program - Part 3

Once we have it open, we redefine it to be 200 wide, 100 deep and centred in the middle of a 512 by 256 'standard' window using the following block of 4 words.

```
conDef       dc.w    200,100,156,78
```

Listing 20.4: Simple PE Program - Part 4

The next 24 bytes are used by the IOP_RPTR call and it stores the pointer record. There is more on this later on in the text.

```
ptrRec      ds.w      12
```

Listing 20.5: Simple PE Program - Part 5

This is where we start the main code. It's pretty simple as you can see and all we basically do here is open a console device, redefine it as mentioned above, set a border of one pixel in red and finally clear the screen (to black paper by default) If anything gives an error we simply kill the job and exit.

```
start       moveq     #IO_OPEN,d0         ; Open a channel
            moveq     #me,d1              ; Channel required for this job
            moveq     #openOld,d3         ; Device exists
            lea       conChan,a0          ; Define device to open
            trap      #2                  ; Do it
            tst.l     d0                  ; Ok?
            bne.s     exit                ; No, exit

            moveq     #sd_wdef,d0         ; Redefine open window
            moveq     #2,d1               ; Border colour is red
            moveq     #1,d2               ; Border width is one pixel
            moveq     #timeout,d3         ; Infinite timeout
            lea       conDef,a1           ; Console definition block
            trap      #3                  ; Do it
            tst.l     d0                  ; Ok?
            bne.s     exit                ; NO, exit

            moveq     #sd_clear,d0        ; CLS
            moveq     #timeout,d3         ; Infinite timeout
            trap      #3                  ; Do it
            tst.l     d0                  ; Ok
            bne.s     exit                ; No, exit
```

Listing 20.6: Simple PE Program - Part 6

That's about all the main setting up that we have to do. We now have a channel open redefined and a nice border showing. The next stage is to look for the PE, if it isn't found, we have a problem and simply exit.

```
FindPE      moveq     #iop_pinf,d0        ; Get PE Information
            moveq     #timeout,d3         ; Infinite timeout
            trap      #3                  ; Do it
            tst.l     d0                  ; Ok?
            bne.s     exit                ; No, exit
```

Listing 20.7: Simple PE Program - Part 7

So far so good. If the PE exists, we now need to make sure that our window is outlined. This indicates to the PE that the window is to be 'managed' It also defines the limits of the 'hit area' where a hit or do or keypress will be registered by our program. This gets a better explanation later in the series.

```
GotPE       moveq     #iop_outl,d0        ; OUTLN our window
            move.l    #$00020002,d1       ; Shadow size of 2
            moveq     #0,d2               ; Don't preserve window contents
            moveq     #timeout,d3         ; Infinite timeout
            lea       conDef,a1           ; Window size without shadow
```

```
                trap    #3,                     ; Do it
                tst.l   d0                      ; Ok?
                bne.s   exit                    ; No, exit
```

Listing 20.8: Simple PE Program - Part 8

The shadow size is added to the sized defined in our console definition block but the shadow is outside of the hit area for our window. Now we read the pointer. The call to `IOP_RPTR` will not return unless the timeout expires or an event happens that has been set in the termination vector to cause a return. We are looking for a button or keypress while the pointer is inside out window.

```
Pointer         moveq   #iop_rptr,d0            ; Read the Pointer
                moveq   #0,d1                   ; Set pointer co-ordinates 0,0
                moveq   #termVec,d2             ; Return on button or keypress
                moveq   #timeout,d3             ; Infinite timeout
                lea     ptrRec,a1               ; Storage for pointer record
                trap    #3                      ; Do it
                moveq   #0,d0                   ; We ignore CHANNEL NOT OPEN
errors
```

Listing 20.9: Simple PE Program - Part 9

When the user click in the window with the mouse, either button will do, or presses a key, the call to `IOP_RPTR` will return having filled the pointer record with useful information. We are not bothering with that here in this first simple demo. We are also ignoring the possibility of A0 pointing at a channel that is closed because by the time we get here, we have carried out lots of actions on it - so it should still be open!

The next part of the code simply kills off our job reclaiming any resources it allocated and closing channels etc.

```
exit            move.l  d0,d3                   ; Save any error codes
                moveq   #MT_FRJOB,d0            ; Kill a job
                moveq   #timeout,d1             ; The job to kill is this one
                trap    #1                      ; Kill me
                bra.s   exit                    ; We never get here...
```

Listing 20.10: Simple PE Program - Part 10

That is all there is to this small demonstration. When assembled and run, you should see a 200 by 150 window centered on screen (well on a standard QL screen anyway) cleared to show black paper with a single pixel red border and a shadow down the right and bottom borders. It is waiting for you to move the pointer into. When you do it will change to an arrow and will remain as an arrow while you move it around. Click a button or press a key while the pointer is in our window and the job will kill itself.

So there we are, a very simple pointer Environment program. More next time as we extend our programming knowledge of the PE. See you then.

20.3 Coming Up...

That's it then, a very quick and useless introduction to the Pointer Environment code. Next time, we shell delve a little deeper and investigate the Pointer Record to see what it can tell us.

21. The Pointer Record Investigated

21.1 Introduction and Corrections

George Gwilt has made a couple of suggestions for improving the code in the previous chpater. In summary:

- All the calls to the `TRAP #3` and checking the error return can be extracted to a small subroutine and called as required.
- The timeout value in D3 is actually preserved through all the `TRAP #3` calls and so need not be implicitly set after the call to `SD_WDEF`.

Both of these improvements have been incorporated into *this* article's code.

In addition to what George spotted, I have one of my own to add. The code at Exit (line 60) reads as follows:

```
60 exit        move.l   d0,d3             ; Save any error codes
61             moveq    #MT_FRJOB,d0      ; Kill a job
62             moveq    #timeout,d1       ; The job to kill is this one
63             trap     #1                ; Kill me
64             bra.s    exit              ; We never get here...
65 case
```

Listing 21.1: Simple PE Program - Part 10 Original

This is slightly incorrect as line 62, which moves the *timeout* value into D1 should read:

```
62             moveq    #me,d1            ; The job to kill is this one
```

Listing 21.2: Correction to line 62

The reason it works is simple, the equates for timeout and me are both -1, so on this occasion, I got away with it!

Having got the errors out of the way, let us progress.

21.2 The Pointer Record

I mentioned in the previous (short) chapter on the Pointer Environment that the pointer record needs a bit of discussion and to this end, I've written a small pointer record diagnostic program that allows you to see what happens when you press a key and so on in a call to `IOP_RPTR`. The code will be shown later in this article. Note however, that it doesn't include any sub-windows yet - those are a feature of a later article.

When you make a call to `IOP_RPTR` you have to have A1 pointing at a 24 byte buffer, aligned on an even address, where the call will write information about things that happened, and where, during the call.

The pointer record looks like Table 21.1.

Offset	Size	Description
$00	Long	Channel ID
$04	Word	Sub window number (-1 = main window)
$06	Word	X coordinate
$08	Word	Y coordinate
$0A	Byte	Keystroke or button (0 = none)
$0B	Byte	Key down or button down (0 = none)
$0C	Long	Event vector LSB only used
$10	Word	Window or sub-window width
$12	Word	Window or sub-window height
$14	Word	Window or sub-window X co-ordinate
$16	Word	Window or sub-window Y co-ordinate

Table 21.1: The pointer record

Now, remembering back to the termination vector in the last article, you will remember that this tells `IOP_RPTR` when to return, so the data in the pointer record depends to a certain extent on what you set in the termination vector. In our first pointer environment example, we simply set bit 0 so we would return from the call to `IOP_RPTR` when a button on the mouse was pressed or a key on the keyboard (where else) was pressed.

What are all those fields in the pointer record used for?

The channel id is simply the channel ID of the window enclosing the pointer. This will not be a sub-window because sub-windows don't have an Id, they are 'simply' sections of the main window. There will be more of sub-windows in a future chapter.

If the window is indeed adorned with sub-windows, the second field holds a word sized sub- window number. This can be used to index into the sub-window table to fetch back the dimensions and so on of the sub-window in question. If this value is $FFFF (minus 1) then the pointer was not in any sub-windows but in the main window.

The X and Y coordinates are those of the pointer position within either the main window or the sub-window. The values are in pixels and both are word sized values.

The next two fields denote which key or mouse button was pressed (and released) or is being held down. For most values this corresponds to the ASCII value of the character code so the ESC key would be $1b or 27 (decimal) however, certain keys have different values:

- a HIT with the space bar gives a code of $01

- a DO with ENTER gives $02 for example.

You will see this as we experiment later with our code for this article. A zero in these fields says that no key or mouse button was pressed/held.

Next we have the event vector which is a long word in size. Only the lowest byte is used (at offset $0F). This appears to be a bitmap of certain operations that have taken place, one or more may have caused the termination of the `IOP_RPTR` call.

Ok, the documentation says that only the lowest byte is used, but the documentation is very old. Things have moved on and it is possible for jobs to be sent an event, rather than generating one themselves, so it is possible that you will see data in bytes other than the lowest one.

Finally, we have 4 words defining the width, height, x and y positions of the window or sub-window in which the pointer event took place. You do not - some might say unfortunately - get the border colour and width or paper and ink colours from the pointer record.

So, now you have details of what the PE documentation has to say about the pointer record, what else can we find out about it ourselves? To answer this question and to see exactly what is stored in it after a call to `IOP_RPTR`, I have written the following almost useful utility to allow us to view the contents of the pointer record after an event has occurred.

I have deliberately kept it simple - as I don't want to clutter up the code with unnecessary adornments - this is not a Windows program after all!:-)

You may notice that it is very similar to our very first introduction to PTR_GEN programming as per the last article.

As ever, we start with a number of equates. None of these need any explanation, so I won't! You can experiment with the value of TermVec as described in the previous article - if you wish.

```
Me          equ     -1          ; Current job id
Timeout     equ     -1          ; Infinite timeout
OpenOld     equ     0           ; Open existing exclusive device
iop_pinf    equ     $70         ; Get PE information
iop_outl    equ     $7a         ; Outline a primary Window
iop_rptr    equ     $71         ; Read the pointer
TermVec     equ     $01         ; When to stop reading
KeyStroke   equ     $0a         ; Keystroke or button
ESC         equ     $1b         ; ESC key code
Space       equ     ' '         ; One space
LineFeed    equ     $0a         ; Linefeed
```

Listing 21.3: Pointer Record Examiner - Equates

The usual standard QDOSMSQ job header needs no introduction by now either.

```
            bra.s   start
            dc.l    0
            dc.w    $4afb
JobName     dc.w    JobName_x-JobName-2
            dc.b    'PTR_RECORD Test v1'
JobName_x   equ     *
```

Listing 21.4: Pointer Record Examiner - Job Header

A few channel definitions and useful tables and such like come next. We are using a bigger window than the previous article as we have a bit of text to print in our window this time. The previous utility didn't do much at all, simply closing down when you clicked a button or pressed a key. This one loops around until you explicitly quit by pressing ESC.

```
ConChan     dc.w    4                   ; Console channel name
            dc.b    'con_'

ConDef      dc.w    412,156,50,30       ; Primary Window width, height, x, y

HexBuff     ds.w    1                   ; 2 Bytes storage for hex conversion

SpaceTab    dc.w    20,18,16,14,13,12,8,6,4,2

PtrRec      ds.w    12                  ; Pointer Record for IOP_RPTR
```

Listing 21.5: Pointer Record Examiner - Definitions

Next up we have the start of the code proper. Like last time, much of this could be considered boiler plate in that it never varies much. Obviously, my error trapping is quite simple, in the event of an error, bale out of the program. This is suitable for a small test program but in real life would need to be slightly more robust.

We start off by opening a channel to a console device. This will default the colours and so forth to a black paper and white text.

```
Start       moveq   #io_open,d0         ; Open a file or channel.
            moveq   #me,d1              ; Open for me
            moveq   #OpenOld,d3         ; Old exclusive device
            lea     ConChan,a0          ; Channel definition
            trap    #2                  ; Do it
            tst.l   d0                  ; OK
            bne     Exit                ; Nope, bale out.
```

Listing 21.6: Pointer Record Examiner - Open Console

Assuming the console has opened ok, we now redefine the size we want it to be and give it a red border one pixel wide. Once that has been done, we call CLS on the window.

```
            moveq   #sd_wdef,d0         ; Redefine window
            moveq   #2,d1               ; Red border
            moveq   #1,d2               ; One pixel wide
            moveq   #timeout,d3         ; Infinite timeout
            lea     ConDef,a1           ; Definition block
            bsr     Trap3               ; Do trap #3 return here if all ok.

            moveq   #sd_clear,d0        ; cls
            bsr.s   Trap3               ; Do trap #3 return here if all ok.
```

Listing 21.7: Pointer Record Examiner - Redefine Console

From this point onwards, both A0 and D3 are preserved by all the calls to `TRAP`s etc that we make in the program. You will not see these being set again.

Next, we have to find out if the user has loaded the Pointer Environment or not. If they have, we can continue with the remainder of the program, otherwise we simply bale out. A real program would display a message to the user telling them what the problem is and not simply 'vanish'.

```
FindPE      moveq   #iop_pinf,d0        ; Get PE information
            bsr.s   Trap3               ; Do trap #3 return here if all ok.
```

Listing 21.8: Pointer Record Examiner - Get Pointer Environment

The PE exists and is usable. We now have to outline our primary window. This defines the area in which all pointer operations take place for this application. We also add a 4 by 4 shadow to our display to give the appearance that our application's window is floating above the screen.

```
GotPE       moveq   #iop_outl,d0     ; Outline primary window
            move.l  #$00040004,d1    ; Shadow 4 by 4
            moveq   #0,d2            ; Ignore window contents
            lea     ConDef,a1        ; Outln size
            bsr.s   Trap3            ; Do trap #3 return here if all ok.
```

Listing 21.9: Pointer Record Examiner - Outline Primary Window

All the preparatory work for the PE has been done, we now display a message telling the user to 'press ESC to quit'. As we cleared the screen earlier on, this will appear at the top of our window. We also print a string containing headers to explain what each field of the (soon to be) printed output relates to.

```
            lea     SignOn,a1        ; Message, ESC to quit
            move.w  ut_mtext,a2      ; Print message vector
            jsr     (a2)             ; Do it
            bne     Exit             ; Bale out on error

            lea     Title,a1         ; Headings
            move.w  ut_mtext,a2      ; Print message vector
            jsr     (a2)             ; Do it
            bne     Exit             ; Bale out on error

            moveq     #-13,d4        ; to count 14 lines = first screen.
```

Listing 21.10: Pointer Record Examiner - Sign On

The main pointer loop begins here. As mentioned in the text, we are using the same termination vector as last time, return from `IOP_RPTR` when the user clicks a mouse button or presses a key.

```
Pointer     moveq   #iop_rptr,d0     ; Read pointer
            moveq   #0,d1            ; Initial x,y for pointer
            moveq   #TermVec,d2      ; Return on button or keypress
            lea     PtrRec,a1        ; Pointer record storage
            trap    #3               ; Do it
```

Listing 21.11: Pointer Record Examiner - Read Pointer

When we get to this point, the call to `IOP_RPTR` has returned and as part of that call, the pointer record has been filled in with data. This is where we start to print it all out.

There are 24 bytes in the pointer record, so we start by initialising our byte counter to 23 - as `DBF` requires. A2.L is set to the address of the pointer record and then we start a loop to convert each byte of the pointer record to hexadecimal and print it out.

```
PrintOut    moveq   #23,d7           ; 24 bytes to print out
            lea     PtrRec,a2        ; Location of data = pointer record
            addq.w  #1,d4            ; Line counter
            bmi.s   PLoop            ; Negative, headings won't scroll
            bsr.s   Scroll            ; Scroll and preserve headings

PLoop       move.b  (a2)+,d6         ; Fetch a byte from pointer record
            bsr.s   HexIt            ; Convert to hex in buffer at (A3)
            subq.l  #2,a3            ; Adjust buffer pointer
```

```
            exg     a1,a3           ; Buffer now in A1
            moveq   #io_sstrg,d0    ; Send bytes to channel
            moveq   #2,d2           ; Two bytes only
            bsr.s   Trap3           ; Do trap #3 return here if all ok.
            exg     a1,a3           ; Swap buffers back again
```

Listing 21.12: Pointer Record Examiner - Print Details

As we move through the buffer, D7 is used to keep track of how many bytes are still to be printed (minus one of course) so, at certain points along the way, we check if D7 is equal to one of the entries in our 'space table' and if so, we print a space. This is a quick and simple manner of splitting up the long string of characters that would result from converting 24 bytes to hexadecimal and printing them out.

```
SpaceReqd   lea     SpaceTab,a3     ; Table of space positions
            moveq   9,d5            ; 10 values in table

SpaceNext   cmp.w   (a3)+,d7        ; Is D7 a space position?
            dbeq    d5,SpaceNext    ; Scan until found, or not
            bne.s   LoopEnd         ; It was not found
            bsr.s   DoSpace         ; Print a single space
```

Listing 21.13: Pointer Record Examiner - Space Table

At the end of the main loop, when all 24 bytes have been converted and printed out, we throw a new line and get ready to see if we should quite or not.

```
LoopEnd     dbf     d7,PLoop        ; Do some more bytes

            bsr.s   DoLinefeed      ; Print a linefeed now
```

Listing 21.14: Pointer Record Examiner - Loop End

At this point we now start to use the data in the pointer record in 'anger'. We have printed the contents to the screen - so we will see what is in the buffer, however, if the key we pressed was ESC, we terminate the program. If it was some other key, we skip back to the start of the pointer loop and start off by reading the pointer again.

The ESC key has keycode 27 decimal or $1B hexadecimal and we look in the pointer record for that value as the key that was pressed. Remember, our termination vector said to exit when a key was pressed or button clicked so we are looking for a keystroke. It could be that we will find data elsewhere in the pointer record about our 'event' - time will tell.

```
Escape      lea     PtrRec,a2       ; Pointer record again
            cmpi.b  #esc,KeyStroke(a2) ; Got ESC key?
            bne.s   Pointer         ; Go around again
```

Listing 21.15: Pointer Record Examiner - Handle ESC

This is the end of the program. We arrive here when the user presses the ESC key or if any errors occur in setting up our windows and so on.

```
Exit        move.l  d0,d3           ; Error code in D3
            moveq   #mt_frjob,d0    ; Force remove a job.
            moveq   #me,d1          ; Job id of current job.
            trap    #1              ; Kill me
            bra.s   Exit            ; We never get here, but ...
```

Listing 21.16: Pointer Record Examiner - Exit Program

The next subroutine was added on advice from George. We scrolls up one line if we have filled the screen. This helps to keep the headings on the screen at all times. (Not in *my* original code.)

```
Scroll      moveq   #2,d2           ; line 2
            bsr.s   Pos             ; Set cursor below headings
            moveq   #-10,d1         ; Scroll up one line
            moveq   #sd_scrbt,d0    ; Scroll the lower part only
            bsr.s   Trap3
            moveq   #14,d2          ; line 14 (bottom line)
Pos         moveq   #0,d1           ; Set cursor back to x=0, y=14
            moveq   #sd_pos,d0      ; Drop in to trap3 code & return.
```

Listing 21.17: Pointer Record Examiner - Scroll Screen

This is another of George's suggested improvements, replace all those `TRAP #3` calls, and error checks with a single subroutine to do it all.

```
Trap3       trap    #3              ; Do the trap
            tst.l   d0              ; Did it work?
            bne.s   Oops            ; Fraid not
            rts                     ; Yes it did

Oops        addq.l  #4,a7           ; Delete the return address
            bra.s   Exit            ; Bale out
```

Listing 21.18: Pointer Record Examiner - Handle TRAPs

A sub-routine to take the byte value in D6 and convert it to a pair of Hexadecimal digits in the buffer pointed to by A3. This code trashes A3 and D6 but everything else is unaffected.

```
HexIt       lea     HexBuff,a3      ; Buffer for output
            move.b  d6,-(a7)        ; Save hex byte
            lsr.b   #4,d6           ; Keep high nibble in low nibble
            bsr.s   Nibble          ; Convert nibble to hex
            move.b  (a7)+,d6        ; Restore hex byte

Nibble      andi.b  #$0f,d6         ; Keep lower nibble
            cmpi.b  #10,d6          ; Check for a-f
            bcs.s   Add_0           ; No, 0-9 only
            addq.b  #7,d6           ; Offset to 'A'
Add_0       add.b   #'0',d6         ; ASCII code now
            move.b  d6,(a3)+        ; And buffer it
            rts                     ; Done
```

Listing 21.19: Pointer Record Examiner - Print Hexadecimal

A sub routine to print out a space to the channel in A0.L. This is used between fields of the pointer record to break up the monotony of 48 hexadecimal characters in a long string across the screen. This code trashes registers as per `IO_SBYTE` which is what it calls to do the work. There is another subroutine here as well that prints a linefeed at the end of each decoding of the pointer record.

```
DoSpace     moveq   #Space,d1       ; Print a space
            bra.s   DoIt            ; Skip next bit

DoLinefeed  moveq   #LineFeed,d1    ; Print a linefeed

DoIt        moveq   #io_sbyte,d0    ; Send one byte to channel
            trap    #3              ; Do it
```

```
                tst.l    d0              ; Ok
                bne      Exit            ; No bale out
                rts
```

Listing 21.20: Pointer Record Examiner - Print a Space

And finally in this file, the two messages we print at the start of the program. One telling the user how to quit and the other is used as the headings for the columns of data produced when we run the program.

Take note that there are two spaces after ‘Channel’ and one space before ‘wide’ in the following. ‘KS’ simply refers to Key Stroke and ‘KD’ is Key Down.

```
SignOn          dc.w     signon_x-signon-2
                dc.b     'Press ESC to quit...',10,10
SignOn_x        equ      *

Title           dc.w     Title_x-Title-2
                dc.b     'Channel  SubW PtrX PtrY KS KD EventVec'
                dc.b     ' Wide High Xorg Yorg',Linefeed
Title_x         equ      *
```

Listing 21.21: Pointer Record Examiner - Messages

The way the QDOSMSQ is written and the above program takes advantage of the fact, is that A0.L is never corrupted by any of the channel handling routines. I never have to - at least in the above simple code - preserve it anywhere. It simply remains unaffected from the time the channel is opened until the job is killed off. As George pointed out in his comments on my previous article, the timeout in D3 is also preserved. The above code takes that into consideration as well.

Running the program is simple, simply EX or EXEC it and a window will appear centralised on your screen. It will be showing a prompt that says to press ESC to quit. As written the code will return from the `IOP_RPTR` call when a key or button is pressed, but you can experiment with different settings in the termination vector to see what happens under different circumstances.

I’ve written the code to put a space between each field of the pointer record when printed out on the screen. It’s not the best way of doing things but is a lot easier to read than a string of 48 hex digits on screen in one line! Feel free to modify the code to print things in a better fashion if you wish!

When the code is run, move the pointer around, press various keys - try pressing keys together and see what results appear in the output. The channel Id should remain constant as should the width and placement of the window, but some of the other fields will change as you press different keys or click mouse buttons - try some together and see what you get.

As I experimented with my version of the utility, I discovered the following.

Using a termination vector of $01 - exit when a button or key is pressed:

- A HIT with the button space bar sets both KeyStroke and KeyDown to $01.
- A DO with the button or ENTER sets both to $02.
- A normal keypress only sets KeyStroke to the ASCII code of the key. KeyDown is zero.

The event vector takes on different values according to what has been happening in the window:

After the start of the program, the pointer remains inside the hit area, a click with the mouse buttons sets the vector to $2B. This is the value when SPACE or ENTER are pressed.

If the pointer remains inside the windows as above, any other keypress sets it to $2D.

If the pointer has been outside of the window and comes back in - which it has to for the program to register events, SPACE, ENTER, HIT or DO buttons set it once to $3B. Other keypresses set it once to $3D.

If the job is 'picked' the KeyStroke is set to $08 and the event vector is set to $3D.

If the pointer is on the border then that counts as being inside the hit area for the primary window, however, if it is on the shadow, that counts as outside the primary window. So the hit area is exactly the size you defined in the call to IOP_OUTL and the additional shadow area is just window decoration.

The event vector is a single long word which records all the events which have occurred in the call to IOP_RPTR. The documentation says that Table 21.2 is the structure of the event vector, so who am I to argue?

Pointer Level	Bit 0	Keyclick detected
	Bit 1	Key down
	Bit 2	Key up
	Bit 3	Pointer moved
	Bit 4	Pointer moved out of window
	Bit 5	Pointer was in the window
	Bit 6	Pointer hit the window edge
Sub-window	Bit 8	Sub-window split
	Bit 9	Sub-window join
	Bit 10	Sub-window pan
	Bit 11	Sub-window scroll
Window	Bit 16	Do
	Bit 17	Cancel
	Bit 18	Help
	Bit 19	Move
	Bit 20	Resize
	Bit 21	Sleep
	Bit 22	Wake
Job Level	Bit 24	Key or button pressed. Request resize (with bit 31)
	Bit 25	Key or button pressed subject to autorepeat. Request move (with bit 31)
	Bit 26	Key or button released
	Bit 27	Pointer moved from given co-ordinates
	Bit 28	Pointer moved out of window
	Bit 29	Pointer is inside the window
	Bit 30	Pointer hit the window edge
	Bit 31	Window request. Used also with bits 24 and 25.

Table 21.2: The Event Vector

George has also pointed out to me that a job can wait for a set of events or can send a set of events to another job. There are eight possible events each represented by a different bit in a byte. Thus sending the value 255 to another job is to send all events 0 to 7. Sending 36 would be to send events 2 and 5. Bits 24 to 31 of the event vector contain the job events that have occurred.

Not mentioned are events that can be sent to your job by another job. I do not have any documentation about the bits for that level and what they define. I'm sure one or two of my eagle eyed readers will let me know!

You can use the values returned from the code above to check the bits that are set in the event vector and see exactly what events were recorded while the call to `IOP_RPTR` was taking place.

21.3 Coming Up...

In the next chapter, we move on from PTR_GEN and into WMAN - at least, that's the plan.

22. WMAN, The Window Manager

22.1 Introduction

At the end of the last chapter I mentioned that we would be delving into the WMAN system next. Well, here we are. However, before we get down and dirty in the code, I need to make sure you all know what I'm talking about, so let's start with a brief introduction/reminder to WMAN and all it's constituent parts.

22.2 WMAN

Until now, we have been playing with the PE or Pointer Environment routines. These allow for a window to be outlined, the pointer to be drawn and read and so on. However, to use these few routines to write applications with multiple windows and so on, loose items, menus whatever, would be quite difficult. This isn't to say that it cannot be done, it's just difficult.

What we really need is a utility to allow us the ability to define our window structure, the loose items and so on contained within it and convert that into what QDOSMSQ really needs to have to be able to give us all the goodies we get from the PE, well, WMAN is just that.

Using WMAN we can define a window and all it's contents, then use the vectors from WMAN to setup, display, remove and interact with our application without having to write code to handle everything ourselves.

George Gwilt mentioned in a comment about part 20 of this series that I treated the call to `IOP_PINF` as a method of finding out whether or not the Pointer Environment had been loaded. While it does indeed do this, it also returns a vector to the current location of the WMAN utilities in memory in A1.L - and it is these vectors we will be exploring in the coming articles.

22.3 A Very Brief Overview Of WMAN

Before we go on, we need to know what all the bits of the PE actually are, so there now follows a small briefing on that very subject. I won't be spending a lot of time in the discussions so if you need further information there is a very good "Idiot's Guide To The PE" available on Dilwyn's web site at http://www.dilwyn.uk6.net/pe/peig/pe.html if you want to read it online or http://www.dilwyn.uk6.net/pe/peig.zip if you want to download it to read at your leisure.

22.3.1 Selection Keys

A selection key is simply the key that you press - when the pointer is over the appropriate primary window (see below) - to activate some function or feature of the program in question. It may cause an action to be carried out or simply highlight an option is a menu. Normally, the selection key is shown underlined, but this is not necessary, although it is more helpful to the user of the program if it is.

22.3.2 Hit and Do

When the mouse buttons are in use then a HIT is what happens when you click with the left mouse button and a DO is when you click with the right one. On the keyboard, a HIT is when you tap the spacebar and a DO is when you tap ENTER. The actions carried out when you HIT or DO may be the same or may be different - it's all down to how the programmers wrote the code.

22.3.3 Outline or Primary Window

I have mentioned outlines before, however, for the same of completeness, I'm reiterating here. The outline (or primary window) is the rectangle of your screen that the program will perform all it's workings within. Any secondary windows (see below) opened by the program must be fully contained within the area bounded by the outline.

Of course, some programs allow you to move their windows around the screen. This also moves the outline around and wherever the window ends up when the user has moved it, becomes the new outlined area and all secondary windows will now appear within the new location.

The biggest size that an outline can be is the maximum width and height of the screen minus the shadow width and depth.

22.3.4 Secondary Windows

Secondary Windows are things like QMENU's file open utilities and so on, pop-up messages giving you error messages and anything else that takes place within the outline or primary window.

22.3.5 Information Sub Windows

These are small areas of the primary or secondary windows that show static text or little images or whatever. The most commonly seen and recognisable ones are those green and white stippled 'caption bars' that most PE programs have at the top of every window.

Indeed, the caption bar for most PE programs that I know of is set up with a green and white stippled information window all the way across the top of the window, then on top of that there is another plain white information window nicely centralised horizontally on top of the first one. The program name or caption is then inserted as an Information Objects (see below) into this second information window.

22.3.6 Information Objects

Once an information sub window has been created you need something to put in it - for information purposes. To this end you need to create information objects. These can be text or blobs, sprites or patterns (see below). The most noticeable ones are the program name shown in the 'caption bar' of most PE programs.

22.3.7 Loose Items

Loose items are small 'buttons' with text or graphics on them. They usually have a border that magically appears when the pointer is within the bounds of the loose item in question. A hit or do on a loose item will cause some action to be carried out.

The popular loose items known to most users would probably be the ZZz, ESC, resize and move ones that appear in the caption bars of may PE programs.

22.3.8 Application Sub Windows

There's not much to say about the application sub windows really. They are what's left of the primary or secondary window after borders, information sub windows and loose items etc have been removed. They are the areas of the screen that the program prints it's output or allows input from the user and so on.

A graphics drawing program, for example, would use the application sub window to allow the user to draw whatever it is that they are drawing.

22.3.9 Pan and Scroll Bars

These are displayed if the data in an application sub window is too wide (pan) or too tall(scroll) to be displayed completely within the area of the screen set aside for the application sub window. GUI users on other system (Linux or Windows) will be familiar with the concept.

At first, these can be a nightmare as a 'DO' within the scroll bar (or pan bar) will split it and you then end up with two separately scrollable (and/or pannable) windows within the application window. Could be useful at times I suppose!

22.3.10 Sprites, Blobs and Patterns

A SPRITE is a picture that appears on the display somewhere. A pointer is just a sprite that is moved around the screen. Sprites may be drawn to look like text, for example, in logos and programmer's names etc, or they may be small pictures to represent some function of the program.

A BLOB is part of a sprite and holds only data that defines the shape. It has no colour information at all. The PATTERN is the part of the sprite that holds the colour data. Why separate them like this? I suspect it was to save memory - why bother having sprites defined with the same shape, just different colours - by defining the BLOB once and the PATTERS for the colours, you save repeating the blob data - perhaps?

Blobs and patters can be used independently of sprites though.

22.3.11 Border

The border around the primary and secondary windows, and indeed any other object, is optional and up to the programmer. However, most programs use borders.

When you move the pointer over a loose item, a border may appear around it to indicate that you can carry out some form of action if you were to hit or do the loose item in question. Once the pointer is outside the loose item boundary, the border may vanish.

22.3.12 Shadow

The shadow for a window is drawn down the right side and along the bottom. It is optional and entirely at the discretion of the developer. When in use, a shadow gives the impression that the window is hovering above the desktop. The shadow is outside the outline and does not register hit or do actions. It is purely decorative.

22.4 More Useful Utilities From George

The GWASL assembler that George Gwilt wrote has been used as the assembler of choice throughout this long running series. George has come up trumps again with another utility that allows the easy generation of assembler code that defines a WMAN windows definition (more on this later) and I've been testing it out. Unfortunately, my holiday got in the way and I have a new version of the utility and GWASL to test out at the moment.

I'm sure that these programs will soon be available from George's usual repository of fine code. In addition, I shall be trying these utilities out myself and reporting back.

22.5 WMAN Windows Definition.

As mentioned above, WMAN is slightly more involved that the bare bones PE in as much as it carries out a huge amount of work on your behalf. This is all work that you would have to write into each and every program you write using the WMAN system (here-after known collectively as the PE or the Pointer Environment) but in order to take advantage of all this hard work, you have to set things up in a standard manner.

If you look back an issue or so, you will notice that up until now, all my PE test programs simply opened a console and set an outline before entering the main loop to read the pointer, act upon it, repeat as necessary. Obviously, my test programs were small and insignificant - but even though, they could benefit from a bit more added 'sparkle'. The WMAN routines make this possible.

The first thing we have to do is create a definition of our window in memory. This will be in a standard format and when done, we call a WMAN routine (`WM_SETUP`) to initialise the various internals required to make our window work under WMAN. Let's now take a look at the standard definition as required by WMAN.

22.6 Standard Windows Definition

So, now you know what all the bits in a window are, we can get right in and start discussing the standard way we have to define a windows and all its *decorations*. Let's take a look at one that *someone else* prepared earlier.

The following is extracted from a small utility written by *Oliver Fink* many years ago. The utility shows various bits of information about the running QDOSMSQ system. I have modified the original in a few places but the full credit must remain with Oliver. The code is in the public domain.

The start of the definition is the main window itself:

```
;Main window definition :
          dc.w   160                 ; default window width
          dc.w   84                  ; height
          dc.w   146                 ; initial pointer x position
          dc.w   8                   ; y position
```

Listing 22.1: Main Window - Fixed Part

So far so simple, nothing much here that we haven't met already. All we are doing here is telling WMAN how big our window is to be and where within the window the pointer is to be positioned when the window is first drawn.

The above positioning of the pointer is relative to the window outline. So in our window which is 160 pixels wide, the pointer is located 146 pixels along - nearly at the far right end. It is located 8 pixels down from the top. When drawn on screen, this places the pointer directly over the ESC loose item.

When the program is first executed, the PE attempts to position the main window on the screen so that the requested position of the pointer is superimposed on the current pointer position on screen. This prevents disconcerting jumps of the pointer every time you start up a new program.

You can see this in action if you move the mouse around on screen, note where it ends up, then EX a new program that uses the PE. You will see that the main window appears wrapped around where you last saw the pointer.

Next we define the attributes for our window.

```
          dc.b   $00               ; MSbit clear to call CLS
;                                  ; LSbit clear allows cursor keys
;                                  ; to move pointer.
          dc.b   2                 ; shadow depth
          dc.w   1                 ; border width
          dc.w   0                 ; border colour (black)
          dc.w   7                 ; paper colour (white)
```

Listing 22.2: Main Window - Window Attributes

Again, there's nothing remarkably difficult here. Bit 7 of the first byte tells WMAN whether or not the window is to be cleared. Setting bit 7 says that the window *must not* be cleared. Following on, we define a shadow size for the bottom and right edges of our window. Bit 0 of this byte enables or disables the ability to move the pointer using the cursor keys.

Note

The ability to disable/enable cursor key pointer movement is only mentioned, briefly, in the QPTR manual under the section on the Working Definition.
Disabling the cursor key movement *also* disables the ability to select items in an application sub-window menu using the space bar or enter keys. This *could* be a bug!

Remembering back to our initial forays into the raw Pointer Environment, you may remember that we could have a different shadow depth on both of those sides, using WMAN, it appears that the shadow must be the same down each side. Oh well!

Note The documentation says that *for sub-windows the shadow depth should be zero.* Best we stick to that advice. Remember, a sub-window is one 'embedded' within the main window. See application sub-windows or information sub-windows above.

Next, and finally for the main window, we have the definition of where the default pointer sprite for the window is to be found.

```
            dc.w    0                       ; use default pointer
```

Listing 22.3: Main Window - Default Pointer

This is one of my changes. In a need to reduce the amount of code in the magazine and also, to reduce your typing, I've modified Oliver's definition to use the default arrow pointer in the main window and in the application sub-window which will be defined below. Oliver had a custom sprite for the main window and another for the application sub-window. Both have been removed. The original file had this definition (don't type this in!) and a chunk of code to define the sprite to be used.

```
            ; DO NOT TYPE THIS IN!
            dc.w    sprt-*                  ; pointer to pointer sprite
```

Listing 22.4: Do Not Type This In!

You should be aware that all pointers in a window definition are *word* sized and *relative* to their own position in the definition block.

Now, that implies that all object lists must be within plus or minus 16KB of the pointer position, which might be a problem when there are a lot of objects and so on to define. To this end, if bit zero is set - an odd address - then that offset is used as a pointer to a long word which itself is a relative pointer to the object in question. Obviously, the word length odd pointer obviously has to be made even first, this is done simply by clearing bit zero.

In the above, if the Pointer Sprite above was defined a long long way away, we would see something like this:

```
...
            dc.w    sprt-*+1                ; ODD Pointer to long pointer to
;                                           ; our window's pointer sprite
            ...
sprt        dc.l    Real_sprt-*             ; Long pointer to pointer sprite
            ...
Real_sprt   ....                            ; Pointer sprite definition
```

Listing 22.5: Do Not Type This In Either!

If any pointer is to something we don't need, then simply set it to zero. So, for example, instead of using Oliver's original '?' sprite (shaped like a question mark) we have defined this word pointer as zero and get the default arrow sprite instead. In this case, zero means 'use the default' but in other places, it means 'not used'.

Next to be defined are the attributes for all the loose items we will be using in the window. Starting with the easy bits:

```
; menu item attributes
            dc.w    1                       ; Current item border width
            dc.w    0                       ; Border colour (black)
```

Listing 22.6: Main Window - Current Loose Item - Border Attributes

As before, it is simple. We define a black border 1 pixel in width. When the pointer is over any of our loose items, this border will be drawn around it to indicate that 'you can do something here'.

Following the border, we have the attributes for the loose items that are unavailable, available and selected. These attributes require 8 bytes each and define paper and ink (for text objects contained within them) and also blobs and patterns for the other object types. We are only using the paper and ink attributes, but the others must be there. We use zero to indicate 'not in use'.

```
; Menu item unavailable
            dc.w   30                  ; Paper - green/white stipple
            dc.w   30                  ; Ink - green/white stipple
            dc.w   0                   ; Pointer to blob for pattern
            dc.w   0                   ; Pointer to pattern for blob

; Menu item available
            dc.w   7                   ; Paper - white
            dc.w   0                   ; Ink - black
            dc.w   0                   ; Pointer to blob for pattern
            dc.w   0                   ; Pointer to pattern for blob

; Menu item selected
            dc.w   4                   ; Paper - green
            dc.w   0                   ; Ink - black
            dc.w   0                   ; Pointer to blob for pattern
            dc.w   0                   ; Pointer to pattern for blob
```

Listing 22.7: Main Window - Loose Item Attributes

In this example program, the loose items are never anything except available (and very briefly, selected) so the unavailable attributes are never used, but they still have to be defined. Oliver has chosen to use a paper and ink colour of 30 for unavailable loose items. That value gives a pleasant green/white stipple. Don't worry about it because you will never see it!

Following the loose item attributes, we have a relative pointer to the help window. In this program, we are not using one, so that pointer gets the value of zero.

```
            dc.w   0                   ; Pointer to help window
```

Listing 22.8: Main Window - Help Window Details

That is the end of the fixed part of the window definition. So far so good, there has been nothing too difficult yet. I wonder what is coming?

The rest of the definition block defines the *repeating parts* of the window definition. What exactly does that mean?

The documentation has this to say about the repeating parts:

"To allow for a variety of different layouts within the window as the size of the window varies, part of the window definition may be repeated several times. The definition should be made in order of decreasing window size. The last definition which defines the smallest allowable window, should be followed by a word containing -1. If the top nibble of a layout size word is zero, then the layout may not be scaled. If it is %0100 then it may [be scaled]."

Note

This actually shows up what I think is a contradiction in the documentation. There are other values that can be used in the top nibble, not only %0000 and %0100. More on scaling later in the series.

So there you have it. The fixed part of the window defines the default layout for the window. That layout and all other possible ones allowed, need to be defined in the repeating part of the window definition.

A window can be scaled by WMAN if the definition allows for it. The scaling flag is the top nibble (4 bits) of the size words for the window layout. If the top nibble is %0000 then it cannot be scaled and if it is %0100 then it may be scaled.

Scaling applies separately to the width and to the height of the different layouts. You don't have to scale vertically and horizontally, you can pick one, the other or both as desired.

For simplicity, and because I have not investigated scaling yet, Oliver and I will be sticking to non-scaled windows for now. Scaling will be a subject for a future article.

The following is the repeating parts for our single, non-scaling layout in our small program.

```
; Base of repeated part of window definition
            dc.w    160                 ; Width for this layout
            dc.w    84                  ; Height for this layout

; Pointers to definition lists
            dc.w    il-*                ; Information sub-windows
            dc.w    ll-*                ; Loose menu items
            dc.w    al-*                ; Application sub-windows
```

Listing 22.9: Main Window - Repeating Part

The above would be repeated for each and every different allowable layout for the window, from the biggest to the smallest. Following on from the very last layout, the smallest allowed, we have the terminating word.

```
            dc.w    -1                  ; End flag
```

Listing 22.10: Main Window - Repeating Part - End Flag

So we allow one and only one layout, which just happens to be exactly the same size as the default one defined in the fixed part of the definition block. It has three pointers at the end for the information sub-windows, the loose items and any application sub-windows that are required in this layout. Each layout will have it's own list and they need not be the same for each different layout.

We shall pause for breath at this point and discuss these lists in the next article. Hopefully, the above was not too taxing and I've explained it better that I ever had it explained to me!

22.7 Coming Up...

The next chapter continues our look at the window definition by looking into the lists of objects attached to our window.

23. WMAN, The Journey Continues

23.1 Introduction

At the end of the previous article I promised that we would continue our look at the standard window definition from where we left off. In this article that is exactly what we shall be doing as we take a look into the lists of objects that hang off of our window. I'm referring to the information sub-windows, loose items and applications sub-window lists. In addition, we have also to consider the various objects that are used within these lists.

23.2 WMAN Standard Windows Definition - Continued

At the end of the previous article, we had reached the following definition for our example window:

```
; Main window definition.
          dc.w   160              ; Default window width
          dc.w   84               ; Height
          dc.w   146              ; Initial pointer x position
          dc.w   8                ; Y position
          dc.b   $00              ; MSbit clear to call CLS
          dc.b   2                ; Shadow depth
          dc.w   1                ; Border width
          dc.w   0                ; Border colour (black)
          dc.w   7                ; Paper colour (white)
          dc.w   0                ; Use default pointer

; Loose item attributes.
          dc.w   1                ; Current item border width
          dc.w   0                ; Border colour (black)

; Loose item unavailable.
          dc.w   30               ; Paper - green/white stipple
```

```
            dc.w    30                      ; Ink colour
            dc.w    0                       ; Pointer to blob for pattern
            dc.w    0                       ; Pointer to pattern for blob

; Loose item available.
            dc.w    7                       ; Paper colour (white)
            dc.w    0                       ; Ink colour (black)
            dc.w    0                       ; Pointer to blob for pattern
            dc.w    0                       ; Pointer to pattern for blob

; Loose item selected.
            dc.w    4                       ; Paper colour (green)
            dc.w    0                       ; Ink colour (black)
            dc.w    0                       ; Pointer to blob for pattern
            dc.w    0                       ; Pointer to pattern for blob

; Help window, if used.
            dc.w    0                       ; Pointer to help window

; Repeated part of window definition - from largest to smallest layout.
            dc.w    160                     ; Width for this layout
            dc.w    84                      ; Height for this layout

; Pointers to definition lists for this layout.
            dc.w    infoList-*              ; Info sub-windows
            dc.w    loosList-*              ; Loose items
            dc.w    appList-*               ; App sub-windows

            dc.w    -1                      ; End of layouts
```

Listing 23.1: WMAN Example Window

In this article, we will be concentrating on the final part of the above.

Before we move on, a little light relief. If I replace the pointers to the three lists in the final part of the layout definition above, with zero - to indicate that I have no loose items, information sub-windows or application sub-windows - and then run the resulting code, the following screenshot in Figure 23.1 shows what I get.

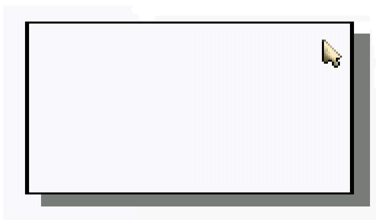


Figure 23.1: Basic WMAN Window

You can see that so far, all we have defined is a small white window, with a shadow and a black border. The pointer we are using is the default arrow and it is positioned close to the top at the far right of the window. At least it works!

Note You will not be able to assemble the code I have given you so far. There is a lot more coding to do before you get to that stage. I have a test harness wrapped around my window definition to make things easier for me to explain as I go along.

23.2.1 Information Sub-Window List

Most PE programs that I have ever seen have a caption bar across the top, possibly with a few loose items such as sleep (ZZz), Move and so on. The caption bar is usually - but not always - green and white stripes with the program name displayed in the middle on a white background. There are surprisingly, very few programs that do not stick to this colour scheme, however, the new graphics drivers are changing this and we are starting to get multi-coloured programs with trendy new 3D effects.

That sort of thing can wait until we get to grips with the basics, and so, in the age old traditions of green and white stripes, we shall continue! In addition, the fancy effects are only for those of us running SMSQ and so on, they are not available to the 128KB Standard Black Box QL users.

The usual method of getting the green and white caption bar is to define an information sub-window that covers the required length of the window and position it at the top of the window layout we are defining. The white background for the program name is simply a second information sub-window positioned over the first one. Finally, the title of the program itself is a text *object* that the second (plain white) information sub-window is linked to.

To be accurate, the program title is a text string embedded within a text object linked to the second information sub-window. All will become clear below.

The process could almost be likened to the following SuperBasic code.

```
1000 REMark Main Window
1010 OPEN #3,con_
1020 WINDOW #3,160,84,50,32
1030 PAPER #3,7
1040 BORDER #3,1,0
1050 CLS #3
1060 :
1070 REMark Caption Bar background
1080 :
1090 WINDOW #3,98,14,50+30,32+0+1
1100 PAPER #3,85
1110 CLS #3
1120 :
1130 REMark Caption Bar White Bit
1140 :
1150 WINDOW #3,52,10,50+54,32+3+1
1160 PAPER #3,7
1170 INK #3,0
1180 CLS #3
1190 :
1200 REMark Program title
1210 :
1220 PRINT #3,' SysInfo'
1230 :
1240 CLOSE #3
```

Listing 23.2: Pseudo SuperBasic Equivalent

It isn't quite the same as that, but things should hopefully become clear as we progress. For now, the definitions of the information sub-windows is shown below and should look strangely familiar.

```

; Information sub-window No. 0
```

```
infoList dc.w   98                    ; Sub-window width
         dc.w   14                    ; Sub-window height
         dc.w   30                    ; Sub-window x origin
         dc.w   0                     ; Sub-window y origin
         dc.b   $00                   ; MSbit clear to clear window
         dc.b   0                     ; Shadow depth
         dc.w   0                     ; Border width
         dc.w   0                     ; Border colour
         dc.w   85                    ; Paper colour (green/white)
         dc.w   0                     ; Pointer to info object list
```

Listing 23.3: WMAN Example Window - Information Window 0

Most of the above you have seen before in the fixed part of the main window definition. As mentioned in the previous article, the shadow depth for sub-windows must be zero. If you are like me, you'll be wondering what happens if you define a shadow on a sub-window. It appears, nothing. I tried putting a shadow of size 1 on an information sub- window and it simply was not drawn. I suspect that internally, WMAN is making as many sanity checks as it can and is probably ignoring the shadow size.

The definition above is equivalent to lines 1070 to 1120 in the SuperBasic code in Listing 23.2. That's an awful lot of typing for a simple result!

Next we need to define the second of our information sub-windows, the plain white one used as a background for the title.

```

; Information sub-window No. 1
         dc.w   52                    ; Sub-window width
         dc.w   10                    ; Sub-window height
         dc.w   54                    ; Sub-window x origin
         dc.w   3                     ; Sub-window y origin
         dc.b   $00                   ; MSbit clear to clear window
         dc.b   0                     ; Shadow depth
         dc.w   0                     ; Border width
         dc.w   0                     ; Border colour
         dc.w   7                     ; Paper colour (white)
         dc.w   infoObjs-*            ; Pointer to info object list

         dc.w   -1                    ; End flag
```

Listing 23.4: WMAN Example Window - Information Window 1

As this is our final information sub-window, there is a terminating word of -1 at the end of the definition. The one thing to notice in these definitions is a pointer to a list of information objects. These are explained next.

Setting the information objects list pointer to zero, in the above, and running the resulting program gives us the window in Figure 23.2. You can see both of the information sub-windows now, the green and white stripes is the first and the white one is the second. Next we shall look at adding an information object to the second one.

Information Sub-Window Object List

There are 4 different types of object that you can place within an information sub- window. These are shown in Table 23.1.

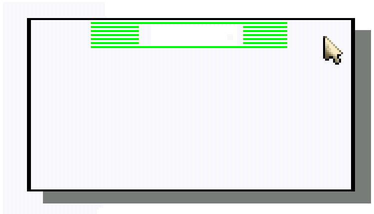


Figure 23.2: Basic WMAN Window - With Informations Windows

Type	Code	Description
Text	-N	This object is text. Character N will be underlined.
Text	0	This object is text. There will be no characters underlined.
Sprite	2	This object is a sprite.
Blob	4	This object is a blob.
Pattern	8	This object is a pattern.

Table 23.1: Information Sub-Window Object Types

If the type of the object is negative, then a text object is to be used and the character in the string corresponding to the negative number 'positivised' (I think I just made up a new word!) will be underlined. We are not using that here, but when we come to discuss Loose Items, we shall see an example or two.

The following is the definition of our text object for the program title.

```
infoObjs dc.w   42                      ; Object width
         dc.w   10                      ; Object height
         dc.w   6                       ; X origin
         dc.w   0                       ; Y origin
         dc.b   0                       ; Object type (See table)
         dc.b   0                       ; Spare
         dc.w   0                       ; Text ink colour
         dc.b   0                       ; Text character x size
         dc.b   0                       ; Text character y size
         dc.w   prgTitle-*              ; Pointer to object of correct type

         dc.w   -1                      ; end flag
```

Listing 23.5: WMAN Example Window - Information Object

As we only require one object for our information sub-window, there is the usual end of list indicator word of -1 after the definition.

Note

The information in the following paragraphs has been added by George Gwilt since the original article was published. These paragraphs correct the original one written by me.[1]

If the object is text, the word at offset 10 gives the colour of the ink to be used to display the text.

If the object is a blob (type 4) the word is used as a word relative pointer to the *pattern* to be used with the blob.

[1] In *QL Today* Magazine

If the object is a pattern (type 6) the word points to the *blob* to be used with the pattern.

If the object is a sprite, the word is not used. In all cases the word at offset 14 points to the object itself whether it is text, sprite, blob or pattern. Thus, for a sprite, its pointer is at offset 14, not 10 as Norman says.

Because this is a text object, we define the ink colour and the character sizes. However, if the object type is non-text ie a blob, pattern or sprite, then the 'ink' word is used as a word sized relative pointer to a pattern or blob or sprite and the character sizes are ignored. It may be wise to set those to zero just in case.

You will notice that the actual object content is defined elsewhere and one of those word sized relative pointers (or zero!) is used to tell WMAN where the content can be found.

Because our object is a text object, we simply define a QDSOMSQ format string as normal and make sure our pointer above actually points to the string. The definition for our program's title is as follows.

```
; Object No. 2          -> TEXT
prgTitle dc.w   7
         dc.b   'SysInfo'
```

Listing 23.6: WMAN Example Window - Information Object Text

Now that we have defined all the required information sub-windows and objects that are required for each, assembling my test program and running it gives the window in Figure 23.3.

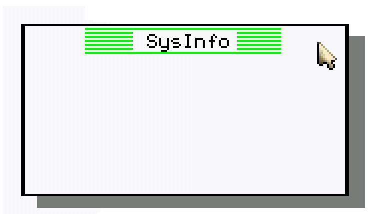


Figure 23.3: Basic WMAN Window - With an Information Object

Looks much better than the previous plain white version wouldn't you say? You can see spaces along the caption bar and these will be used - very soon - for a couple of loose items. Read on!

23.2.2 Loose Item List

Loose items are probably the QL's equivalent of Windows Buttons. The following is the definition of a loose item with a text object displayed upon it.

```
;Loose menu item No. 1
         dc.w   24                    ; Hit area width
         dc.w   11                    ; Height size
         dc.w   132                   ; X origin
         dc.w   2                     ; Y origin
         dc.b   0                     ; Object x justification
         dc.b   0                     ; Object y justification
         dc.b   0                     ; Object type
         dc.b   3                     ; Selection keystroke
         dc.w   objESC-*              ; Pointer to object
         dc.w   1                     ; Loose item number
```

```
        dc.w   escape-*              ; Pointer to action routine
```

Listing 23.7: WMAN Example Window - Loose Item 0

You can see a subtle difference between an information sub-window and a loose item definition. Loose items have the properties listed in Table 23.2.

Property	**Description**
Hit area width	The width of the loose item. Includes the border defined above in the fixed definition
Hit area height	The height of the loose item. Includes the border defined above in the fixed definition
X origin	Where the loose iten will be drawn. Relative to the start of the layout
Y origin	Where the loose iten will be drawn. Relative to the start of the layout
X justification	How the object will be positioned horozontally within the hit area
Y justification	How the object will be positioned vertically within the hit area
Object type	Same types and rules as for Information sub-window objects above
Selection Keystroke	For a letter, the upper case letter. For an event it is the event number minus 14
Pointer to object	The usual word sized relative pointer to an object of the correct type. Zero if no text.
Loose item number	The loose item number. You get to choose it.
Pointer to action routine	The address of the code to be called when this loose item is HIT or DOne

Table 23.2: Loose Item Properties

As mentioned in Table 23.2, objects are justified within the loose item hit area. This is different from the positioning of objects in information sub-windows. Table 23.3 shows the justification settings.

Code	**Description**
Positive	The object is left or top justified within the hit area
Zero	The object will be centred within the hit area
Negative	The object is right or bottom justified within the hit area

Table 23.3: Loose Item Object Justification Rules

If a key press is required to activate the loose item, it is defined by setting the code of the capital letter to be used.

Note

More from George:

First of all, *any* keypress including such things as TAB and the arrow keys can be used. The selection is not confined to letters and those keypresses which are defined as ‘events’. However, lower case letters are not allowed.

If, on the other hand, some event is to be used to activate the loose item, then the event number minus 14 is used instead. In our example above, the keystroke is set to 3 for ESC.

If you remember back to Chapter 21 when the event record was described, then you may get an inkling of what the event number actually is. It is the bit set in the event vector for the given action. Table 23.4 shows the events and their details.

Event Name	Event Number	Event Code	Description
DO	16	2	ENTER pressed or right mouse button clicked
CANCEL	17	3	ESC pressed
HELP	18	4	F1 pressed
MOVE	19	5	CTRL+F4 pressed
RESIZE	20	6	CTRL+F3 pressed
SLEEP	21	7	CTRL+F1 pressed
WAKE	22	8	CTRL+F2 pressed

Table 23.4: Events, Codes and Descriptions

Note

More updates by George.

The official documentation refers to "event number" and "event code". The event number is the number of the bit set in the event vector which is at position $14 in the window status area. For the seven events listed by Norman the corresponding bits to be set are 16 to 22. The event code is the event number less 14.

If a loose item is to be activated by a keypress producing an event the selection keystroke must be the event code as Norman says.

The action routine is called when the loose item is HIT or DOne. The parameters passed to the action routine will be discussed in a later article.

Loose Item Object List

Loose item objects are identical to those for information sub-windows and so, are the same to define. The following is an example of the text object required by our example loose item above.

```
; Object No. 4        -> TEXT
objESC     dc.w   3
           dc.b   'ESC'
```

Listing 23.8: WMAN Example Window - Loose Item Object Text

Nothing at all surprising there, it is a text object after all and as such, we simply define a QDOSMSQ string in the normal manner. Had the object been a blob, pattern or sprite, we would define one of those in the normal manner. More on those objects later on in the series.

Now that we have defined all the required loose items and objects that are required for each, assembling my test program and running it gives the following. I have moved the pointer from its default position in the screenshot in Figure 23.4 so that you can see the contents of all the loose items without obstruction.

All we need now is an application sub-window for our code to write to and we are ready to add actions etc. I shall keep you in suspense until next time.

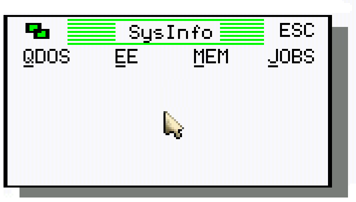


Figure 23.4: Basic WMAN Window - With Loose Items

23.3 Coming Up...

In the next chapter we shall continue looking at the remainder of the standard window definition. It seems like there is quite a lot going on, but it will hopefully soon be quite easily understood.

We will take a look at adding simple application sub-windows and creating loose item action routines. We might even get a working program to play with, who knows? See you then.

VII

SETW and Easy PEasy

24. Creating Your Own Windows With SETW

24.1 Introduction

In this chapter, I shall be taking a small diversion into one of George Gwilt's utility programs. This one, SETW, allows you to interactively create windows for your applications. SETW then goes away and does all the hard work of setting everything up.

24.2 Downloading SETW

SETW, and other useful utilities, are available from George's web site, `http://gwiltprogs.info/` and from there I advise you to download the following three utilities:

- SETW - setwp05.zip
- EasyPEasy - peassp02.zip
- GWASL - gwaslp07.zip

The latest version of GWASL is required to enable you to assemble PE programs created using SETW and using the EasyPEasy library files in peassp02.zip. As you will require these for the remainder of the tutorial then you should download them all now to save time later.

There are other files there similarly names but with a 'p' replaced by and 's' - these are the sources for the utilities and while educational, you don't need them.

The files are zipped up using the QDOS version of zip, so copy them from wherever you downloaded them to into your QL system (QPC etc) and unzip them using the QDOS version of unzip. There is one supplied with the C68 system and that works fine.

24.3 Running SETW

In order to create correctly written assembly source for GWASL, we need to pass a single parameter to SETW when we execute it. The parameter is "-abin" with no spaces. This tells SETW that the code produced will be used to build a binary file rather than a relocatable one which will be subsequently linked with other relocatable files to produce the final binary.

We GWASL users don't have a linker so all our programs need to be self contained, or may include pre-assembled modules and libraries using the LIB command.

```
EX SETW ; '-abin'
```

Listing 24.1: Executing SETW

The command above is all we need. If you do not have Toolkit 2, then the EXECUTE command from Turbo Toolkit can be used instead.

We will use SETW to create a file that we will use later on. It will be a very simple window with a single information window near the top and a single text object within the information window. Feel free to follow along on your own QL system as we go.

The program starts by opening a window as big as it can on your screen, it displays a few bits of information and prompts for the root name of the various files to be created.

For our example, we simply set the name to 'hello' - without the quotes. Type it in and press ENTER.

SETW will create three files when we are done. They will be created on ram1_ (in my case) or wherever you have configured SETW to put them by default. The three files created will be:

- Ram1_hello_wda - a file for use by George's TurboPTR utility. It is of no use to us and can be safely deleted when finished.
- Ram1_hello_asm - a file for use by an assembler, in our case, GWASL, this is the file we will need.
- Ram1_hello_z - a file for use with another of George's utilities, CPTR, a program to help C68 users write PE programs. Again, we don't need this file and it can safely be deleted.

24.3.1 Entering Text Objects

The next screen that appears is titled 'ALTER TEXT' and is where we enter every text object to be used in our finished utility. We must be very careful here and not forget any because SETW creates code for what we enter and we cannot go back and add another if we forget one. (Well possibly we can in the generated assembler file, but I have not confirmed this yet.)

To enter your text objects, press the 'N' key to create a new text object and simply type in the required text. For our example window all we need is one single object containing the text 'Hello World' (without quotes) - for the main reason that this is how everyone starts to learn a new language! Press ENTER when you have entered the text.

In slightly more complicated programs, there would be a lot more text objects to enter, but for now, press the ESC key to exit from the ALTER TEXT screen.

24.3.2 Entering Sprites, Blobs & Patterns

We don't need any sprites, blobs or patterns in this example, so simply press the ESC key when prompted for each of these.

24.3.3 The Main Window

The next prompt is to tell SETW about the main window, how many windows are needed and so on. In many cases the default is correct and all we need do is press ENTER at each prompt - however, make sure you read the prompt and think before pressing ENTER - once you have done so, there's no going back! (Ask me how I know!)

When asked for the number of main windows, accept the default of 1 by pressing ENTER.

When asked for the number of loose Items, accept the default of zero by pressing ENTER.

When asked for the number of Information Windows, we will need one, so press the '1' key and press ENTER.

We are now asked to enter the number of information objects in each information window. We require one information object in our one single information window. Type '1' and press ENTER.

When asked how many Applications Windows you want, accept the default of zero.

Next we are asked to select a shadow size. I find a size of 2 to be adequate. Type '2' and press ENTER.

For the border size choose a width of 1.

For all the prompts asking us to select a colour, select option 1 each time. Use the arrow keys to highlight the desired option and press ENTER to select it. We want "1. Default" for our colours. (I will explain the others later on in the series.)

Next we get to choose the sprite to be used as a pointer in the main window. I much prefer the standard arrow, so select it as above, and press ENTER. If you wish, you can choose another sprite to use as the pointer instead - it's your choice.

24.3.4 Information Windows & Objects

Now that all the details for the main window have been entered, or default chosen, we get to enter the requirements for each (or in our case, one!) Information Window.

First of all we need to enter the border width, I use a width of one pixel for all my programs. Type '1' and press ENTER.

Next we need the border colour, as before, select "1. Default" and press ENTER.

Select the default again for the paper colour.

We are now asked to select a type for our information objects for this information Window. As we only entered a single information object way back at the beginning and that was a text object, we should select "text" and press ENTER. Other object types would be available if we had entered any sprites, blobs or patterns.

Next we see a window appear with the list of (one!) text objects. As there is only one, it has been highlighted for us. Press ENTER to select it.

Select the default colour again.

When asked for the character sizes for X and Y for this text object, select zero for both.

24.3.5 Interactively Sizing The Window & Contents

Now the fun begins! A window appears that allows us to interactively resize the main window and the information window we have created. Once done, we can position these items almost at will.

Looking at the window currently being displayed, the lower right corner shows the currently defined dimensions for the main window itself. At the top left is an outline of the noted dimensions. We can use the arrow keys to change the dimensions - up makes the window less tall, down makes it taller, left makes the window narrower and right makes it wider.

Pressing the ALT key makes the change in size bigger. This saves wearing out your keyboard getting the window to the size you would like!

For this demonstration program, we require a window size of 200 wide by 100 deep. Use the ALT and arrow keys to make the dimensions 200 wide and 100 deep. When the desired dimensions have been achieved, press ENTER.

We are now asked if a variable window is to be created, this will be covered in a future tutorial so for now, type 'N'. (There is no need to press ENTER.)

Next we need to set the origin of the window. Again the arrow keys move things around and the ALT key makes the movements bigger. For the demonstration, set the origin to 50, 50. You will get a rough idea of where the origin will be as a small dot moves around the screen under the control of your arrow keys. Press ENTER when the origin is where you would like it to be.

Next up, we get to size our information windows, or window in our case! I have decided to make it slightly wider than the space required for the text object. That itself is 12 characters of 6 pixels wide or 72 pixels in total. As I like to have a bit of leading and trailing space in my information windows, use the arrow keys and ALT, as before, to resize information window number one to be 74 pixels wide by 12 deep. You may choose a different dimension if you like, but it will need to be a minimum of 72 pixels wide to hold all the text.

The program starts off in position mode rather than in size mode. You may need to press F2 to toggle between the two modes. Check the prompt on screen for advice about which mode you are currently in.

Once you have the desired size, press F2 and move the window to a position of 62 across by 2 down. If the information window size plus the position causes it to extend off the edge(s) of the main window, you will not be allowed to position it where you want to. In this case, toggle between size and position with the F2 key until you have it correctly sized and positioned.

Press ENTER when done. This takes you now to the sizing and positioning of the information window object (where the actual text object will be placed). If you remember the text object is 12 characters or 72 pixels wide which means that we need an object big enough to take that plus a little space at the beginning and end. As I like a couple of pixels either end of my objects, set the information object to be at position 4 across and 1 down. Press ENTER when satisfied.

That's it for our little test window. SETW now displays some information about the files it created and after a pause, or when you press a key, it will cycle through all the main windows we created - one in our case - and display them on screen as they have been defined.

At this point, there's not much we can do if it all went horribly wrong. We simply have to start again - or get down and dirty in the generated assembly file! Press ENTER to exit from SETW.

On ram1_, in my case, we now have the files that SETW generated for us. We are only interested in the hello_asm file and can happily delete the others. Feel free to examine the generated file in an editor and compare what has been created with the previous articles where I explain what the individual bits of a WMAN window definition are.

The assembly source file generated is not able to be assembled as it is and then run, it has no code in it to make it a correctly functioning QDOS job. That comes later.

Until next time, feel free to generate more windows of your own and get to know George's SETW utility - we will be using it in future articles in this series.

24.4 Coming Up...

We take the file we created with SETW and feed it into another of George's utilities, EasyPEasy, in the next chapter. EasyPeasy tries to make coding for the PE much easier. Until next time, happy windowing.

25. Easy PEasy - Part 1.

25.1 Introduction.

At the end of the previous chapter, we had created a very minimal ‘Hello World’ window using George Gwilt’s SETW application. In this chapter, we take a first look at George’s other utility to make PE programming easy, EasyPEasy. As mentioned last time, you should have downloaded the peasp02.zip file from George’s website. If you find a later version (say peasp03.zip or higher, get that instead!) The website address is `http://gwiltprogs.info/`.

25.2 Easy PEasy.

Unlike many other utilities, Easy PEasy isn’t a program you can run, it is a collection of information and small binary files that you can include with your own programs - using the LIB and IN commands in your source code and assembling with GWASL - to make programming the Pointer Environment a little easier. Actually, quite a lot easier as George has done much of the hard work, all we have to do is open a console, make a few checks and write the code to handle our own needs as opposed to the needs of getting the PE up and running.

Much of what follows in this chapter is a blatant theft of George’s readme file. For this I make no apology - there is no better way to document something that straight from the horse’s mouth!

25.3 The Nine Steps To Happiness.

With Easy PEasy, there are nine steps to happiness. The following is basically a skeleton for writing a PE program in assembly language:

1. Initialise your program and open a con_ channel.
2. Are the PTR_GEN & WMAN present? Abort the program if not.
3. Set up the window working definition.

4. Position the window.
5. Draw the window contents.
6. Read the pointer.
7. Did we have an error - exit if so, else it was an event.
8. Process the event.
9. Goto step 6.

Each step in the above, thanks to the coding that George has done, is quite simple.

25.3.1 Initialise.

The initialisation step consists of setting up your console channel and opening it. The standard QDOSMSQ job header is also required. The code is very simple, and looks like that shown in Listing 25.1. It is best to do the setup as soon as possible after the program is executed rather than setting up other stuff first. It saves time and effort - in case something goes wrong and you have to bale out.

```
          bra.s     start
          dc.l      0
          dc.w      $4afb

id        equ       0                      ; Storage for channel id
wmvec     equ       4                      ; Storage for WMAN vector
slimit    equ       8                      ; Storage for Window limits call

jname     dc.w      jname_e-jname-2
          dc.b      "My EPE Program"
jname_e   ds.b      0
          ds.w      0

; Console definition.
con       dc.w      4
          dc.b      'con_'

; The job starts here.
start     lea       (a6,a4.l),a6           ; Get the dataspace address in A6.L

          lea       con,a0                 ; Con_ channel definition
          moveq     #-1,d1                 ; Required for this job
          moveq     #0,d3
          moveq     #io_open,d0
          trap      #2                     ; Open the channel
          tst.l     d0                     ; Did it work?
          bne       sui                    ; Exit via sui routine in EasyPEasy
          move.l    a0,id(a6)              ; Save the channel id
```

Listing 25.1: EasyPEasy Standard Code - Initialisation

25.3.2 Check The PE & WMAN.

The console is open now, or we have baled out of the program. Obviously we don't get much feedback from the program if anything went wrong, a proper user friendly application would, of course, display a suitable error message. The next easy step is to check for the presence or otherwise of PTR_GEN and WMAN as per Listing 25.2.

The following code requires a channel id, for a CON_ channel, to be in A0.

```
; Check for PE being present.
        moveq     #iop_pinf,d0
        moveq     #-1,d3
        trap      #3
        tst.l     d0                  ; Ptr_gen present?
        bne       sui                 ; No, bale out
        move.l    a1,wmvec(a6)        ; Yes, save WMAN vector
        beq       sui                 ; Oops! Bale out, no WMAN
        movea.l   a1,a2               ; Keep WM vector in A2
        lea       slimit(a6),a1       ; Storage, 4 words long
        moveq     #iop_flim,d0        ; Need maximum size of window
        trap      #3
        subi.l    #$C0008,(a1)        ; Less 12 (width) and 8 (height)
```

Listing 25.2: EasyPEasy Standard Code - Checking for the PE

The code in Listing 25.2 checks for the PE being present and if not found, bales out via the code at sui. If the PE is found, the WMAN vector is saved in data space for later use - however, if WMAN is not loaded (but PTR_GEN is) the job will exit via the familiar sui routine. Easy PEasy requires both the PTR_GEN and WMAN files to be loaded in order to create and run PE programs.

Next up, we find out the maximum size that the con_ channel can grow to. We assume that that code always works - but it may be good practice to check, just in case. The 4 words returned indicate the size and position of the con_ channel, and these 4 words are placed into the job's data space and a small margin is subtracted from the width and height.

25.3.3 Set The Window Definition.

The window definition is expected to hold a value in wd0 for the size of working definition and status area space. The code in Listing 25.3 reads the amount of memory required for the window definition (created by SETW and defined in ww0_0) and allocates space in the common heap for our program to use. If this fails, the call to getsp will never return - it exits through the sui code on error.

```
; Reserve memory for the window working definition.
        lea       wd0,a3              ; Address of window definition
        move.l    #ww0_0,d1           ; Size of working definition
        bsr       getsp               ; Allocate space
        movea.l   a0,a4               ; Save in A4.L too
```

Listing 25.3: EasyPEasy Standard Code - Allocate Memory for the Window Definition

If the memory allocation worked, the address is returned in A0 and we save it in A4 for later use. This is a handy feature of Easy PEasy and the way it was written by George.

Before we can call the WMAN routine to set up our window - `wm_setup` - we need to make sure that the status area for loose items is all initialised properly. The code in Listing 25.4 assumes that all loose items will be available when the program starts. Zero is the value we need for available.

The labels wst0 and wst0_e are defined by the SETW program. (As you can see SETW does most of the hard work of calculating various sizes and labels for us!)

```
; Preset all Loose Items to available.
          movea.l   id(a6),a0           ; Restore channel ID
          moveq     #wst0_e-wst0-1,d1   ; Size of status area - 1
```

```
            lea       wst0,a1          ; Wst0 = status area address

loop        clr.b     (a1)+            ; Zero = Loose Item is available
            dbf       d1,loop          ; Clear entire area
            lea       wst0,a1          ; Reset pointer to status area
            moveq     #0,d1            ; Default window size
            lea       wd0,a3           ; Wd0 = window definition address
            jsr       wm_setup(a2)     ; Create the working definition
```

Listing 25.4: EasyPEasy Standard Code - Loose Item Initialisation

25.3.4 Position The Window.

This is probably one of the easiest parts of the code! We assume that the pointer is in the position on screen where we wish the window to appear. The position of the window may move to make sure that it remains on the screen, however, in normal circumstances, the pointer in our window will be positions in exactly the same place where the on screen pointer is now.

> **Note** If you don't default the pointer position to -1 to indicate where the pointer is now, then you must note that the value in D1 is in absolute screen coordinates relative to the start of the screen (at 0,0) and not relative to the program's main window or to any application sub-windows within.

This can be useful if a window has no 'move' abilities - you can simply put the pointer where you wish the window to appear, and execute the program. The window will be drawn exactly (adjusted to fit on screen) where you have put the pointer.

```
; Position, but do not draw, the window.
            moveq     #-1,d1           ; Position at pointer position
            jsr       wm_prpos(a2)     ; It's a primary window
```

Listing 25.5: EasyPEasy Standard Code - Position the Window

25.3.5 Draw The Contents.

The windows has been positioned, however, it has not been drawn on screen, so we need to draw it now. This is even simpler than the positioning of the windows.

```
; Draw the window.
            jsr       wm_draw(a2)      ; Draw the window and its contents
```

Listing 25.6: EasyPEasy Standard Code - Draw the Window

That was difficult! ;-)

25.3.6 The Pointer Loop.

At this point, we have the windows on screen and the user is waiting to use the application. We have to enter a loop to read the pointer and act accordingly.

```
; Main pointer reading loop.
read_ptr    jsr       wm_rptr(a2)      ; Read the pointer.
;                                      ; Does not return until
```

```
;                                       ; Either D0 or D4 are non-zero
```

Listing 25.7: EasyPEasy Standard Code - Reading the Pointer

For most loose items, application windows and so on, an action routine will have been defined and coded. These action routines will be discussed later. The pointer reading routine - `wm_rptr` - will not return until either D0 or D4 are non-zero as a result of an action routine.

25.3.7 Error Or Event?

If D0 is non-zero, and error has occurred and we should (somehow) handle it and probably bale out of the program. Alternatively, we can simply ignore errors and try again. The program developer decides.

If D4 is non-zero, an event has occurred and we need to handle that in our code before, possibly, returning to the pointer reading loop again.

An event is defined as a key press such as ENTER while the pointer is not positioned on a loose item or menu item, ESC, F1 (Help) or any of the CTRL+Fn key combinations - SLEEP, WAKE, MOVE or SIZE - but only provided that the key press doesn't select a menu item.

An event can be generated by any of the action routines as well. Within the action routines the programmer has the choice of either handling the action code there and then, or, setting an event in D4 and returning. This will cause the call to `wm_rptr` to exit and return back to the application where the event can be handled.

Some programmers like to control where and when the action handling code is performed and like to keep it all in the main code, others like to carry out the actions within the action handlers. It's entirely up to the developer - the end user will see no difference whichever method is chosen.

Obviously, how a program handles errors and events is up to the programmer and a generic method can't be given here. However, as an example, the following may suffice.

```
; Ignore errors.
        bne.s     read_ptr              ; Error in D0? If so, ignore it
;                                       ; This assumes there is an option in
;                                       ; the program to let the user EXIT
```

Listing 25.8: EasyPEasy Standard Code - Error or Event Check

25.3.8 Process Events.

At this stage in our program, we have returned from reading the pointer (`wm_rptr`) and no errors have been reported (in D0), so we must have detected an event in D4. We have three choices here - if our action routines should have handled things, then perhaps we should ignore the event and read the pointer again - alternatively, this could be an error and we should abort the program. The other alternative is that our action routines have set the event in D4, so our code should now process the appropriate event.

As above when trapping errors, there's no 'one size fits all' answer and every program should handle events accordingly. The following is an example whereby the events are simply ignored and we return to reading the pointer.

```
; Ignore events.
        bra.s     read_ptr              ; Ignore event numbers in D4
```

Listing 25.9: EasyPEasy Standard Code - Ignore Events

Obviously, if your code is processing events 'outside the action routines' then your own code, to process the appropriate event, would go here, rather than simply ignoring the events.

The event numbers are discussed below in 'Loose Item Action Routines'.

25.3.9 Repeat.

Repeat has already been handled above. All we do - in this simple example - is loop back to read the pointer when we hit an error or when any event occurs.

25.4 Loose Item Action Routines.

There are two kinds of action routines you need to be aware of. Those for loose items and those for application menu items. As we have not yet discussed much for Application Windows or their menu items, they will be discussed later.

An action routine for a loose item is called from within the `wm_rptr` call, and if the action routine exits with D0 and D4 both set to zero, the `wm_rptr` call will resume again - in other words, control will not return to your own code just yet.

On entry to a loose item action routine various registers are set with specific parameters:

Register	Description
D1.L	High word = pointer X position, Low Word = pointer Y position.
D2.W	Selection keystroke letter, in its *upper cased* format, or 1 = Hit/SPACE or 2 = DO/ENTER. D2.W may be an *event code* if an event triggered this action.
D4.B	An event number - see below - if an event triggered this action routine.
A0.L	Channel id.
A1.L	Pointer to the status area.
A2.L	WMAN vector.
A3.L	Pointer to loose menu item.
A4.L	Pointer to window working definition.

Table 25.1: Loose Item Action Routine - Entry Registers

If the loose item was triggered as the result of a selection keystroke, D2.W will hold the uppercased letter code.

If the loose item was triggered as a result of an event, D4.B holds the *event number* and D2.W holds the *event code* in the table below.

In addition to the above, the status of the loose item which triggered the action routine will be set to selected. It is not reset to available on exit, this is your responsibility.

Action routines must exit with D5, D6 and D7, A0 and A4 preserved to the same value that they had on entry to the routine. In addition, the code must set the SR according to the value in D0. A5 and A6 can be used and left at any value by the action routines, while D1 - D3 and A1 - A3 appear to be undefined as to their exit status.

If an error is detected, the routine should exit with an error code in D0 and the SR set accordingly. If the action routine simply wishes to cause `wm_rptr` to return to the user's code where an event will be processed - rather than processing it in the action routine itself, D4 should be set to the correct event number that the user code should process.

Event	Keystroke	Event Number (D4.B)	Event Code (D2.W)
Hit	Space	0	1
Do	Enter	16	2
Cancel	ESC	17	3
Help	F1	18	4
Move	CTRL+F4	19	5
Size	CTRL+F3	20	6
Sleep	CTRL+F1	21	7
Wake	CTRL+F2	22	8

Table 25.2: Loose Item Action Routine - Event Settings

Obviously, if setting D4 with an event number then D4 should be set before D0 otherwise the SR will take on the settings for D4 instead of D0.

If no error was detected by the action routine, and no event is to be returned, both D0 and D4 must be set to zero on exit.

The action routine needs to perform the application specific code to process the loose item that was triggered, however, it must also reset the status of the loose item that triggered the action routine. This can be done as follows.

```
; Action routine code goes here ...
        move.l    d5-d7/a0-a4,-(a7) ; Preserve registers that we need
        ...                         ; Do your stuff!
        move.l    (a7)+,d5-d7/a0-a4 ; Restore registers prior to exit

; Reset loose item status as part of action routine.
        move.w    wwl_item(a3),d1  ; Get the loose item number
        move.b    #wsi_mkav,ws_litem(a1,d1.w)  ; Redraw as available
        moveq     #-1,d3           ; Request a selective redraw
        jsr       wm_ldraw(a2)     ; Redraw selected lose items
        moveq     #0,d4            ; No event signalled here
        moveq     #0,d0            ; No errors either
        rts                        ; Back to wm_rptr
```

Listing 25.10: EasyPEasy Standard Code - Actions

The code above could be used as a template for loose item action routines. It begins by preserving the registers that we must preserve plus, it stacks A1 and A2 as well - for added safety, as they will be required in the code to reset the loose item status.

Should you reset the status on entry to the routine or exit? It's up to your code obviously. However, I prefer to do it at the end of the action routine. If the action routine is short and quick, it probably makes no difference. If the routine takes some time - lets say, it's formatting a floppy disc - then it's best to leave it at selected until the format finishes and then reset it. However, it's your choice.

25.5 Coming Up...

In the upcoming chapter, we'll take a deeper look at Easy PEasy and the routines that George has written for us. If we have time and space, we might take a look at an example of its use. See you then.

26. Easy PEasy - Part 2.

26.1 Introduction.

At the end of the previous chapter - Easy PEasy Part 1, I promised to take a look at the various code routines that George has written to make life a lot easier for PE assembly language programmers. If you haven't already done so, get over to George's web site and download the programs mentioned last time. The website address is `http://gwiltprogs.info/`.

26.2 Easy PEasy.

As I mentioned last time, Easy PEasy isn't a program you can run, it is a collection of information and small binary files that you can include with your own programs - using the LIB and IN commands in your source code and assembling with GWASL - to make programming the Pointer Environment a little easier.

26.3 Supplied Files.

With Easy PEasy, there are a number of files supplied, these are:

Keys_pe A file that can be included in your source file to define a number of equates for the various Trap #3 routines introduced by the PE.

Keys_wdef Another include file. This one defines the WMAN window definition equates.

Keys_wman Similar to keys_pe above but this file defines the equates for WMAN routines and vectors.

Keys_wstatus This file defines the equates etc for the window status area.

Keys_wwork This file contains the definitions for the window working definition.

Qdos_pt The equates etc for the PE interface.

Csprc_bin Some sprites, mostly for mode 4 but a few exist for mode 8. This file should be LIBbed by your own programs to use the sprites.

Csprc_sym_lst This file lists the names of all the sprites in the above file. If you need to use a sprite in the above (binary) file, you must use the name listed in this file.

Peas_bin This file contains all the useful code subroutines that George has written to make using the PE from assembly language easy. This file is binary and as such, should be LIBbed by your source code.

Peas_sym_lst This file lists all the routines supplied in the above file. Make sure that you use the name(s) listed in this file if you wish to use George's code in your own PE programs.

26.4 Subroutines in Easy PEasy.

The file peas_bin should be included at the very end of your own program's code, as follows:

```
        in          win1_source_easypeasy_peas_sym_lst
        lib         win1_source_easypeasy_peas_bin
```

Listing 26.1: Invoking EasyPEasy in Your Own Programs

The first 'in' line includes the peas_sym_lst file which defines offsets from the current position to the entry points for the routines in the peas_bin file which is copied 'as is' straight into your final executable file. For this reason, you must keep these lines together and in the order shown above.

Routine	Description
GetSp	Allocates an area of memory and returns the address in A0.L. The size of the area required must be passed in D1.L on entry. No other registers are affected. Exits via SUI (see below) if the memory allocation causes an error.
Rechp	Deallocates and frees an area of memory allocated by GetSp above. The address should be passed in A4.L. No other registers are affected.
Move	Processes a MOVE request then returns with D4 and D0 both set to zero. No other registers are affected. Can be called from inside the MOVE action routine in your own programs.
Sleep	Puts the program to sleep and creates a button in the button frame - if present. If the button frame is not present, the button will be placed on the top left of the display. See below for register usage.
Set_AP	Set an application window menu. See below for register usage. All registers are preserved on exit.
Sui	The program exits without warning and without any error messages. GetSp above will exit through here if there is an error when allocating memory.

Table 26.1: EasyPEasy Library Routines

26.4.1 GetSp

GetSp allocates an area of memory for the current job, and returns the address in register A0.L. There are no errors returned (in D0) as the routine exits through sui (below) if it detects an error. Only register A0.L is affected by the routine - all others are preserved.

On entry, the number of bytes required should be held in D1.L. On exit, A0.L holds the address of the allocated area. An example of use, taken from George's example EX0_asm, can be see in Listing 26.2.

```
        ...
        move.l    #ww0_0,d1         ; Size of working definition.
        bsr       getsp             ; Return ALCHP'd address in A0.
        movea.l   a0,a4             ; Copy to A4.
        ...
```

Listing 26.2: EasyPEasy - GetSP Example

There is no requirement to check for an error with this routine, if it returns to your program then it has worked.

26.4.2 Rechp

Rechp returns an area of memory, probably allocated using GetSp above, to the system. The address to deallocate must be passed in A4.L. All other registers are preserved and no errors are returned by this routine. An example of use would be after unsetting a widow definition, as per the following from EX0_asm:

```
        ...
        jsr       wm_unset(a2)
        bsr       rechp
        ...
```

Listing 26.3: EasyPEasy - Rechp Example

Again, there is no need to check for errors as the routine never fails.

26.4.3 Move

Move is called when a program detects that the user has requested a MOVE be carried out. The routine can be called either from your own code (if the read pointer loop exits with D0/D4 not zero) or from within an action routine called by the read pointer loop. In either case, calling the move routine is as simple as this:

```
; MOVE loose item action routine.
afun0_0   bsr       move              ; Process a MOVE.
          ...
```

Listing 26.4: EasyPEasy - Move Example

The above is another example taken from George's EX0_asm example program. After processing the move, the program needs to reset the loose item that caused the move request. See below for a fuller explanation of the example program and the code that is used to reset the loose items.

26.4.4 Sleep

Sleep sets the program to a button which contains the name of the program and is placed in the button frame if there is one or at the top left of the screen if there isn't.

While in button mode, A HIT - left mouse click or SPACE - on the button will cause the program to waken and restore itself to full size again.

A DO - right mouse click or ENTER - on the button will cause the program to waken if the program is currently located in the button frame, or, causes a move if the button frame is not present.

The registers required to call sleep are shown in Table 26.2.

Register	Description
D1.L	The size needed for the button. Can be obtained from ww0_1.
D2.L	The size needed for main window. Can be obtained from ww0_0.
A2.L	The WMAN vector.
A4.L	Pointer to the window working definition for the button window.

Table 26.2: EasyPEasy Sleep Entry Registers

On exit from the sleep routine, the registers are set as per Table 26.3.

Register	Description
D1-D3	Undefined.
A0.L	The channel ID.
A1.L	Undefined.
A2.L	Preserved - the WMAN vector.
A3.L	The window definition address.
A4.L	Pointer to the working definition which may have changed.

Table 26.3: EasyPEasy Sleep Exit Registers

As before, the following is an example from EX0_asm where the SLEEP loose item sets the sleep event in D4 and returns. This causes the read pointer loop to exit back to the user's code where the events etc are checked. The following extract shows the checks made to handle the sleep event being detected:

```
            ...
no_er2      btst     #pt__zzzz,wsp_weve(a1) ; Was it a SLEEP event?
            beq.s    wrpt                   ; No, read the pointer again
            move.l   #ww0_1,d1              ; Get main window button size
            move.l   #ww0_0,d2              ; Get main window size
            bsr      sleep                  ; Process a SLEEP
            bra.s    wrpt                   ; Read the pointer again
            ...
```

Listing 26.5: EasyPEasy - Sleep Example

In the above extract, we can see D1 and D2 being set to the sizes calculated (by SETW - see previous chapter) for the main window and the buttonised window. Registers A2 and A4 are correctly set.

After calling sleep, the program must continue to read the pointer otherwise it won't know if a DO or a HIT has been detected, or if it has been woken from slumber etc.

26.4.5 Set_AP

Set_AP is used to create an application window menu within a particular application window for a program. It is assumed that each item in the menu will be exactly the same length, although if QDOS strings are being used the word count for each one will determine what appears.

The registers required to call Set_AP are defined in Table 26.4.

Register	Description
D1.W	How many items are present?
D2.W	The length of each item.
A0.L	Pointer to the start of the list of items.
A1.L	Pointer to the application window.
A4.L	Pointer to the window working definition.

Table 26.4: EasyPEasy Set_AP Entry Registers

On exit, all registers are preserved.

George has provided an example program that uses this routine, EX1_asm. When run, the program displays a list of files on flp1_ when you click on the Display loose item. You can then select as many files as you wish, and click the Copy loose item. The selected files will then be copied to ram1_. The appropriate extract from this demonstration program is:

```
        ...
        movea.l   fnmes(a6),a0     ; Pointer to list of file names
        moveq     #36,d2           ; Interval between entries
        movea.l   a1,a5            ; Needed later on, saved
        movea.l   ww_pappl(a4),a1  ; List of app window pointers
        movea.l   (a1),a1          ; Get app window 0 from list
        bsr       set_ap           ; Set the app window menu
        ...
```

Listing 26.6: EasyPEasy - SetAP Example

The program has previously read the directory of all files (but no directories etc) on flp1_ into the area of memory addressed by A0. The data stored there (effectively) looks like the following:

```
        ...
item_1  dc.w      4                ; Length of string
        dc.b      'boot'           ; Filename from flp1_
        dc.b      0,0, ...         ; 32 padding bytes

item_2  dc.w      7                ; Length of string
        dc.b      'boot_pe'        ; Filename from flp1_
        dc.b      0,0, ...         ; 29 padding bytes
        ...
```

Listing 26.7: EasyPEasy - SetAP Item List

You can see from the above that each entry is a total of 36 bytes long (the difference between addresses item_1 and item_2) although the actual menu items themselves, the filename, need not be exactly 36 bytes. Regardless of the value of the padding bytes, the data displayed in the menu items will only show the actual filenames as defined by the QDOS strings making up each item.

An article of application window menus will be coming soon in this series.

26.4.6 Sui

Sui is a dramatic routine to call. Wherever your program is in its processing, calling sui will cause it to exit. In addition, the GetSp routine (above) will call sui if it cannot allocate a suitable area of memory. George's example program calls sui when it detects that the Pointer Environment is not present, as follows:

```
        ...
        moveq     #iop_pinf,d0      ; Find Pointer Environment & WMAN
        moveq     #-1,d3            ; Timeout
        trap      #3                ; Do it
        tst.l     d0                ; Did it work?
        bne       sui               ; Failed, or PE absent, bale out
        ...
```

Listing 26.8: EasyPEasy - Sui Example

No registers are used by this routine. It never returns an error - because it never actually returns!

26.5 The Example Program, EX0_asm.

So, having discussed the various bits and pieces of Easy PEasy, lets dissect one of George's example. The simplest example is EX0_asm and it's corresponding SETW designed window file, EX0w_asm, so those are what we will look at next.

This example simply shows how to use the four main events in a PE program:

- Move - moves the window around the screen.
- Resize - allows the window to be sized.
- Sleep - puts the program to sleep either in the button frame, if present, or on screen.
- Esc - exit from the program.

The program looks like Figure 26.5 when running on QPC:

Figure 26.1: Example program EX0 in action.

The window in Figure 26.5 shows an outline with a green border and an *interesting* paper colour. A white information window is displayed listing the various loose items in the program. The information window is white with a green border and black ink.

Along the very top of the window we can see the program's title - BASICS - in a red papered information window with a green border and ink, and the four loose items for MOVE and SIZE on the left with ESC and SLEEP on the right.

The window has been created by SETW and the definitions are all held in the file EX0w_asm which is supplied in the peass download from George's web site.

What follows is a slightly amended version of the program supplied by George. I have updated some of the comments to make then more readable and understandable (by me!) and in a couple of places, I have rearranged the order of some of the instructions - with George's blessings of course.

```
;------------------------------------------------------------------------------
; Standard job header.
;------------------------------------------------------------------------------
          bra.s      start
          dc.l       0
          dc.w       $4afb

fname     dc.w       fname_e-fname-2
          dc.b       "EX0 v1.05"
fname_e   ds.b       0
          ds.w       0

;------------------------------------------------------------------------------
; Include the various Easy PEasy include files. These give us names for
; all the various offsets, vectors, traps etc used by the PE.
;------------------------------------------------------------------------------
          in         win1_ass_pe_keys_pe
          in         win1_ass_pe_qdos_pt
          in         win1_ass_pe_keys_wwork
          in         win1_ass_pe_keys_wstatus
          in         win1_ass_pe_keys_wman
          in         win1_ass_pe_keys_wdef


;------------------------------------------------------------------------------
; Define a few explicit equates for this example program. These are
; offsets into the program's dataspace (relative to A6) where we store
; various bits of useful information, channel ids and so on.
;------------------------------------------------------------------------------
id        equ        0
wmvec     equ        4
slimit    equ        8                 ; Size - origin
```

Listing 26.9: Ex0 - Standard Job Header

The above is the usual QDOSMSQ job header, and so on. The various IN lines pull in the include files from Easy PEasy.

```
;------------------------------------------------------------------------------
; Here is when the example code really starts.
;------------------------------------------------------------------------------
start     lea        (a6,a4.l),a6      ; Dataspace in A6
          bsr.s      ope               ; Open a con channel
          move.l     a0,id(a6)         ; Keep the ID safe
          moveq      #iop_pinf,d0      ; Find Pointer Environment & WMAN
          moveq      #-1,d3            ; Timeout
          trap       #3                ; Do it
          tst.l      d0                ; Did it work?
          bne        sui               ; Failed, or PE absent, bale out
          move.l     a1,wmvec(a6)      ; Keep WMAN vector safe too
          beq        sui               ; WMAN not present, bale out
          movea.l    a1,a2             ; Copy WMAN vector to A2
          lea        slimit(a6),a1     ; Buffer for results
          moveq      #iop_flim,d0      ; Find maximum size of window
```

```
        trap     #3                 ; Do it
        subi.l   #$C0008,(a1)       ; Less 12, 8 from width, height
        lea      wd0,a3             ; Address of main window definition
        move.l   #ww0_0,d1          ; Size of working definition
        bsr      getsp              ; Return ALCHP'd address in A0
        movea.l  a0,a4              ; Copy to A4.
```

Listing 26.10: Ex0 - Initialisation

The section of code above carries out various initialisations and checks for the Pointer Environment and WMAN before allocating enough space for the working definition for the main window which SETW stores for us in ww0_0.

```
;---------------------------------------------------------------------
; We need to set the status area to zeros
; and the loose items to "available" (zero).
;---------------------------------------------------------------------
        lea      wst0,a1            ; Status area address
        movea.l  a1,a0              ; Copy to A0
        moveq    #wst0_e-wst0-1,d1  ; Bytes to clear - 1

st1     clr.b    (a0)+              ; Set status to zero/available
        dbra     d1,st1             ; And repeat

        movea.l  id(a6),a0          ; Get the channel ID again
        move.l   wd_xmin+wd_rbase(a3),d1 ; Minimum size (x,y) in D1
        andi.l   #$0FFF0FFF,d1      ; Lop off the scaling factors
;                                   ; Wm_setup gets upset if you leave
;                                   ; scaling stuff attached. The x,y
;                                   ; sizes in D1 must be actual sizes.
        jsr      wm_setup(a2)       ; Set up the working definition
```

Listing 26.11: Ex0 - Loose Item Initialisation

Just before we (finally) set up the window, we need to be sure that all the loose items are set to available - in this case - and that the status area is filled with zeros. As ever, SETW has put the status area details in an easy to find location - wst0 - and we use this to initialise the status area easily. Regardless of the actual size of the status area itself, the above code will always work.

Please note, in the above George picks the smallest window definition as the one to use when the program first starts. The size of the smallest definition is obtained from wd_xmin + wd_rbase(a3) and placed in D1.L with the high word containing the width and the low word holding the height. Because this definition has scaling details embedded in the top nibble of each word, these must be masked out before calling `wm_setup`.

The same applies if you set D1.L to zero - which means *use the default (largest) definition* - unless the scaling factors are masked off, the call to `wm_setup` will return, but your window will not display correctly, if at all. This problem also affects the `wm_fsize` routine which returns the size, in D1.L, for a given definition. You *must* mask off the scaling nibbles.

```
        moveq    #-1,d1             ; Set the window position ...
        jsr      wm_prpos(a2)       ; ... to where the pointer is
        jsr      wm_wdraw(a2)       ; Draw the contents
```

Listing 26.12: Ex0 - Position and Draw Window

The snippet of code above sets the window position to be where the pointer is on screen right now, then draws the window.

```
wrpt        jsr         wm_rptr(a2)        ; Read the pointer
```

Listing 26.13: Ex0 - Reading the Pointer

The above starts the pointer reading loop. This code will not return unless an action routine sets D0 with an error code, or, sets D4 with an event number.

```
            beq.s       no_err             ; As D0 is zero, D4 must be non zero
            bra         sui                ; Error, D0 is non zero, bale out
```

Listing 26.14: Ex0 - Test for Errors or Events

If we have returned from the read pointer loop, then D0 is holding an error code, or D4 holds an event number. Because the Status Register must hold the flags according to the value in D0 on exit from an action routine, checking for the Z flag being set implies that D0 is indeed holding an error.

If no error is detected, the code skips off to a label no_err below, where D4 is checked for events to process, otherwise, the program dies horribly with a call to the sui routine supplied by George.

```
;----------------------------------------------------------------------------
; Default console channel definition.
;----------------------------------------------------------------------------
con         dc.w        3
            dc.b        'con'

;----------------------------------------------------------------------------
; Routine to open a channel for this job.
;----------------------------------------------------------------------------
ope         lea         con,a0             ; To open "con"
            moveq       #-1,d1             ; For this job
            moveq       #0,d3
            moveq       #io_open,d0
            trap        #2
            rts
```

Listing 26.15: Ex0 - Console Channel Details & Code

The code above defines a console channel for our program and opens it.

```
no_err      movea.l     (a4),a1            ; Status area
            btst        #pt__can,wsp_weve(a1) ;Was it a CANCEL event?
            bne         sui                ; Yes, exit

            btst        #pt__move,wsp_weve(a1) ; Was it a MOVE event?
            beq.s       no_er1             ; No, skip
            bsr         move               ; Yes, process a MOVE
            bra.s       wrpt               ; Read pointer again

no_er1      btst        #pt__wsiz,wsp_weve(a1) ; Was it a SIZE event?
            beq         no_er2             ; No, skip
            bsr.s       resze              ; Yes, process a SIZE
            bra.s       wrpt               ; Read pointer again

no_er2      btst        #pt__zzzz,wsp_weve(a1) ; Was it a SLEEP event?
            beq.s       wrpt               ; No, read the pointer again
            move.l      #ww0_1,d1          ; Get main window button size
            move.l      #ww0_0,d2          ; Get main window size
            bsr         sleep              ; Process a SLEEP
```

```
        bra.s    wrpt               ; Read pointer again
```

Listing 26.16: Ex0 - Checking Events

The code above is executed on return from the read pointer loop with an event number in D4. It begins by checking to see if the CANCEL event occurred (or was set in an action routine) and if so, exits the program via the sui routine.

Assuming that the event was not CANCEL, the next check is for a MOVE event. If it was a MOVE, the move is handled by George's move routine and we return to the read pointer loop again.

The next check is for a SIZE event and if detected, we process the MOVE request and return to the pointer reading loop, otherwise we skip to the final check.

The last check we make is for a SLEEP event. If this is not a SLEEP request, we skip back and begin reading the pointer again. It this is a SLEEP request, we set the registers as required by the sleep routine by loading D1 with the current window size and D2 with the button window size - both helpfully defined by SETW - and jump into the sleep routine.

The sleep routine returns control to our code again and we skip back to reading the pointer. We must do this or we will never be able to know when the sleeping program has been wakened etc.

All of the above checks were made by looking at the individual bits in the window byte of the event vector.

```
;------------------------------------------------------------------------------
; Loose item action routines.
;------------------------------------------------------------------------------
; MOVE
;------------------------------------------------------------------------------
afun0_0 bsr      move               ; Process a MOVE

af1     move.w   wwl_item(a3),d1    ; Loose item number
        move.b   #wsi_mkav,ws_litem(a1,d1.w) ; Ask for redraw
        moveq    #-1,d3             ; Selective redraw
        jsr      wm_ldraw(a2)       ; Redraw loose items
        clr.b    ws_litem(a1,d1.w)  ; Available status
        moveq    #0,d4              ; No events
        moveq    #0,d0              ; No errors
        rts                         ; Read the pointer again
```

Listing 26.17: Ex0 - Move Loose Item Action Routine

The program demonstrates both methods of handling loose item action routines. MOVE and SIZE are handled within the read pointer loop and not by the above code which checks the event bits outside of the read pointer loop.

The action routine above, for a MOVE, carries out all the processing necessary to make the window move on screen. It simply calls the move routine supplied by George.

The code at label af1 is necessary as it resets the loose item's status to available - when a loose item is hit or done, it's status changes to selected. Once this has been done and the loose item redrawn, D4 and D0 are set to tell the read pointer loop to continue, the action has been processed.

```
;------------------------------------------------------------------------------
; RESIZE
;------------------------------------------------------------------------------
afun0_1 move.l   a3,-(a7)           ; Save working register
```

```
        movea.l    ww_wdef(a4),a3   ; Window definition x,y size
        bsr.s      resze            ; Process a SIZE
        movea.l    (a7)+,a3         ; Restore pointer to loose item
        bra.s      af1              ; And reset status etc
```

Listing 26.18: Ex0 - SIZE Loose Item Action Routine

The action routine above processes a SIZE request when the Size Loose Item is hit or done. It does this by calling code common to the action routine itself and called by the user level code (outside the pointer reading loop) when a SIZE event bit is set in the window event vector.

Unfortunately, there is no Easy PEasy way to do a resize (at least, not at the moment) so we programmers have to do it all ourselves. As shown below.

```
;------------------------------------------------------------------------
*  To perform the resize we need to
*     a. Find the amount of resize (by wm_chwin)
*     b. Throw away the current working definition (by wm_unset)
*     c. Find the new size (by wm_fsize)
*     d. Get space for the new working definition (by getsp)
*     e. Set up the new working definition  (by wm_setup)
*     f. Position the new window  (by wm_prpos)
*     g. Draw the contents  (by wm_wdraw)
*
* Comments
*   On a.
*    We have to set the resize bit in the window byte of the
*    event vector in the status area before wm_chwin is called.
*    The change in size is returned in D1 and the window size
*    event number is returned in D4.
*
*   On c.
*    On entry to wm_fsize, D1 must contain the requested size.
*    This size must be chosen carefully. It must be no bigger
*    than the maximum in the window definition. It must be
*    smaller than the maximum size for the window layout. The
*    x-size must be a multiple of 4 (to allow proper stippling.
*    Finally the size must not be bigger than the current screen
*    size with allowance for shadow and border.
*    On exit D1 contains the actual size and D2.W contains the
*    number of the repeated section.
*    In this example we do not really need to use wm_fsize since
*    we know that D2.W will be zero and that the value in D1
*    will be that on entry (since we have a variable window).
*
*   On d.
*    The space needed is found from the label ww0_0 set in the
*    window definition.
*
*   On e.
*    For wm_setup we need on entry:
*      D1 = size
*      A0 = channel ID
*      A1 -> status area
*      A3 -> window definition
*      A4 -> space for working definition
*
```

```
*   On f.
*    On entry to wm_prpos we need the position in D1.
*    In order to ensure that the bottom right corner of the
*    resized window is in the same position as that of the old
*    we need to subtract the increase in size from the pointer
*    origin in the old window.
*    The new position is thus wd_org plus ww_xsize minus the
*    new size.
*
;------------------------------------------------------------------------
resze    move.l    ww_xorg(a4),d7
         move.l    ww_wdef(a4),a5              Window def
         add.l     wd_xorg(a5),d7
         add.l     ww_xsize(a4),d7             Ptr pos for PRPOS (optr)
         bset      #pt__wsiz,wsp_weve(a1)
         jsr       wm_chwin(a2)                Sets change to D1 (mv)
         bclr      #pt__wsiz,wsp_weve(a1)
         move.w    wd_rbase+wd_xmin(a5),d5
         andi.w    #$fff,d5
         move.w    ww_xsize(a4),d4
         swap      d1
         sub.w     d1,d4
         cmp.w     d5,d4
         bgt.s     resze1                      D4 is greater
         move.w    d5,d4

resze1   move.w    wd_xsize(a5),d5
         cmp.w     d5,d4
         blt.s     resze2                      D4 is smaller
         move.w    d5,d4

resze2   moveq     #3,d3
         add.w     d4,d3
         andi.w    #$fffc,d3                   Keep answer in D3.W  (mv)
         move.w    wd_rbase+wd_ymin(a5),d5
         andi.w    #$fff,d5
         move.w    ww_ysize(a4),d4
         swap      d1
         sub.w     d1,d4
         cmp.w     d5,d4
         bgt.s     resze3                      D4 is greater
         move.w    d5,d4

resze3   move.w    wd_ysize(a5),d5
         cmp.w     d5,d4
         blt.s     resze4                      D4 is smaller
         move.w    d5,d4

resze4   swap      d3
         move.w    d4,d3                       D3 = new mv x | y
         jsr       wm_unset(a2)
         bsr       rechp

;------------------------------------------------------------------------
; Now restrict size to the screen size less (12,8)
;------------------------------------------------------------------------
```

```
        move.l  slimit(a6),d1
        cmp.w   d3,d1
        ble     resze7                  D1 OK
        move.w  d3,d1

resze7  swap    d1
        swap    d3
        cmp.w   d3,d1
        ble     Resze8                  D1 OK
        move.w  d3,d1                   New size

resze8  swap    d1                      New limited size
        move.l  d1,d3
        jsr     wm_fsize(a2)
        move.l  d1,-(a7)                Keep size pro tem
        move.l  #ww0_0,d1               Space needed
        bsr     getsp
        move.l  (a7)+,d1                Replace size
        movea.l a0,a4                   New wwd
        movea.l id(a6),a0               Replace ID
        jsr     wm_setup(a2)

;------------------------------------------------------------------
; The position for PRPOS is optr-mv with minimum of 4 | 2
;------------------------------------------------------------------
        move.l  d7,d1
        swap    d1
        swap    d3
        sub.w   d3,d1
        cmpi.w  #4,d1
        bge.s   resze5                  D1 not less than 4
        move.w  #4,d1                   Set minimum of 4

resze5  swap    d1
        swap    d3
        sub.w   d3,d1
        cmpi.w  #2,d1
        bge.s   resze6                  D1 not less than 2
        move.w  #2,d1                   Set minimum of 2

resze6  jsr     wm_prpos(a2)
        jmp     wm_wdraw(a2)
```

Listing 26.19: Ex0 - SIZE Processing

The above code is George's way of processing a SIZE request from within an action routine or from user code that detected the SIZE bit set in the window event vector.

The other two action routines, for SLEEP and ESC, demonstrate how an action routine can simply set the appropriate bit in the window vector, set D4 to indicate an event and exit with D0 cleared.

In this case, the actions cause the pointer reading loop to return to the user's code where the events can be checked for (see above) and processed accordingly.

```
;------------------------------------------------------------------
; EXIT - set the CANCEL event in the windows event vector, put the
; CANCEL event number in D4 and exit with D0 set to zero.
```

```
;------------------------------------------------------------------------------
afun0_3   bset      #pt__can,wsp_weve(a1) ; Set CANCEL window event
          moveq     #pt__can,d4         ; ESC event number
          moveq     #0,d0               ; No errors
          rts                           ; Return to exit from reading the
;                                       ; pointer and into the PROCESS EVENT
;                                       ; section of the user's code.
```

Listing 26.20: Ex0 - EXIT Loose Item Action Routine

First of all, the action routine for the ESC loose item. This is the simplest action routine as it only has to set the event bit, set D4 and D0 then exit. It doesn't have to reset the ESC loose item status from selected back to available because the program is about to exit and the user will never see the redrawn loose item. Simple.

```
;------------------------------------------------------------------------------
; SLEEP - set the ZZZ event bit in the window event vector, put the ZZZ
; event number in D4, redraw the ZZZ loose item as available - else it
; is still selected when we wake from the button frame - then exit with
; D0 set to zero.
;------------------------------------------------------------------------------
afun0_2   move.w    wwl_item(a3),d1     ; Item number for the ZZZ loose item
          move.b    #wsi_mkav,ws_litem(a1,d1.w) ; Redraw as available
          moveq     #-1,d3              ; Selective redraw
          jsr       wm_ldraw(a2)        ; Redraw loose items
          clr.b     ws_litem(a1,d1.w)   ; Available status set
          bset      #pt__zzzz,wsp_weve(a1) ; Set ZZZ window event.
          moveq     #pt__zzzz,d4        ; ZZZ event number
          moveq     #0,d0               ; No errors
          rts                           ; Return to exit from reading the
;                                       ; pointer and into the PROCESS EVENT
;                                       ; section of the user's code.
```

Listing 26.21: Ex0 - SLEEP Loose Item Action Routine

The sleep loose item's action routine is almost as simple, but because the program will - hopefully - be awakened at some point, it has to reset the loose item status and redraw it.

The code above starts off by obtaining the correct loose item number and changing it's status to indicate that it is available. It then calls `wm_ldraw` to redraw only those loose items asking for a status change & redraw - as signalled by the value of minus one in D3. This prevents redrawing up to 32 loose items which don't need redrawing because nothing has changed.

Once redrawn, the loose item's status is set to available as well, the SLEEP bit is set in the window event vector, D4 is set to show the event number and we exit with D0 cleared to show that no errors occurred.

On return from the above two action routines, the read pointer loop will exit and processing will continue from the beq.s no_err just after the wrpt label. (Many lines above!)

```
;------------------------------------------------------------------------------
; Pull in window definition as created by SETW.
;------------------------------------------------------------------------------
          in        win1_ass_pe_EX0w_asm

;------------------------------------------------------------------------------
; Pull in the Easy PEasy stuff next - code routines and sprites.
;------------------------------------------------------------------------------
```

```
        in        win1_ass_pe_peas_sym_lst
        lib       win1_ass_pe_peas_bin

        in        win1_ass_pe_csprc_sym_lst
        lib       win1_ass_pe_csprc_bin
```

Listing 26.22: Ex0 - Includes and Libraries

The last few lines of code pull in the SETW defined window from the file EX0w_asm, then LIBs in the Easy PEasy routines and the various sprites that your program might want to use.

26.6 Coming Up...

The next chapter in the ongoing saga of writing PE programs in assembler, will concentrate on application windows and their window menus.

VIII

The Pointer Environment - Continued

27. The Return of WMAN

27.1 Introduction

Before my recent delving into George's SETW, his Easy PEasy utilities and my brief foray into pdf magazine production, I was walking through the various requirements of setting up a window definition using WMAN. This edition continues from where we left off but incorporates George's utilities into the examples. As George has made it easy to write Pointer Environment programs, I think we should make use of his hard work. Code reuse is all the rage!

However, before we start coding, we better take a moment to discuss Application Sub-Windows.

27.2 Application Sub-Windows

There are a number of uses for application sub-windows, for the program's display or to hold a menu, correctly known as an *application window menu*. An application window is a variable thing and as such, can be defined in a number of ways. Because of this, and because an application can have multiple application sub-windows, when defining our main window (or the variable parts of our main window) we don't have a pointer to an application window. Instead, we have a pointer to a list of pointers and each of these, in turn, points to an application window definition. The list is terminated by a zero word.

The following is the relevant extracts from a definition of a main window which contains information windows, loose items and a pair of application sub-windows.

```
; Main window definition :
            ...
            dc.w   info_list-*        ; Pointer to information window list
            dc.w   loos_list-*        ; Pointer to list of loose items
            dc.w   appw_list-*        ; Pointer to list of app windows
            ...

```

```
; Application window list :
appw_list   dc.w   appw_0-*          ; Pointer to 1st sub-window defn
            dc.w   appw_1-*          ; Pointer to 2nd sub-window defn
            dc.w   0                 ; No more app sub-windows
```

Listing 27.1: Example Application Sub-Window List Definition

The definition of an application sub-window is described below.

```
; Application sub-window definition :
appw_0      dc.w   192               ; Width in pixels (+ scaling)
            dc.w   119               ; Height in pixels (+ scaling)
            dc.w     4               ; X org relative to 0 in main window
            dc.w    18               ; Y org relative to 0 in main window
            dc.b   256               ; Bit 7 set = clear window
            dc.b     0               ; Shadow depth - must be 0!
            dc.w     1               ; Border width
            dc.w     0               ; Border colour
            dc.w     7               ; Paper colour
            dc.w     0               ; Pointer to pointer sprite, or 0
            dc.w     0               ; User defined setup routine, or 0
            dc.w     0               ; User defined drawing routine, or 0
            dc.w     ahit0-*         ; Hit routine
            dc.w     0               ; Sub-window control routine, or 0
            dc.w     0               ; Max X control sections (splits)
            dc.w     0               ; Max Y control sections (splits)
            dc.b     9               ; Selection key
            dc.b     0               ; Spare byte - must be 0
            ...
```

Listing 27.2: Example Application Sub-Window Definition

For our current needs, this definition allows us to have a simple application sub-window with no pan and scroll control bars and no menu. The user defined setup and drawing routines are most often defined as zero words to allow WMAN to do the hard work.

27.3 Application Sub-Window Hit Routines

The hit routine for an application sub-window is called from within the `wm_rptr` call either when you HIT in the window, or when you press the selection key for that sub-window. Similar to loose item action routines previously discussed, if the code exits with D0 set to zero, the `wm_rptr` call will resume again - in other words, control will not return to your own code just yet - unless the hit code sets any event bits in the event vector. This is slightly different from loose item action routines in this respect.

On entry to a sub-window hit routine various registers are set with specific parameters as defined in Table 27.1

Hit routines should exit with D1, D5 - D7, A0 and A4 preserved to the same value that they had on entry to the routine. D2, D4, A1 - A3, A5 and A6 are undefined on exit (which means that they don't care what value they have.) The hit code must set the SR according to the value in D0 on exit.

D3, on return from a hit routine, should normally be returned as per its value on entry. It is not used by `wm_rptr` however, it is used by `wm_rptrt` (read pointer with return on timeout) from WMAN 1.5 onwards. Wm_rptr ignores the upper word of D3. If your read pointer loop is using

Register	Description
D1.L	High word = pointer X position. Low word = pointer Y position in *absolute* screen coordinates. Ie, the pointer position within the *entire screen* and *not* within the program's window or the application sub-window itself.
D2.W	Selection keystroke letter, in its upper cased format, or 1 = Hit/SPACE or 2 = DO/ENTER. If D2 is -1, then the application sub-window was "hit" by an external keystroke. Zero indicates no key was pressed.
D4.B	An *event number*. This can only be 0, pt__do (16) or pt__cancel (17) as all other events are handled by WMAN. If you have a loose item with ESC as the selection keystroke, then the loose item action routine will catch the ESC keystroke - the application sub-window hit routine will not see it if the ESC causes the program to exit.
A0.L	Channel id.
A1.L	Pointer to the status area.
A2.L	WMAN vector.
A3.L	Pointer to sub-window definition.
A4.L	Pointer to window working definition.

Table 27.1: Application Sub-Window Hit Routine - Registers

the `wm_rptrt` vector instead, and you have changed the value of D3 within the hit code, you must clear the high word on exit.

It is important to note that WMAN *doesn't* set the event bits for you, it is up to the hit code to do that for you. For example, if someone HITs the application window then the hit routine will be called with D2 = 1 which is also the case also when someone DOes the application window but the pt__do bit in the window byte of the event vector will *not* be set.

On exit, if D0 is clear and the status (Z) bit is set, control will return to the `wm_rptr` loop and not to your application's code. To return to your own code, the hit routine needs to set at least one event bit in the event vector.

If an error is detected within the hit code, then it should exit with the appropriate error code in D0 and the status register set accordingly.

27.4 Example Application Window

As before, we now create a useful (!) demonstration program to show us the simplest use of an application sub window. The program will look like the following when completed and running:

You can see from Figure 27.1 how I'm sticking to accepted QDOSMSQ design standards here can't you!

The window above consists of the following:

- An outline with white paper, a black single pixel border and a shadow. The default arrow sprite is used for the entire window.
- A 'caption bar' consisting of a single information window with green/white striped paper (paper colour 92).
- Within the information window is a single information text object which is simply the program name.

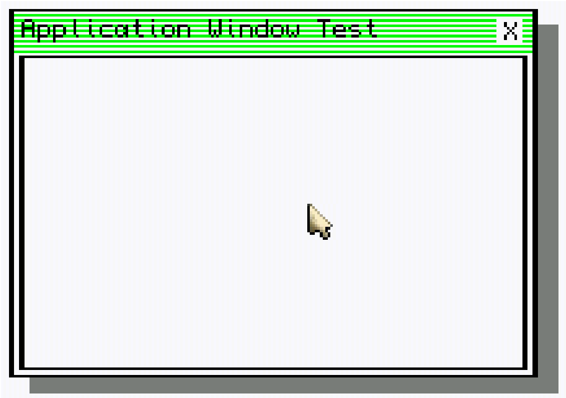


Figure 27.1: Application Window Test

- Also located within the information window is one loose item containing a text object ('X') and this has the keypress code set up to close the window.
- The remainder of the outline is filled with an application window, with white paper and a black single pixel border. (No shadow - they are forbidden for application windows!). This window also uses the default arrow sprite and has a selection key of TAB. This means that if you press the TAB key, the pointer will jump into the application sub-window.

The window was set up using SETW as follows:

1. When prompted for 'name$' enter *ApplTestWin*.
2. On the 'Alter Text' screen.
    - Press N for new, type 'X' (without the quotes) then ENTER.
    - Press N for new, type 'Application Window Test' (without the quotes) then ENTER
    - Press ESC.
3. On the 'Alter Sprite' screen.
    - Press ESC.
4. On the 'Alter Blob' screen.
    - Press ESC.
5. On the 'Alter Patt' screen.
    - Press ESC.
6. Number of main windows = 1
7. Number of Loose Items = 1
8. Number of Information windows = 1
9. Number of IW Objects = 1
10. Number of application windows = 1
11. Application windows menu items = 0
12. For main window 1:
    - Shadow = 2
    - Border size = 1
    - Border colour = colour_ql -> black
    - Paper colour - colour_ql -> white
    - Sprite = arrow
13. Loose Items:
    - Press N for 'system palette defaults'
    - Confirm N when prompted again for defaults
    - Border size = 1

  - Border colour = colour_ql -> black
  - Unavailable background = colour_ql -> white
  - Unavailable Ink = colour_ql -> grey
  - Available background = colour_ql -> white
  - Available Ink = colour_ql -> black
  - Selected background = colour_ql -> green
  - Selected Ink = colour_ql -> black

14. Loose Item 1:
    - Type = text
    - Object -> select the ‘X’ text object
    - Selection key = ESC
15. Information Window 1:
    - Border size = 0
    - Paper = colour_ql -> No 92
16. Object 1:
    - Type = text
    - Object -> select the ‘Application Window Test’ text object.
    - Colour = colour_ql -> black
    - Xcsize = 0
    - Ycsize = 0
17. Application Window 1:
    - Border size = 1
    - Border colour = colour_ql -> black
    - Paper colour = colour_ql -> white
    - Sprite = arrow
    - Selection key = TAB
18. Main window size: (Use the arrow keys to change the size, press ENTER when correct)
    - Width = 200
    - Height = 140
    - Do you want a variable window = N
    - Set the origin to 0,0 (Press ENTER when correct)
19. Loose Item 1: (Toggle hit/position with F2. Press ENTER when correct)
    - Hit size = 10 x 10
    - Position = 186 x 3
20. Information Window 1: (Toggle size/position with F2. Press ENTER when correct)
    - Size = 200 x 16
    - Position = 0 x 0
    - Object position = 2 x 2
21. Application Window 1: (Toggle size/position with F2. Press ENTER when correct)
    - Size = 192 x 119
    - Position = 4 x 18

When you have completed this procedure, and SETW has exited, you should save the file ram1_ApplTestWin_asm to a safer place. The file should look like the following, although I have added some extra comments to my copy of the generated code.

```
; ApplTestWin_asm

; Undefined Labels - need to be defined elsewhere in my own code.
;      ahit0    - application window 0 hit action routine.
;      afun0_0 - Loose item 0 hit action routine.
```

```

; Labels for External Use
;       wst0    - Window status area
;       wd0     - Window definition address
;       ww0_0   - Window default size
;       ww0_1   - Window button size

SYS_SPR  dc.w       0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
         dc.w       17,18,19,20,21,22,23,24,25,26,27,28,29,30
         dc.w       31,32,33,34,35,36,37


; Definition of all text objects here

txt0     dc.w       txt0_e-2-txt0
         dc.b       "X"
txt0_e   ds.b       0
         ds.w       0

txt1     dc.w       txt1_e-2-txt1
         dc.b       "Application Window Test"
txt1_e   ds.b       0
         ds.w       0


; Application window list.
app_list0
         dc.w       appw0-*
         dc.w       0


; Application windows 0 definition.
appw0    dc.w       192        xsize
         dc.w       119        ysize
         dc.w       4          xorg
         dc.w       18         yorg
         dc.w       256        flag
         dc.w       1          borw
         dc.w       0          borc
         dc.w       7          papr
         dc.w       0          pspr *
         dc.w       0          setr *
         dc.w       0          draw *
         dc.w       ahit0-*    hit *
         dc.w       0          cntrl *
         dc.w       0          nxsc
         dc.w       0          nysc
         dc.b       9          skey
         dc.b       0          spr1

; Information Object(s)
pobl0    dc.w       138        xsize
         dc.w       10         ysize
         dc.w       2          xorg
         dc.w       2          yorg
         dc.b       0          type
```

```
            dc.b        0           spar
            dc.l        0           Ink, xcsize,ycsize
            dc.w        txt1-*      pobj *
            dc.w        -1

; Information window(s)
infw0       dc.w        200         xsize
            dc.w        16          ysize
            dc.w        0           xorg
            dc.w        0           yorg
            dc.w        0           flag
            dc.w        0           borw
            dc.w        526         borc
            dc.w        92          papr
            dc.w        pobl0-*     pobl *
            dc.w        -1          end

; Loose item(s)
litm0       dc.w        10,10       xsize, ysize
            dc.w        186,3       xorg, yorg
            dc.b        0,0         xjst, yjst
            dc.b        0,3         type, skey
            dc.w        txt0-*      pobj *
            dc.w        0           item
            dc.w        afun0_0-*   pact *
            dc.w        -1          end

litm1       dc.w        16404,12    xsize, ysize
            dc.w        0,0         xorg, yorg
            dc.b        0,0         xjst, yjst
            dc.b        0,0         type, skey
            dc.w        0           pobj *
            dc.w        0           item
            dc.w        0           pact *
            dc.w        -1          end

; Window definition
wd0         dc.w        200         xsize
            dc.w        140         ysize
            dc.w        0           xorg
            dc.w        0           yorg
            dc.w        258         flag
            dc.w        1           borw
            dc.w        0           borc
            dc.w        7           papr
            dc.w        0           sprt *
            dc.w        1           curw
            dc.w        0           curc
            dc.w        7           uback
            dc.w        255         uink
            dc.w        0           ublob *
            dc.w        0           upatt *
            dc.w        7           aback
            dc.w        0           aink
            dc.w        0           ablob *
            dc.w        0           apatt *
```

```
          dc.w      4         sback
          dc.w      0         sink
          dc.w      0         sblob *
          dc.w      0         spatt *
          dc.w      0         help
          dc.w      200       xsize
          dc.w      140       ysize
          dc.w      infw0-*   pinfo *
          dc.w      litm0-*   plitem *
          dc.w      app_list0-*   pappl *
          dc.w      16384     xsize
          dc.w      12        ysize
          dc.w      0         pinfo *
          dc.w      litm1-*   plitem *
          dc.w      0         pappl *
          dc.w      -1

; Sizes
ww0_0     equu      290
ww0_1     equ       148

; Status Areas
wst0
          ds.b      65
wst0_e    ds.b      0
          ds.w      0
```

Listing 27.3: Test Window - ApplTestWin_asm

27.5 Example Program

Having defined our application window test definition, we need a program to run it. However, we must also decide what the program is intended to do when running. As this is our first program with an application window, we will simply write some information to the application window when ‘things’ happen.

I mentioned ‘code reuse’ above, so the following is based very heavily on George’s example code, with (I hope) all the unnecessary bits removed. Unnecessary, that is, for this example of mine!

The following is enough of a test harness to get our newly designed window up and running, but only the ESC key and the ‘X’ loose item works. The rest of the program will come later.

In the source code that follows, where I use the ‘in’ or the ‘lib’ commands, you will need to replace ‘win1_georgegwilt_’ with the location of the files being included. Unless you have exactly the same source setup as I do!

We start, as ever, with a standard QDOSMSQ job header and then pull in the various include files from Easy PEasy. Three offsets into the job’s data area are then defined.

```
        bra.s   start
        dc.l    0
        dc.w    $4afb

fname   dc.w    fname_e-fname-2
        dc.b    "Application Window Test 1"
fname_e ds.b    0
```

```
        ds.w    0

; We need the various equates files etc.

        in      win1_georgegwilt_peass_keys_pe
        in      win1_georgegwilt_peass_qdos_pt
        in      win1_georgegwilt_peass_keys_wwork
        in      win1_georgegwilt_peass_keys_wstatus
        in      win1_georgegwilt_peass_keys_wman
        in      win1_georgegwilt_peass_keys_wdef

id      equ 0                       ; Channel id storage
wmvec   equ 4                       ; WMAN vector storage
slimit  equ 8                       ; IOP_FLIM results - 4 words ...
;                                   ; X-size, Y-size, X-org, Y-org
```

Listing 27.4: ApplTest_asm - Standard Job Header & Equates

Following on from the above, we have the job's start and initialisation code. As the vast majority of this has been explained before in the introductory article on George's Easy PEasy, I shall not go into it again here. See *Easy PEasy Part 1* in *Volume 14 Issue 3* for full details.

```
start   lea     (a6,a4.l),a6        ; Make A6 point to the job's dataspace
        bsr     op_con              ; Open a con channel
        move.l  a0,id(a6)           ; And store the channel id
        moveq   #iop_pinf,d0        ; Trap to get Pointer Information
        moveq   #-1,d3              ; Timeout
        trap    #3                  ; Do it
        tst.l   d0                  ; Is ptr_gen present?
        bne     sui                 ; No, bale out via SUI
        move.l  a1,wmvec(a6)        ; Yes, store the WMAN vector
        beq     sui                 ; Oops! WMAN wasn't actually found

flim    movea.l a1,a2               ; The WMAN vector is required in A2
;                                   ; The channel id is already in A0
        lea     slimit(a6),a1       ; Result buffer
        moveq   #iop_flim,d0        ; Query maximum size of window
        moveq   #0,d2               ; D2 must be zero
;                                   ; D3 is preserved timeout from above
        trap    #3                  ; Do it (No errors)
        tst.l   d0                  ; Did it work?
        bne     sui                 ; No, exit via SUI

        subi.l  #$C0008,(a1)        ; Adjust max height & width for shadow
;                                   ; and borders.
        lea     wd0,a3              ; Get address of window definition
        move.l  #ww0_0,d1           ; Get size of the working definition
        bsr     getsp               ; ALCHP memory and set A0 to address
        movea.l a0,a4               ; Which we save in A4
```

Listing 27.5: ApplTest_asm - Initialisation

So far so good. Next we use a generic piece of code to go through the status area and set all the lose item status bytes to 'available'.

```
        lea     wst0,a1             ; Status area starts here
        movea.l a1,a0               ; Copy to A0
```

```
        moveq #wst0_e-wst0-1,d1  ; How many bytes to clear - 1

st_clr  clr.b (a0)+              ; Clear one byte
        dbf d1,st_clr            ; Then the remainder
```

Listing 27.6: ApplTest_asm - Loose Item Initialisation

At this point we are just about ready to go. So, the next piece of code will call out the various WMAN routines to setup the window definition, position it on screen where the pointer currently is located and draw it before vanishing into the twilight zone that is the read pointer loop within WMAN. All of these have been described before so I don't go into detail.

```
        movea.l id(a6),a0        ; Get the channel id where we need it
;                                  A1 is the status area address
;                                  A3 is the window definition address
;                                  A4 is the working definition address
        move.l wd_xmin+wd_rbase(a3),d1 ; Get the minimum dimensions
        andi.l #$FFF0FFF,d1      ; Mask off any scaling factors
        jsr wm_setup(a2)         ; Set up the working definition

        moveq #-1,d1             ; Draw the window where the pointer is
        jsr wm_prpos(a2)         ; Position it as a primary window
        jsr wm_wdraw(a2)         ; Draw the contents
wrpt    jsr wm_rptr(a2)          ; Enter the "read pointer" loop in WMAN
```

Listing 27.7: ApplTest_asm - Window Creation & Display

At this point, WMAN takes over and we never get beyond the above code unless an event is detected - or set in an action routine - or an action routine flags an error. You will learn more about this as we add some meat to this programs workings later on.

The following code will exit from the program if an error occurred. The Z flag is already set or unset according to the value in D0.

Note Of course, the following checks only work if the application sub-window hit routine, or the various loose item action routines set D4 first, then D0. Otherwise, we need to make sure that the Z flag is set according to D0 before we test it.

```
        beq.s no_err             ; Since D0 is zero D4 is non zero
        bra sui                  ; An error occurred exit via SUI
```

Listing 27.8: ApplTest_asm - Error Handling

If we are here, we need to check for any events that may have been detected or set in an action routine. In this example, we don't check every event, only the CANCEL event caused by the ESC key being pressed.

On return from the `wm_rptr` call, A0 is the channel id and A4 is the working definition address.

```
no_err  movea.l (a4),a1          ; Status area address
        btst #pt__can,wsp_weve(a1) ; Check for CANCEL event
        bne sui                  ; Exit

        bra.s wrpt               ; No more events, read pointer again
```

Listing 27.9: ApplTest_asm - Event Handling

As mentioned above, this example currently is only checking for the ESC key being pressed. However, we have a loose item that can also be clicked to escape from the program. Rather than handling the ESC key and the loose item separately, we simply set the CANCEL event within the loose item action routine and let WMAN take care of it by passing control out of the read pointer loop into the above event handling code.

The following is the loose item action routine to do this.

```
; Loose item action routine

afun0_0 bset    #pt__can,wsp_weve(a1) ; Set the CANCEL event bit
        moveq   #pt__can,d4           ; CANCEL event number
        moveq   #0,d0                 ; No errors
        rts                           ; Exit here and exit from wm_rptr too
```

Listing 27.10: ApplTest_asm - ESC Loose Item Action Routine

Because we have an application window defined, then the following is the default application window hit routine. When you hit the application window, or press the TAB key, the following code is executed. The default simply sets the registers to show no errors, no events and returns control back to the `wm_rptr` loop.

```
; Application sub-window hit routine

ahit0   moveq   #0,d4
        moveq   #0,d0
        rts
```

Listing 27.11: ApplTest_asm - Application Window HIT Routine

Note

The loose item action routine and application window hit routine names, `afun0_0` and `ahit0` are hard coded by SETW and, unless we physically edit the code generated by SETW, we must use the names SETW chooses for us.

There is a pattern to the names though, `ahit0` is the application sub-window hit routine for application sub-window zero. `Ahit1` would be the hit routine for sub-window 1 and so on. For loose items, you have a layout number and a loose item number to contend with. So `afun0_0` is for layout zero and loose item zero within that layout. `Afun0_1` is the next loose item within that layout, and so on.

Note also, if, as above, the hit routine sets D4 first, then D0 prior to exiting, the code that checks for errors need not execute an instruction to set the Z flag according to D0, it can simply test the Z flag immediately. See line 68 in Listing 27.8 above.

The remainder of the code, so far, consists of helper routines and is shown below without any further discussion.

```
; Various helper routines go here...

con     dc.w con_e-con-2              ; Size of channel definition
        dc.b 'con'
con_e   equ *

op_con  lea   con,a0                  ; We want a console
        moveq #-1,d1                  ; For this job
        moveq #0,d3                   ; Open type = "OPEN"
        moveq #io_open,d0
        trap  #2                      ; Do it
```

```
        rts
```

Listing 27.12: ApplTest_asm - Console Handling

And finally, we need to load in the window definition we generated using SETW and all the Easy PEasy code libraries supplied by George.

```
; Pull in our window definition file.

        in   win1_source_ApplTestWin_asm

; We need George's Easy PEasy code next.

        in   win1_georgegwilt_peass_peas_sym_lst
        lib  win1_georgegwilt_peass_peas_bin

; And finally, George's sprites.

        in   win1_georgegwilt_peass_csprc_sym_lst
        lib  win1_georgegwilt_peass_csprc_bin
```

Listing 27.13: ApplTest_asm - Incorporating the EasyPEasy Library

Save the code as ApplTest_asm. At least, that's what I called mine!

The code above can be assembled and executed and the window we designed a couple of issues ago will be displayed on screen. Currently it does nothing useful but if you press the TAB key when the pointer is outside the application window (but inside the main window) then you will see it jump into the application window. HITting the 'X' loose item will cause the program to exit as will pressing the ESC key.

27.6 Coming Up...

In the coming chapter we will add some code to this example to allow us to monitor events.

28. Application Sub-Windows

28.1 Introduction

Last issue I left you looking at a wonderful but practically useless program - ApplTest_asm - which displayed itself on screen and reacted only to the user clicking on a loose item or pressing the ESC key to quit the program. Apart from that, the most useful thing it did was to move the pointer into the middle of the application sub-window when the user pressed the TAB key.

This time, we get to add a little code, and see what happens when we hit an application sub-window. Let's get coding.

28.2 The Hit Routine.

You don't actually need to have a hit routine for all your application sub-windows, there's nothing wrong with setting up an application sub-window in a program and then not having a hit routine. This is especially true if you intend to display information in it rather than handle user interactions and so on. As you will see when running the code below, the hit routine for an application sub-window gets called very frequently, so if you don't need one, don't use one.

You can disable the hit routine, if you don't need or want one, by setting the pointer to the hit routine to zero in the window definition file created by SETW. Now, it's time to get down to editing our new program.

We need to copy the two files we created last time, and rename them. So, copy ApplTest_asm to ApplHitTest_asm, and ApplTestWin_asm to ApplHitTestWin_asm. These are going to be used in our first experiment.

First of all, we need to change one text object in the file ApplHitTestWin_asm, so edit that file and change the caption text to 'Application Window Hit Test' from 'Application Window Test 1'. Save the file.

Next up, we need to edit ApplHitTest_asm.

Note Where possible, line numbers reference those in the file we created in the previous chapter. Lines starting at 1, will be new ones that should be inserted. Anything else will be noted in the text.

```
5 fname    dc.w   fname_e-fname-2
6          dc.b   "Application Window Hit Test"
7 fname_e  ds.b   0
```

Listing 28.1: ApplTest_asm - New Job Name

Next we need to change the application sub-window hit routine. Look through the code for the routine named ahit0, which should be around line 83, and change it to the following.

```
83 ; Application sub-window hit routine
84
85 ahit0    movem.l d1/d3/d5-d7/a0/a4,-(a7) ; Save the workers
86          moveq   #0,d1                   ; D1.W = Application window number
87          moveq   #0,d2                   ; D2.W = Ink colour = black
88 ;                                          A4 already = Working definition
89          jsr     wm_swapp(a2)            ; Set channel id to the sub window
90
91          movem.l a1-a2,-(a7)             ; A1 & A2 get corrupted
92          lea     hit,a1                  ; Text string to print
93          move.w  ut_mtext,a2             ; Print string vector
94          jsr     (a2)                    ; Print the message
95
96          lea     hitter,a1               ; Hit counter location
97          move.w  (a1),d1                 ; Hit counter value
98          addq.w  #1,(a1)                 ; Increment counter
99          move.w  ut_mint,a2              ; Print integer vector
100         jsr     (a2)                    ; Print it
101
102         movem.l (a7)+,a1-a2             ; Restore working registers
103
104         movem.l (a7)+,d1/d3/d5-d7/a0/a4 ; Restore the workers
105
106         moveq   #0,d4                   ; No events
107         moveq   #0,d0                   ; No errors
108         rts
109
110 ; Strings and things go here
111
112 hitter   dc.w 0                         ; How many times have I been hit?
113
114 hit      dc.w hit_e-hit-2               ; Hit message
115          dc.b 'HIT: '
116 hit_e    equ *
```

Listing 28.2: ApplTest_asm - New Application Window HIT Routine

Then, right at the end, line 100, where we include our window definition, change the file name to suit our new name - from ApplTestWin_asm to ApplHitTestWin_asm.

```
100 ; Pull in our window definition file.
```

```
        in    win1_source_ApplHitTestWin_asm
```

Listing 28.3: ApplTest_asm - Including the Window Definition

Save the file. We are done.

Note in the above code, the call to `wm_swapp`. This sets the channel id in A0 to point to the application sub-window that we specify in D1. The ink colour is set according to the value in D2.W and the *sub-window is cleared*. If we don't do this call, we might still print to the application window however, if we do, it's just luck! You should *always* ensure that the channel id in A0 has been explicitly pointed to the appropriate application window before attempting to print, cls, set paper or ink, etc when executing code within a hit routine. If you don't, any text printed by the code in the hit routine may well end up writing all over a loose item, or an information window or some random place in the window.

Obviously if you are running a program that has an application sub-window that doesn't have a hit routine, you will still need to call `wm_swapp` to make sure that the channel id in A0 points to where the sub-window is on screen - assuming of course, that you wish to write text to that sub-window as part of the application.

When you assemble it and execute it, note how the counter changes as you move the pointer over and around the application sub-window. It seems that you don't have to use the left mouse button or space bar to get a HIT. In fact using the right mouse button or ENTER both add 1 to the counter. The documentation says that *If there is no keystroke, or the keystroke is not the selection keystroke for a loose menu item or an application sub-window, then, if the pointer is within a sub-window, the hit routine is called, or else the loose menu item list is searched to find a new current item.*

Press ESC to stop the program, then execute it again, try to keep the pointer outside the application sub-window. Now, press TAB. The pointer jumps into the sub-window, but what happens to the counter? It increases by two rather than one on every *subsequent* press of the TAB key.

This happens when you press some other key combinations, F1 increases the counter by two as well. Other letter or digit keys increment the counter by one.

I wonder why? Maybe, in the case of F1, the keystroke itself causes a hit and then the HELP event that the keystroke causes forces another hit? This doesn't explain the TAB key having the same effect though - that doesn't cause a hit. I mentioned above that a hit routine gets called very frequently didn't I?

Note

Ok, as an aside, I tested it. Using QMON2, I put a breakpoint at Ahit0 - the entry point for the application sub-window hit routine. On hitting the TAB key, I got a breakpoint. Looking at the registers I found that the pointer position in D1 was well outside my window limits, very strange. D2, the keystroke was set to -1 to indicate that an *external keystroke* fired the hit routine. There was no event in D4 - it was zero and D6, an undefined register was zero. So far so good, I noted down the registers and let the program continue.

It immediately stopped at the hit routine again, this time the pointer position had moved into my screen bounds from wherever it had been in hyperspace. D2 was now zero to indicate that no key has been pressed. D4 was still zero - so no events either. D6 had changed to $80. Wonder what that means?

Letting the program run again, I pressed F1 this time - without moving the pointer. Once again I hit the breakpoint. I could see the pointer position in D1 had not changed, D4 still showed no events, D2 showed no key press and D6 had returned to zero.

And again, I let the program run and it broke again. D6 was back at $80 again. D1, D2, D4 were all unchanged (as were all the other registers.)

I hit space this time, when I let QMON run the program. As expected this showed a $01 in D2 - the key press for a HIT is converted to $01, D6 was showing zero again. D4 still showed no events.

Once more, QMON let the program run and this time, I pressed ENTER. D4 showed the value $10 or the event number for a DO. The key press in D2 was set to $02 for a DO. D6 was zero and nothing else changed.

Finally, I pressed ESC to quit. Now, according to the docs, I should have stopped at Ahit0 again with D4 showing the CANCEL event, however, as documented above there was a loose item which had the ESC key set as the selection keystroke. That code was executed to exit from the program, rather than the application sub-window hit code being called with D4 set to the CANCEL event number.

28.3 The Advanced Hit Routine.

So, that's our first very simple hit test program done and dusted. It's quite simple but quite useless, all it does is show you the running total of hits in the window. You will soon get bored of it.

For our next trick, we shall improve the utility to display full details of what data gets passed to the hit routine. Copy ApplHitTest_asm to ApplHitTest_2_asm, and ApplHitTestWin_asm to ApplHitTestWin_2_asm.

As before, we need to change one text object in the file ApplHitTestWin_2_asm, so edit that file and change the caption text to 'Application Window Hit Test 2' from 'Application Window Hit Test'. Save the file.

Next up, we need to edit ApplHitTest_2_asm.

```
fname    dc.w   fname_e-fname-2
         dc.b   "Application Window Hit Test 2"
fname_e  ds.b   0
```

Next we need to change the application sub-window hit routine. Look through the code for the routine named ahit0 and change it to the following.

```
; Application sub-window hit routine

ahit0    movem.l d1/d3/d5-d7/a0/a4,-(a7) ; Save Hit Routine registers

         bsr.s apinit                   ; Initialise the sub-window
         bsr.s ptrpos                   ; Show the pointer position
         bsr.s keystr                   ; Display keystroke
         bsr   events                   ; Print event details

         movem.l (a7)+,d1/d3/d5-d7/a0/a4 ; Restore Hit Routine registers
         moveq #0,d0                    ; No errors
         rts
```

The main code in this advanced hit routine is simple - it stacks all the registers that we require to preserve throughout the hit routine, and makes calls to a few helper routines to carry out one specific task. I admit, this is not the most efficient method, but it allows me to split the code into manageable chunks for describing in the text.

```
; Helper - Initialise the sub-window.

apinit  movem.l d1-d2/a1-a2,-(a7) ; We need these registers later
        moveq #0,d1               ; D1.W = Application window number
        moveq #0,d2               ; D2.W = Ink colour
;                                   A4 = Working definition
        jsr wm_swapp(a2)          ; Set channel id to the sub window
        movem.l (a7)+,d1-d2/a1-a2 ; Ptr position & keystroke back again
        rts
```

The first subroutine called simply initialises the application sub-window setting the ink to black and forcing the channel Id to cover the application sub-window. Any registers corrupted by the routine are stacked on entry and restored on exit.

```
; Helper - Display pointer position details.

ptrpos  movem.l d1-d3/a1,-(a7)    ; These get corrupted here
        lea ptrx,a1               ; 'Ptr_x: '
        bsr.s print               ; Print it

        move.l (a7),d1            ; Restore the old D1 again.

        swap d1                   ; Lo = pointer X, Hi = pointer Y
        bsr pr_int2               ; Print pointer X

        lea ptry,a1               ; ' Ptr_Y: '
        bsr.s print               ; Print it

        movem.l (a7)+,d1-d3/a1    ; Retrieve other registers
        bsr.s pr_int2             ; Print pointer Y
        rts
```

The code above preservers all registers that will be corrupted and then displays the current pointer position in absolute screen coordinates. These are relative to the 0,0 position of the entire screen and not relative to the 0,0 position of the actual main window for our application.

```
; Helper - Display keystroke

keystr  movem.l d1-d3/a1,-(a7)    ; These get corrupted here
        lea keystk,a1             ; 'Key: '
        bsr.s print               ; Print it

        move.l 4(a7),d2           ; Retrieve D2
        cmpi.b #-1,d2             ; External keystroke?
        bne.s k_hit               ; no, try a HIT

        lea keyext,a1             ; 'External'
        bra.s k_doit              ; Print & exit

k_hit   cmpi.b #1,d2              ; HIT?
        bne.s k_do                ; No, try a DO

        lea keyhit,a1             ; 'HIT'
```

```
        bra.s  k_doit              ; Print & exit

k_do    cmpi.b #2,d2               ; DO?
        bne.s  k_zero              ; No, must be a key code or zero

        lea    keydo,a1            ; 'DO'
        bra.s  k_doit              ; Print & exit

k_zero  cmpi.b #0,d2               ; Zero = no key pressed
        bne.s  k_keys              ; Has to be a key press

        lea    keyzero,a1          ; 'No key'
        bra.s  k_doit              ; Print & exit

k_keys  move.w d2,d1               ; Need keystroke in D1.B
        moveq  #io_sbyte,d0
        moveq  #-1,d3
        trap   #3                  ; Print keystroke
        bra.s  k_done              ; Exit

k_doit  bsr.s  print               ; Print message
k_done  movem.l (a7)+,d1-d3/a1     ; Restore working registers
        rts
```

The code above starts, as usual, by preserving the working registers. It then checks the value in D2 to see which, if any Key was pressed to cause a hit in the application sub-window. D2 can be any of the following:

- Negative 1 = the activation key was pressed to place the pointer into the application sub-window.
- 1 - HIT - the left mouse button was clicked within the application sub-window, or the space bar was pressed.
- 2 - DO - the right mouse button or the ENTER key was pressed while the pointer was within the application sub-window.
- Zero - no key or mouse button has been pressed.
- Anything else - this will be the upper cased key code for the actual key that was pressed.

If the TAB key is pressed, you might briefly see the ‘external keystroke’ message flash across the screen quickly followed by ‘No key pressed’ - as I mentioned previously, pressing TAB (the activation key for the sub-window) results in two separate calls to the hit routine.

```
; Helper - Print event details

events  movem.l d1-d3/a1,-(a7)     ; Save the usual bunch
        lea    event,a1            ; 'Event: '
        bsr.s  print               ; Print message
        move.w d4,d1               ; Event number
        bsr.s  pr_int2             ; Print it
        movem.l (a7)+,d1-d3/a1     ; Restore the workers
        rts
```

Finally, we have the helper routine that displays details of whatever event was detected which caused the hit routine to be activated. As ever, the code starts by preserving the working registers and then examines D4 to see which, if any, event took place.

```
; Helper - Print string at (a1) to channel in A0.
; Then CLS to end of line.

print   move.w ut_mtext,a2          ; Vector to print string
        jsr (a2)                    ; Print it
        movem.l d1/d3/a1,-(a7)      ; These get corrupted
        moveq #sd_clrrt,d0          ; CLS to end of cursor line
        moveq #-1,d3
        trap #3                     ; Do it
        movem.l (a7)+,d1/d3/a1      ; Restore
        rts


; Helper - Print word int at (a1) to channel in A0.

pr_int  move.w (a1),d1              ; Get word to print
pr_int2 move.w ut_mint,a2           ; Print word int vector
        jsr (a2)                    ; Print it
        rts
```

The above routines are called by the main sub-routines themselves to display messages and numeric values on screen. The various messages are defined in the code below.

```
; Assorted TEXT messages etc follow.

ptrx     dc.w ptrx_e-ptrx-2
         dc.b 'Ptr_X: '
ptrx_e   equ *

ptry     dc.w ptry_e-ptry-2
         dc.b ' Ptr_Y: '
ptry_e   equ *

keystk   dc.w keystk_e-keystk-2
         dc.b $0a
         dc.b 'Key: '
keystk_e equ *

keyhit   dc.w keyhit_e-keyhit-2
         dc.b '1 = HIT'
keyhit_e equ *

keydo    dc.w keydo_e-keydo-2
         dc.b '2 = DO'
keydo_e  equ *

keyext   dc.w keyext_e-keyext-2
         dc.b '-1 = External Keystroke'
keyext_e equ *

keyzero  dc.w keyzero_e-keyzero-2
         dc.b '0 = No Key Pressed'
```

```
keyzero_e equ *

event    dc.w event_e-event-2
         dc.b $0a
         dc.b 'Event: '
event_e equ *
```

So, there you have it, a small and incredibly inefficient hit routine to display some of the data that are passed into an application sub-window hit routine when it is executed. I have not bothered to display the various window definition addresses etc - if you wish, feel free to create a 32 bit long to hexadecimal conversion routine to display these values.

I have deliberately left these out in an effort to save space in the magazine[1] - my listings can get a tad on the long side!

There is one final change we need to make to our source code, right at the end, where we include our window definition, change the file name to ApplHitTestWin_2_asm.

```
; Pull in our window definition file.

         in   win1_source_ApplHitTestWin_2_asm
```

Save the file. We are done.

When assembled and executed, the code in the hit routine displays a few details of register settings on entry to the hit routine. It's not very useful, but shows the pointer position in absolute screen coordinates as opposed to relative to the start of the actual sub-window (which would have been a lot more useful in my opinion), it shows the key press if any key was pressed and it shows which event, if any, occurred. Remember, only a limited number of events get through to a application sub-window hit routine.

If you press the TAB key and watch closely, you might see a brief message saying 'External Keystroke' before the text is replaced by 'No key pressed'. This shows, once again, that the TAB key results in two calls to the hit routine.

28.4 Conclusion

One thing has become obvious from even these two little routines, an application sub-window results in a huge number of hits! Even moving the pointer within a sub-window results in multiple hits. It might be better to display information - whatever the application needs to print on screen - to an information window instead. This should certainly save on processing time. However, as I mentioned above, only use a hit routine if you absolutely need one.

Failing this, the ideal hit routine for an application window should be hugely efficient - and it should exit quickly when it doesn't need to do any [further] processing, rather than just doing everything each time the code is entered.

[1] *QL Today*

28.5 Coming Up...

The next chapter continues looking at application sub-windows, but we will be loading them with menus!

29. Application Sub-Window Menus

29.1 Introduction

At the end of the last chapter, I promised to continue looking at Application Sub-Windows by adding Application Sub-Window Menus to them.

Effectively, there are two different types of application sub-window menus:

- Static - which are defined in the program source code and never change;
- Dynamic - which may change as the applications runs.

In this chapter, we shall look at the Static Application Sub-Window Menus only. In a future chapter, we shall look at Dynamic Menus as they are much more difficult to set up correctly.

29.2 Static Application Sub-Window Menus

Static menus, as I shall call them from this point onwards, are created by the developer as s/he writes the program. As the program runs, static menus do not change - other than setting entries to available or unavailable as required.

We can use SETW to create static menus. All that is required is for the developer to decide on the menu options, the required layout of rows and columns, and what to do when the user clicks on an option - although this latter option is not needed by SETW, only in the application's code.

We need to design a new window using SETW, so proceed to execute the utility and proceed as follows:

1. When prompted for 'name$' enter *AppMenuTest1*. I'd like to use the name *AppMenuTest1Win*, but that is too big for SETW. When finished, the file *AppMenuTest1_asm* is easily renamed to *AppMenuTest1Win_asm*.
2. On the 'Alter Text' screen.
    - Press N for new, type 'X' (without the quotes) then ENTER.

- Press N for new, type ‘Application Menu Test 1’ (without the quotes) then ENTER
- Press N for new, type ‘One’ then enter.
- Press N for new, type ‘Two’ then enter.
- Press N for new, type ‘Three’ then enter.
- Press N for new, type ‘Four’ then enter.
- Press N for new, type ‘Five’ then enter.
- Press N for new, type ‘Six’ then enter.
- Press N for new, type ‘Seven’ then enter.
- Press N for new, type ‘Eight’ then enter.
- Press N for new, type ‘Nine’ then enter.
- Press N for new, type ‘Ten’ then enter.
- Press ESC.

3. On the ‘Alter Sprite’ screen.
    - Press ESC.
4. On the ‘Alter Blob’ screen.
    - Press ESC.
5. On the ‘Alter Patt’ screen.
    - Press ESC.
6. Number of main windows = 1.
7. Number of Loose Items = 1.
8. Number of Information windows = 2.
9. For Information Window 1 of 2, the number of IW Objects = 1.
10. For information windows 2 of 2, the number of IW Objects = 0.10
11. Number of application windows = 1.
12. Application windows menu items = 10.
13. For main window 1:
    - Shadow = 2
    - Border size = 1
    - Border colour = colour_ql -> black
    - Paper colour - colour_ql -> white
    - Sprite = arrow
14. Presentation of loose Items:
    - Press N for ‘system palette defaults’
    - Confirm N when prompted again for defaults
    - Border size = 1
    - Border colour = colour_ql -> black
    - Unavailable background = colour_ql -> white
    - Unavailable Ink = colour_ql -> grey
    - Available background = colour_ql -> white
    - Available Ink = colour_ql -> black
    - Selected background = colour_ql -> green
    - Selected Ink = colour_ql -> black
15. Loose Item 1:
    - Type = text
    - Object -> select the ‘X’ text object
    - Selection key = ESC
16. Information Window 1:
    - Border size = 0
    - Paper = colour_ql -> No 92

17. Object 1:
    - Type = text
    - Object -> select the 'Application Window Test' text object.
    - Colour = colour_ql -> black
    - Xcsize = 0
    - Ycsize = 0
18. Information Window 2:
    - Border size = 1
    - Border colour = ql_colour -> black
    - Paper = colour_ql -> white
19. Application Window 1:
    - Border size = 1
    - Border colour = colour_ql -> black
    - Paper colour = colour_ql -> white
    - Sprite = arrow
    - Selection key = TAB
    - Presentation of Menu Items
      - Select N for system palette defaults
      - Select N for defaults, again.
      - Border size = 1.
      - Border colour = ql_colour -> black.
      - Unavailable background = ql_colour -> white.
      - Unavailable ink = ql_colour -> grey.
      - Available background = ql_colour -> white.
      - Available ink = ql_colour -> black.
      - Selected background = ql_colour -> green.
      - Selected ink = ql_colour -> black.
      - Scroll arrow = ql_colour -> white.
      - Scroll bar = ql_colour -> black.
      - Scroll background = ql_colour -> red.
      - When prompted for the ten menu items, select the text items 'One' through 'Ten'. Give each one a selection key of the digit that matches the number described by the text object. For example, 'One' has a key of '1', 'Two' has '2' and so on up to 'Ten' which has selection key '0' (Zero).
20. Main window size: (Use the arrow keys to change the size, press ENTER when correct)
    - Width = 220
    - Height = 140
    - Do you want a variable window = N
    - Set the origin to 0,0 (Press ENTER when correct)
21. Loose Item 1: (Toggle hit/position with F2. Press ENTER when correct)
    - Hit size = 10 x 10
    - Position = 206 x 3
22. Information Window 1: (Toggle size/position with F2. Press ENTER when correct)
    - Size = 220 x 16
    - Position = 0 x 0
    - Object position = 2 x 2
23. Information Window 2: (Toggle size/position with F2. Press ENTER when correct)
    - Size = 216 x 14
    - Position = 2 x 125

24. Application Window 1: (Toggle size/position with F2. Press ENTER when correct)
    - Size = 208 x 104
    - Position = 2 x 18

When you have completed this procedure, and SETW has exited, you should save the file ram1_AppMenuTest1_asm to a safer place and rename it to AppMenuTest1Win_asm. The file should look similar to the following, although I have added some extra comments to my copy of the generated code.

29.3 The Generated Code

The file should look similar to the following, although I have added some extra comments to my copy of the generated code.

Note I have removed a few sections of the following file in order to reduce duplication of chunks of code in the magazine. These sections are discussed below.

```
; AppMenuTest1Win_asm.

; Undefined Labels - need to be defined elsewhere in my own code.

;         ahit2_0 - menu item 0, hit routine.
;         ahit2_1 - menu item 1, hit routine.
;         ahit2_2 - menu item 2, hit routine.
;         ahit2_3 - menu item 3, hit routine.
;         ahit2_4 - menu item 4, hit routine.
;         ahit2_5 - menu item 5, hit routine.
;         ahit2_6 - menu item 6, hit routine.
;         ahit2_7 - menu item 7, hit routine.
;         ahit2_8 - menu item 8, hit routine.
;         ahit2_9 - menu item 9, hit routine.
;         asmnu0  - User defined setup routine.
;         adraw0  - User defined draw routine.
;         ahit0   - application window 0 hit action routine.
;         afun0_0 - Loose item 0 hit action routine.

; Labels for External Use
;      mst0     - menu items status area
;      wst0     - Window status area
;      wd0      - Window definition address
;      ww0_0    - Window default size
;      ww0_1    - Window button size

SYS_SPR  dc.w     0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
                  19,20,21,22,23,24,25,26,7,28,29,30,31,32,33,
                  34,35,36,37

; Text object for "Close" loose item.
txt0     dc.w     txt0_e-2-txt0
         dc.b     "X"
txt0_e   ds.b     0
         ds.w     0

```

```
; Text object for caption bar.
txt1      dc.w      txt1_e-2-txt1
          dc.b      "Application Menu Test 1"
txt1_e    ds.b      0
          ds.w      0

;-----------------------------------------------------------------------
; **** Text objects for the menu items. Removed - see text.

; **** Menu items list. Removed - see text.

; **** Row list. Removed - see text.

; **** Spacing list. Removed - see text.
;-----------------------------------------------------------------------

; Application window list.
app_list0
          dc.w      appw0-*
          dc.w      0


;-----------------------------------------------------------------------
; **** Application window 0 definition. Removed - see text.
;-----------------------------------------------------------------------


; Information Object(s).
pobl0
          dc.w      138       xsize
          dc.w      10        ysize
          dc.w      2         xorg
          dc.w      2         yorg
          dc.b      0         type
          dc.b      0         spar
          dc.l      0         spce
          dc.w      txt1-*    pobj *
          dc.w      -1

; Information Window(s).
infw0
          dc.w      220       xsize
          dc.w      16        ysize
          dc.w      0         xorg
          dc.w      0         yorg
          dc.w      0         flag
          dc.w      0         borw
          dc.w      526       borc
          dc.w      92        papr
          dc.w      pobl0-*   pobl *
          dc.w      216       xsize
          dc.w      14        ysize
          dc.w      2         xorg
          dc.w      125       yorg
          dc.w      0         flag
          dc.w      1         borw
```

```
            dc.w    0           borc
            dc.w    7           papr
            dc.w    0           pobl *
            dc.w    -1          end

; Loose Item(s).
litm0
            dc.w    10,10       xsize, ysize
            dc.w    206,3       xorg, yorg
            dc.b    0,0         xjst, yjst
            dc.b    0,3         type, skey
            dc.w    txt0-*      pobj *
            dc.w    0           item
            dc.w    afun0_0-*   pact *
            dc.w    -1          end

litm1
            dc.w    16404,12    xsize, ysize
            dc.w    0,0         xorg, yorg
            dc.b    0,0         xjst, yjst
            dc.b    0,0         type, skey
            dc.w    0           pobj *
            dc.w    0           item
            dc.w    0           pact *
            dc.w    -1          end

; Window definition
wd0
            dc.w    220         xsize
            dc.w    140         ysize
            dc.w    0           xorg
            dc.w    0           yorg
            dc.w    258         flag
            dc.w    1           borw
            dc.w    0           borc
            dc.w    7           papr
            dc.w    0           sprt *
            dc.w    1           curw
            dc.w    0           curc
            dc.w    7           uback
            dc.w    255         uink
            dc.w    0           ublob *
            dc.w    0           upatt *
            dc.w    7           aback
            dc.w    0           aink
            dc.w    0           ablob *
            dc.w    0           apatt *
            dc.w    4           sback
            dc.w    0           sink
            dc.w    0           sblob *
            dc.w    0           spatt *
            dc.w    0           help
            dc.w    220         xsize
            dc.w    140         ysize
            dc.w    infw0-*     pinfo *
            dc.w    litm0-*     plitem *
```

```
        dc.w    app_list0-*         pappl *
        dc.w    16384     xsize
        dc.w    12        ysize
        dc.w    0         pinfo *
        dc.w    litm1-*   plitem *
        dc.w    0         pappl *
        dc.w    -1

; Sizes
ww0_0   equ     670
ww0_1   equ     148

; Status Areas:
; Menu item status area.
mst0    ds.b    10
mst0_e  ds.b    0
        ds.w    0

; Window status area.
wst0    ds.b    65
wst0_e  ds.b    0
        ds.w    0
```

Listing 29.1: AppMenuTest1Win_asm

Much of the above is similar to when was discussed in a previous article. If you think back to that article on application sub-windows, we simply used the hit routine to print text all over the application sub-window. That was about as simple as it gets - other than not actually having a hit routine I suppose! Adding a menu to an application sub-window means we have quite a lot more work to do at the coding stage as we have to consider the following:

- Defining the menu objects - these can be text, sprite etc. Every menu item must be defined.
- Defining the menu items list - when we have defined each menu object, we then have to build a list of all the menu items that we wish to include in our final menu.
- Defining the menu row list - when we have the menu items list defined, we amalgamate that list into a menu row list which defines the start and end of each row in the menu.
- Defining the spacing lists - the row list defines the hit size and the spacing for each row and each column in the menu.
- Define the application sub-window - menus need their own section in the application sub-window definition.

So much to do just to show a menu in a window, lets get on and do it.

29.4 Menu Objects

The first stage is to define the various objects that will be incorporated into the menu. In this example, I have used ten separate text objects (as they are the simplest). You can use any of the various Pointer Environment object types if you wish. The code generated by SETW for these items is as follows.

```
; **** Text objects for the menu items.

txt2    dc.w    txt2_e-2-txt2
        dc.b    "One"
```

```
txt2_e    ds.b      0
          ds.w      0


; **** Txt3 through txt10 deleted for brevity.


txt11     dc.w      txt11_e-2-txt11
          dc.b      "Ten"
txt11_e   ds.b      0
          ds.w      0
```

Listing 29.2: AppMenuTest1Win_asm - Menu Objects

As you can see from the above, I have omitted to show text objects txt3 through txt10 as there is really no need to take up space in the magazine with repetitive data.

The above code simply defines ten separate text objects - ‘One’, ‘Two’, ... ‘Nine’ and ‘Ten’ to be used, later, in our static menu.

You should be aware, also, that the line numbers above (and indeed, in the following snippets) bear no resemblance to the ones in the original file.

29.5 Menu Items (and Index) List

The next section of code that I have removed from the main listing above is the menu items list. This is shown below, but please note that once again, I have removed the vast majority of the code for brevity.

```
; **** Menu items list.

meos2     dc.b      1,0               ; x_justification, y_justification
          dc.b      0,49              ; Item type, selection key
          dc.w      txt2-*            ; Pointer to object
          dc.w      0                 ; Item number. (-1 for indexes)
          dc.w      ahit2_0-*         ; Hit routine for this item
;                                     ; Zero for indexes


; NOTE: Menu items 1 through 8 removed here for brevity.


          dc.b      1,0               ; x_justification, y_justification
          dc.b      0,48              ; Item type, selection key
          dc.w      txt11-*           ; Pointer to object
          dc.w      9                 ; Item number. (-1 for indexes)
          dc.w      ahit2_9-*         ; Hit routine for this item
;                                     ; Zero for indexes
```

Listing 29.3: AppMenuTest1Win_asm - Menu Item List

The label meos2 is the start of the menu items list. We told SETW that there would be 10 items, and so, there will be 10 menu items in this list. Each one has the structure described in Table 29.1.

Each item in a menu items list is 10 bytes in length.

Offset	Size	Description
0	1	X (Horizontal) justification. • -ve = Right justified • 0 = Centred • +ve = Left Justified The justification value is the number of pixels from the edge of the hit area that the object is to be positioned at.
1	1	Y (Vertical) justification. • -ve = Bottom justified • 0 = Centred • +ve = Top Justified The justification value is the number of pixels from the edge of the hit area that the object is to be positioned at.
2	1	Item type: • 0 - Text object • 2 - Sprite object • 4 - Blob object • 6 - Pattern object
3	1	Selection key, upper cased if necessary.
4	2	Relative pointer to the actual object.
6	2	Item number. If this is an index items list, set to -1 for all items.
8	2	relative pointer to the hit/action routine for this particular item. For index item lists, must be zero.

Table 29.1: Application Sub-Window Menu Item List Entry

Looking at the first menu object generated by SETW, we see that it is left justified and one pixel from the start of the left end of the hit area. Vertically, it is centred within the hit area. It is a text object and has a selection key of 49, witch is the code for a digit '1' on the keyboard. The text object itself is the text 'One' as shown above in the preceding section. The item is numbered zero and the hit routine for this item is defined to be at label ahit2_0.

So far, so good. We have defined a list of objects and then gathered them into a list of menu items. The menu items list - in this case - is in a contiguous section of memory, it need not be so. The row list defines the menu ordering, that comes next.

The above structure is used to define menu item lists and also index lists. An index is drawn by the `wm_index` vector (which also draws pan and scroll bars & arrows - if necessary). Indices are best thought of as the row and column headings - similar to a spreadsheet, for example, when columns have letters and rows have numbers to identify them.

WMAN takes care of aligning the indices with the contents of the static menu.

29.6 Row List

The row list takes the various menu items, defined above, and organises them into rows - surprisingly enough. For every row you wish to have in your menu, you need a single row list entry. As SETW tries to make as few rows and/or columns as it can - it tries to fit as much as possible into a given space - what SETW generates may not be what you want. In the default case for our SETW session, we have been given two rows and thus, five columns, for our ten menu items.

Our generated row list is as follows:

```
; **** Row list.

drow0     dc.w      0+meos2-*          ; Pointer to row 0 start = item 0.
          dc.w      50+meos2-*         ; Pointer to row 0 end = item 4.

          dc.w      50+meos2-*         ; Pointer to row 1 start = item 5.
          dc.w      100+meos2-*        ; Pointer to row 1 end = item 9.
```

Listing 29.4: AppMenuTest1Win_asm - Row List

Each row list item contains two pointers, the first is to the start of the first entry in the menu items list entry for this row. The second pointer is to the first byte past the end of the last menu items list entry for this row.

Given the above then, we can see that the first row, starting at label drow0, begins at meos2 (relative to the pointer itself - as usual). Meos2 is a list of 10 sets of 10 byte entries defining all ten items in our menu. The first row ends at meos2 + 50, which happens to be the very first byte of the menu items list for menu item number 5 (ie, the sixth menu item - we count from zero)

The second row list entry starts at meos2 + 50 and ends at meos2 + 100. These pointers are to menu items list item number 5 and at the first byte past the very last menu items list entry. As the following diagram attempts to display:

```
Meos2+0     Menu Item 0 <-----+           Start of row 0.
Meos2+10    Menu Item 1       |
Meos2+20    Menu Item 2       |
Meos2+30    Menu Item 3       |
Meos2+40    Menu Item 4       |
Meos2+50    Menu Item 5 <---------+--+    End of row 0, start of row 1.
```

```
Meos2+60     Menu Item 6          |   |   |
Meos2+70     Menu Item 7          |   |   |
Meos2+80     Menu Item 8          |   |   |
Meos2+90     Menu Item 9          |   |   |
Meos2+100    equ   *        <---------------+ End of row 1.
                                  |   |   |   |
Drwo0         Pointer to -------+   |   |   |
              Pointer to -----------+   |   |
              Pointer to ---------------+   |
              Pointer to -------------------+
```

Listing 29.5: Relationship between the Row List & Menu Items List

If the menu items list is a single chunk of memory, then each row start pointer is equal to the previous row end pointer - except for the first row. As in the example above, the end pointer for row 0 is the same address as the start pointer for row 1.

Now that we have our rows defined, we have to set up the spacing lists for each row and each column in the menu.

29.7 Spacing Lists

Each menu item in our static menu has a given hit area and a spacing. The hit area defines where the pointer can be to make the item beneath it the current item, this is normally indicated by a border being drawn around the current item. A HIT or a DO within the hit area, or a press of the selection keystroke while the pointer is withing the application sub-window, will cause the appropriate menu item action routine to be executed.

The spacing defines how many pixels across - or down depending on whether this is the column or row spacing list - there are between the start of ‘this’ menu item and the start if the ‘next’ one. The spacing *must* include an allowance for the border to be drawn around the current item.

```
; **** Spacing lists.

; Spacing list. Defines width of hit area for each COLUMN and spacing
; between columns. (5 columns.)

spls0    dc.w     34      ; Hit area width column 0
         dc.w     36      ; Space between this column and the next.

         dc.w     34      ; Hit area width column 1
         dc.w     36      ; Space between this column and the next

         dc.w     34      ; Hit area width column 2
         dc.w     36      ; Space between this column and the next

         dc.w     34      ; Hit area width column 3
         dc.w     36      ; Space between this column and the next

         dc.w     34      ; Hit area width column 4
         dc.w     36      ; Space between this column and the next


; Spacing list. Defines height of hit area for each ROW and spacing
; between rows. (2 rows.)
```

```

spls1      dc.w       10         ; Hit area height row 0
           dc.w       12         ; Space between this row and the next

           dc.w       10         ; Hit area height row 1
           dc.w       12         ; Space between this row and the next
```

Listing 29.6: AppMenuTest1Win_asm - Spacing Lists

The first list above, at label spls0 defines the columns in our menu. We already know that SETW has decreed that there shall be five columns and two rows, so the column spacing list has five entries, one for each column. Each entry consists of a pair of words - the first defines the width of the column (or the height of the row) and the second defines the space between this column and the next.

In the example above, we see that SETW has calculated that our widest text object is 5 characters wide - this corresponds to ‘Three’, ‘Seven’ and ‘Eight’ - and has allocated 34 pixels of hit area for each column. The spacing for each columns is set to the (border width * 2) plus the hit area width. It must be twice the border width as there is a border on each side (or top & bottom).

The spacing list for the rows shows a height of ten pixels for the hit area and taking the border into consideration again, a spacing of 12 pixels between the tops of each row.

29.8 Menu Section of Application Window Definition

The application window definition needs an extra section adding after the normal definition, to cover the need for a static menu. In addition, two entries in the normal definition part are amended (from what we used for an application sub-window without a menu - see last time) to point to the:

- User defined setup routine, or zero if not required.
- User defined drawing routine, or zero if not required.

The new style application window definition is as follows:

```
; **** Application window 0 definition.

appw0      dc.w       208        ; Width in pixels (+ scaling)
           dc.w       104        ; Height in pixels (+ scaling)
           dc.w       2          ; X origin, relative to 0 in main window
           dc.w       18         ; Y origin, relative to 0 in main window
           dc.w       256        ; Flag - bit 7 set = clear window
;                                ;      - bit 1 set = disable cursor keys
           dc.w       1          ; Border width
           dc.w       0          ; Border colour
           dc.w       7          ; Paper colour
           dc.w       0          ; Pointer to pointer sprite, or 0 for arrow
```

Listing 29.7: AppMenuTest1Win_asm - Application Window Definition

The first part is exactly as we used last time, nothing different to see here. Following the above, we have this:

```
; Note the following for menus.

           dc.w       asmnu0-*   ; User defined setup routine, or 0
           dc.w       adraw0-*   ; User defined drawing routine, or 0
```

```
            dc.w        ahit0-*   ; Application window hit routine
            dc.w        0         ; Control routine, or 0
            dc.w        0         ; Max X control sections (splits)
            dc.w        0         ; Max Y control sections (splits)
            dc.b        9         ; Selection key
            dc.b        0         ; Spare byte - must be 0
```

Listing 29.8: AppMenuTest1Win_asm - Application Window Definition

The first two entries in the above definition are the new ones. These are our pointers to a user defined setup routine and a user defined drawing routine. You will notice that the application window still has its own hit routine, even though it contains a menu and each and every menu item has a dedicated hit routine of its own. Note also, in this small example,that our settings for the pan and scroll sections are all unused. We'll come back to those in a future chapter.

The user defined setup code would normally consist of a single line as follows:

```
asmnu0  jmp  wm_smenu(a2)     ; Vector $08
```

Listing 29.9: AppMenuTest1Win_asm - Application Window Setup Routine

Similarly, the user defined drawing routine need only perform the following tasks:

```
adraw0  jmp  wm_index(a2)     ; Vector $34
        bne.s  adexit         ; Bale out on errors
        jmp  wm_mdraw(a2)     ; Vector $20
adexit  rts
```

Listing 29.10: AppMenuTest1Win_asm - Application Window Drawing Routine

The call to `wm_index` is not required unless your menu has been defined to have sections and/or index items[1]. What are index items? Think of a spreadsheet, each row has a number and each column has a letter. These are the index items. Our example is not using index items, however, if it did then we would set them up exactly as per the menu items list, except, for indexes the list entries have no hit routine (set to zero) and the item number is always -1.

Note If the pointer to the user defined drawing routine is zero, then WMAN will still draw the application sub-window's border and, unless the flag is set to say 'do not clear', will clear it to the defined paper colour. If you find missing menus in your application sub-windows, check that you have a drawing routine!

Following on from the above, there is a brand new section dedicated to the menu.

```
; The following section is required when an application sub-window
; contains a menu.

            dc.w        mst0-wst0 ; Pointer to menu status area. (See text)

            dc.w        1         ; Current Item, border width
            dc.w        0         ; Current Item, border colour

            dc.w        7         ; Unavailable background colour
            dc.w        255       ; Unavailable ink colour
            dc.w        0         ; Unavailable blob pointer
```

[1] George Gwilt has discovered that anything to do with these index items is not actually implemented in the WMAN code. Looks like Tony Tebby had a good idea, that couldn't be fulfilled.

```
        dc.w    0       ; Unavailable pattern pointer

        dc.w    7       ; Available background colour
        dc.w    0       ; Available ink colour
        dc.w    0       ; Available blob pointer
        dc.w    0       ; Available pattern pointer

        dc.w    4       ; Selected background colour
        dc.w    0       ; Selected ink colour
        dc.w    0       ; Selected blob pointer
        dc.w    0       ; Selected pattern pointer

        dc.w    5       ; Number of columns in the menu
        dc.w    2       ; Number of rows in the menu
        dc.w    0       ; X offset to start of menu
        dc.w    0       ; Y offset to start of menu
        dc.w    spls0-* ; Pointer to column spacing list
        dc.w    spls1-* ; Pointer to row spacing list
        dc.w    0       ; Pointer to column index list
        dc.w    0       ; Pointer to row index list
        dc.w    drow0-* ; Pointer to menu row list
```

Listing 29.11: AppMenuTest1Win_asm - Application Window Menu Area Definition

The first new entry we need is a pointer to the menu items status area. This has been defined for us, by SETW, at label mst0. There should be a single byte for each menu item. Note however, that we need to have this status area pointer defined as relative to the window status area. Hence the calculation in the above definition.

Note This fact is not very clearly documented in the PE documentation. I had an extended conversation with George on this setting as I had never seen the fact that the menu status area pointer is relative to the window status area - George had a pencilled in note in his copy of the documentation indicating this need. Obviously, I didn't.

Next up, we see the menu attributes - border width and colour, item paper and ink, blobs and patters for unavailable, available and selected items.

After the attributes section, we define the menu itself with details of how many columns there are, how many rows, offsets to the start of the menu and the pointers to the various sections discussed above.

29.9 Application Sub-Window Menu Item Hit Routines

In addition to the application sub-window's own hit routine, as described previously, each and every item in the menu (Static or dynamic) may also have a hit routine. This routine could be a single one for all, or a separate one for each menu item. It depends on how the program is designed.

Note Whenever a program has a static or dynamic menu, there *must* be a hit routine for the application sub-window containing the menu. The absolute minimum code in the hit routine is as follows:

```
ahit0   jmp wm_hit(a2)          ; Vector $34
```

Listing 29.12: AppMenuTest1Win_asm - Application Window Hit Routine

If you do not have the above code present in the hit routine for the application sub-window, then when you attempt to hit or do a menu item, nothing will work. The above code does not need an `RTS`.

On entry to a menu item hit routine various registers are set with specific parameters as described in Table 29.2.

Register	Description
D1.L	Virtual column/row for the hit menu item.
D2.W	Item number.
D4.L	An event number. This can only be 0 or pt__do (16).
A0.L	Channel id.
A1.L	Pointer to the menu status area.
A2.L	WMAN vector.
A3.L	Pointer to sub-window definition.
A4.L	Pointer to window working definition.

Table 29.2: Menu Item Hit Routine Registers

Registers not mentioned above are free for use as they are not used by the hit routine.

Hit routines should exit with D5 - D7, A0 and A4 preserved to the same value that they had on entry to the routine. D1 - D3, A1 - A3, A5 and A6 are undefined on exit (which means that they don't care what value they have.) D4.B must be either zero or a window event to be set on exit.

D0 should contain zero or an error code and the SR must be set according to the value in D0 on exit.

Note

D3, on return from a hit routine, should normally be returned as per its value on entry. It is not used by `wm_rptr` however, it is used by `wm_rptrt` (read pointer with return on timeout) from WMAN 1.5 onwards. Wm_rptr ignores the upper word of D3. If your read pointer loop is using the `wm_rptrt` vector instead, and you have changed the value of D3 within the hit code, you must clear the high word on exit.

On exit, if D0 is clear and the status (Z) bit is set, control will return to the `wm_rptr` loop and not to your application's code. To return to your own code, the hit routine needs to set at least one event bit in the event vector which can be done by returning a suitable value in D4.B on exit.

If an error is detected within the hit code, then it should exit with the appropriate error code in D0 and the status register set accordingly.

29.10 Coming Up...

So, that's the end of this exciting chapter. We have designed a window and looked deep into the structures involved in defining a static menu.

In the upcoming chapter we'll add some code and play around. We might even see if it's possible to take the design from SETW and massage it to suit our own needs and considerations.

30. Creating and Using Libraries With GWASL

30.1 Introduction

At the end of the last issue, I promised to continue looking at Application Sub-Windows by adding some code to our menu enabled program. Unfortunately, due to the current very busy situation at work, and a mild dose of tendonitis in my thumb, I'm having to do a lot of one-handed typing these days which is slowing me down a lot. To this end, I'm taking a break from application menus for a wee while, and this issue, my article will be small - but hopefully, perfectly formed - looking at how we can create and use our own libraries of useful routines with GWASL.

30.2 The Library Code

The following is the complete code for a small library that allows your own assembly code to clear various parts of the screen. I apologise for the briefness of this article, but as I said, I'm typing one handed at the moment.

The code should be typed into a file named lib_cls_asm or something similar.

```
;==========================================================;
; lib_cls_asm.                                             ;
;==========================================================;
; A small library to demonstrate the use of same in GWASL ;
; It's not particularly useful, it only demonstrates a     ;
; point!                                                   ;
;==========================================================;
; All routines expect the channel id in A0.L.              ;
; All routines assume infinite timeout.                    ;
; All regsiters are preserved, except D0.L.                ;
; Error codes are returned in D0.L and the Z flag.         ;
;==========================================================;

```

```
cls_screen equ $20
cls_top    equ $21
cls_bottom equ $22
cls_line   equ $23
cls_end    equ $24

infinity   equ -1

;--------------------------------------------------------------
; CLEAR_SCREEN - Clears entire screen.
;--------------------------------------------------------------
clear_screen  moveq #cls_screen,d0
              bra.s just_do_it

;--------------------------------------------------------------
; CLEAR_TOP - Clears top of screen.
;--------------------------------------------------------------
clear_top     moveq #cls_top,d0
              bra.s just_do_it

;--------------------------------------------------------------
; CLEAR_BOTTOM - Clears bottom of screen.
;--------------------------------------------------------------
clear_bottom  moveq #cls_bottom,d0
              bra.s just_do_it

;--------------------------------------------------------------
; CLEAR_TO_EOL - Clear to end of cursor line.
;--------------------------------------------------------------
clear_to_eol  moveq #cls_end,d0
              bra.s just_do_it

;--------------------------------------------------------------
; CLEAR_LINE - Clears entire cursor line.
;--------------------------------------------------------------
clear_line    moveq #cls_line,d0

just_do_it    movem.l d1/d3/a1,-(a7)
              moveq #infinity,d3
              trap #3
              movem.l (a7)+,d1/d3/a1
              tst.l d0
              rts
```

Listing 30.1: Example Library - Lib_cls_asm

So, you can see that there's not much to it.

That's the end of step one. The next step is to assemble the file using GWASL in the normal manner, fix any errors, and create an output file most likely named lib_cls_bin. In addition to the binary file, there will be another symbol file named lib_cls_sym created. We need that file shortly, however, it isn't in a format we can use just yet.

Once all errors have been removed and the source assembled, we are ready to move onto creating our library. In actual fact, half of the library is already created - lib_cls_bin - but we need to convert the symbol file into a text file that we can include in our own source code in order to actually call the routines in the library.

Execute the utility named sym_bin in your gwasl directory. The layout of the screen should look pretty familiar if you have used GWASL frequently. Choose option 1 as normal, and type in the path to the lib_cls_sym file.

After a couple of seconds, you can choose the option to exit. Our work is done!

Sym_bin has taken the binary formatted lib_cls_sym file and created from it a new text file names lib_cls_sym_lst. If you open this in an editor, it will look something like the following:

```
CLS_SCREEN      EQU     $00000020
CLS_TOP         EQU     $00000021
CLS_BOTTOM      EQU     $00000022
CLS_LINE        EQU     $00000023
CLS_END         EQU     $00000024
INFINITY        EQU     $FFFFFFFF

CLEAR_SCREEN    EQU     *+$00000000
JUST_DO_IT      EQU     *+$00000012
CLEAR_TOP       EQU     *+$00000004
CLEAR_BOTTOM    EQU     *+$00000008
CLEAR_TO_EOL    EQU     *+$0000000C
CLEAR_LINE      EQU     *+$00000010
```

Listing 30.2: Example Library - Lib_cls_sym_lst

You can see that all the equates defined in our source code have been made visible as well as offsets to the various routines. These offsets are the actual addresses within the lib_cls_bin file where the individual routines start.

It would be nice if there was some way for equates etc within a library to be invisible from outside it without us having to do too much extra work, however, as far as I'm aware, it's not possible to define an equate as 'local' - similar to SuperBasic. What we can do is delete the top few lines leaving only the offsets to the routines. Edit the file to delete the lines from CLS_SCREEN down to INFINITY.

Next, create a new file containing these two lines, call it lib_cls_in:

```
                in      win1_gwasl_libs_lib_cls_sym_lst
                lib     win1_gwasl_libs_lib_cls_bin
```

Listing 30.3: Example Library - Lib_cls_in

And that's all there is to it. To use the code simply include the following at the end of your own assembly code:

```
                in      win1_gwasl_libs_lib_cls_in
```

Listing 30.4: Example Library - Invoking the Library

Obviously, your paths will be different from mine, so change accordingly to suit your own system.

I have combined the IN and the LIB commands into one single file because I like to do as little typing as possible. You need not do this and to use the library, simply add the two lines above into the end of your own code at some point.

To demonstrate the code, all you need is something like this. Not shown in this example are other libraries that I use to set colours, open screens etc.

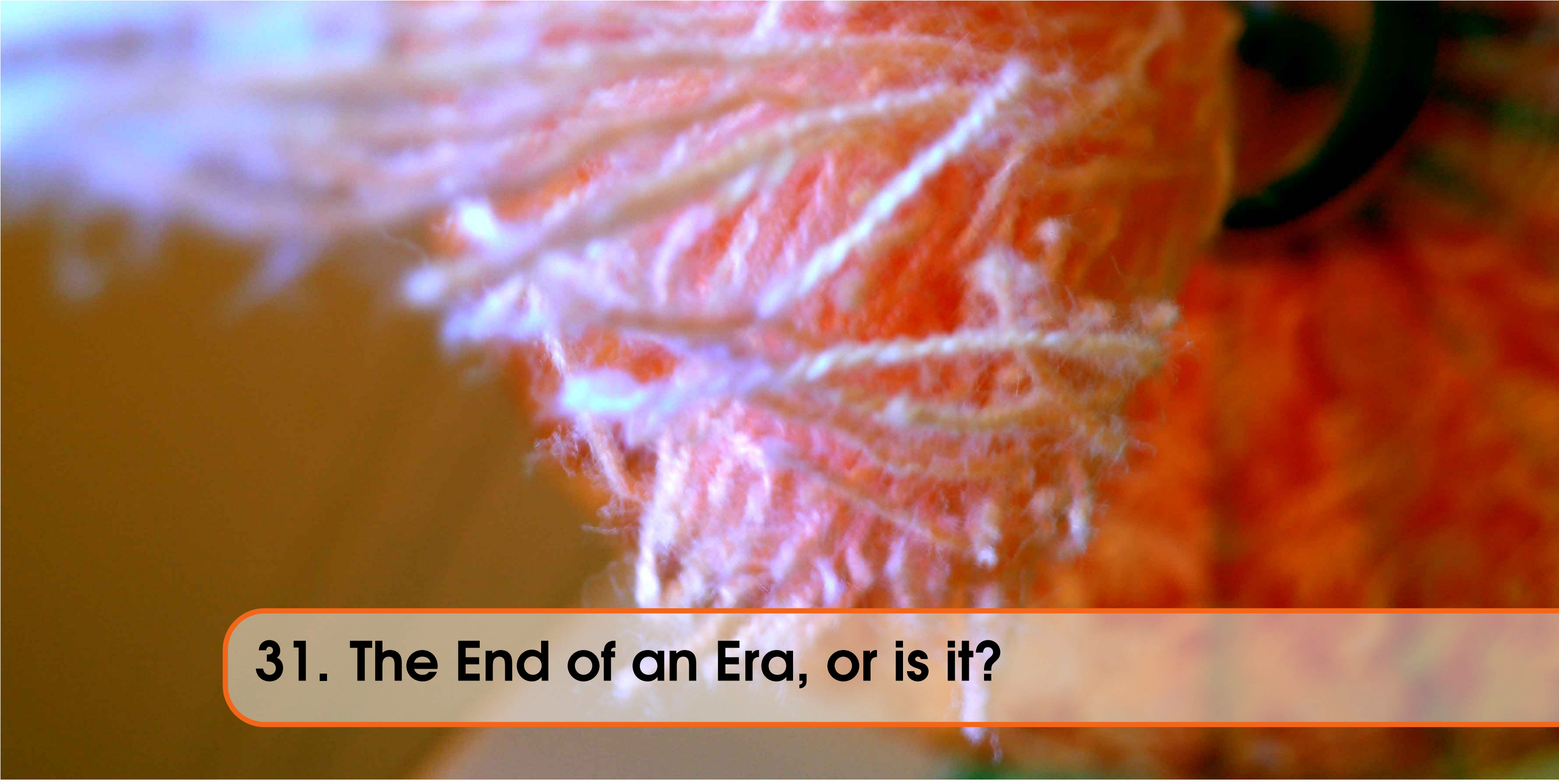
```
start       bsr   open_scr       ; Open scr_ channel return id in A0
            bsr   set_colours    ; Set paper, strip and ink preserves
;                                ; all registers except D0.L
            bsr   clear_screen   ; Clear screen
            ...                  ; Etc

            in  win1_gwasl_libs_lib_cls_in
            in  win1_gwasl_libs_lib_defaults_in
            in  win1_gwasl_libs_lib_colours_in
```

Listing 30.5: Example Library - Brief Example of Use

30.3 End Of Chapter 30

In the next chapter I'll continue where I left off at the end of the previous chapter. I shall also be looking at the recent changes to EasyPEasy.

IX The End - So Far Anyway



31. The End of an Era, or is it?

31.1 Introduction

In the last issue, we ended up with a LibGen[1] application that was getting somewhere. But it's not finished yet. This issue might well be the last *paper* copy of QL Today that you receive, but I have no current plans to stop development of this utility, nor to stop writing down stuff as I go along! I need to take a slight diversion into creating dynamic application sub-window menus before I can finish the utility properly. Unfortunately, this issue will not be continuing the program's development as I am in the middle of a huge amount of work in my current contract, and by the time you read this, I'll hopefully be in a new one.

31.2 So What Now?

Well, I have a half finished application and a few more articles on the Pointer Environment up my sleeve. Time permitting of course. As there will unlikely be a future paper version of QL Today, and I have no idea what the future of a replacement might be, I am setting up a mailing list on my web site so that anyone who wishes to take advantage of the remainder of the series, plus any other work I can think of and have time, to write, can.

You will be required to register on the list with a valid email address and I will also need your name too. My blog gets numerous registrations on a daily basis and most of them are from spam bots hoping to get free spam comments posted on my blog - they don't! Anyone signing up without a valid name and email gets deleted as part of my regular housekeeping. The mailing list will not allow you to register without a name and email address. Please supply your valid name. No nicknames please. That saves me some work clearing out the spam bots as well.

You will not get spammed by me when you register. Traffic will be light I imagine. Whenever I

[1] Well, yes we did, in the first version of the book. Since that was published, I've decided to rewrite the whole set of chapters on LibGen as a separate book, so watch this space....

have an article ready, I'll send an email and supply a link where you can obtain the latest article. I'm looking at mailing list software that allows me to add attachments to the emails sent out, but so far, these seem few and far between - at least amongst the ones I'm allowed to use by my hosting company that is.

I know Dave Park mentioned that he would be setting up a Joomla system to replace the printed QL Today, but I haven't heard much for a while, so I'm not sure of progress on that matter.[2]

Anyway, check the web page at `http://qdosmsq.dunbar-it.co.uk` to see if the details of the mailing list have been added, and if so, join up to keep reading the rest of the series. At least with my own mailing list, I'll have a half decent idea of how many readers I actually have![3] ;-)

31.3 The End

So, that's it. I've been writing these articles since the very first volume of QL Today, 17 years ago! It's been a long hard slog at times, and I haven't regretted a minute of it. I'd like to thank my faithful reader(s), George Gwilt who has far better coding skills than I have, and who kept a watchful eye on everything I wrote, offering corrections, bug fixes and observations on just about every article. Thanks George.

Hugh Rooms has commented on my articles as well as offering solutions too. And for that I'm grateful.

To all of you who read my articles and never once gave me any feedback, I thank you too. Without you, I wouldn't have as many readers as I have - but honestly, if you ever get involved in a series like that again, please give the author some feedback - even just a quick email to say "hello" or similar. Writing in isolation, for free, is fine, but it's far better to know that your efforts are being read by the "masses".

I wish everyone involved in QL Today, best wishes for the future.

Cheers, Norm.

[2] As far as I am aware, this unfortunately, never happened. Dave's other commitments prevented him from getting this off the ground.

[3] At the last count, I have 54 readers!

X Appendices and Other Blurb!

A. How this book Evolved

This book started life on my PC as text files - one for each chapter, well, for a while it did. Eventually, I decided to convert to creating the files directly in Docbook format as my main source code, and *those* were converted to text format prior to sending them off to Dilwyn and/or Geoff for inclusion in the QL Today magazine.

These initial non-DocBook text files were manually edited to wrap paragraphs and listings and warnings etc in the appropriate Docbook syntax. It's quite amazing how much more difficult it is to change text into Docbook than the other way around!

Once all the chapters were Docbook'd, I ran them through a validator to ensure that they were indeed valid XML *and* valid Docbook.

The validated chapters were gathered together into a 'book' - as defined by Docbook - and the raw XML processed by a utility named Publican which allowed me to create numerous different output formats from the same input source file. Sadly, I had a few problems with Publican - it's a great tool, don't get me wrong, and they even have a version for Windows (it's free too!) but it wasn't really what I needed. I decided to go for a proper system instead. (I still use Publican for other work.)

Enter LaTeX

LaTeX is a text processing system as opposed to a PDF generator, it is much used by scientists around the world, amongst others. If it's good enough for CERN, it's good enough for me.

This actual book that you are reading now was slightly different. It was further processed by a utility called dblatex to produce LaTeX format source text and converted into a book by the TeXstudio application.

I think you'll agree that using LaTeX creates a far nicer version of PDF etc. That version of the book was released into the wild around Christmas 2014.

Since then, I have manually gone through all the different chapter files to remove the old DocBook conversion routines etc, and replace them with plain LaTeX ones instead. This reduced the amount of code that I had to have lying around. The final result is a set of plain LaTeX source files.

These source files were collected together and merged into a LaTeX template designed for books, the *Legrand Orange Book*, which I modified quite a lot to produce the wonderfully typeset version of the pdf book that you are reading.

There's a lot of work goes into this you know! ;-)

B. Debugging with QMON2

Many years ago while still working on the Project - QLTdis - I had a small problem. `ADDX` and `SUBX` were being decoded as `ADD` or `SUB` when I tested a file containing `ADDX` and `SUBX` instructions. What was going wrong? Well the original code looked like the following:

```
dtype_24    btst    #8,d0           ; If bit 8 is 0, can't be ADDX/SUBX
            beq.s   t24_not_t30     ; Easy bit done
            move.w  d0,d4           ; Need D4 to hold the op-code

            andi.w  #$00c0,d0       ; Mask bits 7 & 6 of the op-code
            cmpi.w  #$00c0,d0       ; Both set?
            bne.s   t24_not_t30     ; No, skip over type 30 stuff
```

Listing B.1: QLTdis Broken Code

As the original programmer of this code, when I read through it, everything seemed fine - as it always does - but obviously, something was amiss. What to do?

The rest of this exciting article, is a brief foray into the art of debugging using QMON2.

QMON2 is Tony Tebby's original disassembler/monitor tool which allows a QDOSMSQ job code or SuperBasic extension or CALLed code to be debugged by single stepping through the guts of the code until you find the bit that isn't doing what it is supposed to be doing.

I have been using QMON2 to help me debug code for years and although I don't use it as often as I should perhaps, I do happen to like it quite a lot. It seems, unfortunately, that it is no longer available in its English format as Digital Precision still hold the rights to the program - as far as I am aware - but in Germany, you can get a copy from Jochen. Actually, you can get a copy from Jochen in any country in the world, provided you are able to read and understand German manuals.

QMON2 is fine, but as we don't yet have anything like a source code debugger on the QL, it is a bit difficult to figure out where to put breakpoints in your code so that you don't spend ages single stepping through code you know works to find the bit that doesn't work.

George Gwilt has provided a little help here, so not only does he supply you with a neat little assembler but he also gives you help in debugging as well.

When you have assembled the code for QLTdis there is a listing file created with the _LST extension, but another file is created with a _SYM extension. This file holds the goodies we need to debug.

The SYM file is binary and holds a list of all your equates in it, plus a list of all the program labels and their offset from the start of the program. So, if you think that you have a bug in a specific routine, all you have to do is decode the SYM file to extract the routine's offset from the start of the program and set a breakpoint at that place in the code. The problem is, how exactly do you decode the binary file?

George does not document the SYM file format, so you could assemble a few routines and see if you can make any sense of the binary file, but there is a much easier way. Simply by running the SYM_BIN program supplied with GWASL you feed it a SYM file and it spits out a text file holding all the data you will ever need. The output file is named the same as the SYM file but with a further _LST extension, so I have 'dev2_source_qltdis_sym_lst' as my file.

The following is a small extract from this file on my system. Yours may well look different, but don't worry if it does. The first part of the file matches up with my equates:

```
CON_ID                        EQU     $00000000
CON_ID2                       EQU     $00000004
PRT_ID                        EQU     $00000008
PC_ADDR                       EQU     $0000000C
PC_END                        EQU     $00000010
BLACK                         EQU     $00000000
RED                           EQU     $00000002
GREEN                         EQU     $00000004
WHITE                         EQU     $00000007
LINEFEED                      EQU     $0000000A
OOPS                          EQU     $FFFFFFFF
ERR_NC                        EQU     $FFFFFFFF
INFINITE                      EQU     $FFFFFFFF
ME                            EQU     $FFFFFFFF
```

Listing B.2: QLTdis Symbol List

Then we get to the nitty gritty, the labels I have used in my source code and their offsets from the start of the program. The first one is my label 'start' and it is actually the very first instruction in the file, so it has offset zero. Following on are all the other labels I used.

```
START                         EQU     *+$00000000
QLTDIS                        EQU     *+$00000010
JOB_INIT                      EQU     *+$0000003E
EXIT                          EQU     *+$0000003A

.... a few dozen lines removed for brevity!

DTYPE_23                      EQU     *+$00000B9A
DTYPE_24                      EQU     *+$00000BAC

.... another few dozen lines removed for brevity!
```

Listing B.3: QLTdis Symbol List

We can see that regardless of the start address of the program when loaded into memory (by

QMON2 or JMON2) we can still work out where the code for the DTYPE_24 routine, for example, starts simply by adding $0BAC to the actual start address of the program.

The following is a small session showing how I debugged through my DTYPE_24 routine to fix the above mentioned problem.

So, to set the scene, I have edited the source code for the type 24 instructions, assembled QLTdis and produced a new listing of the SYM file. I've looked through the listing and found that my entry point for DTYPE_24 is at offset $0BAC. I then start up JMON2 (in this case, but QMON2 is exactly the same):

```
jmon 'win1_source_qltdis_qltdis_bin'
```

Listing B.4: Debugging QLTdis with Jmon2

If you try this and get an error, make sure you have LRESPR'd the JMON_BIN code for JMON2 or the QMON_BIN code for QMON2 depending on which one you want to use.

When the monitor appears, the very first instruction in the job has already been executed, so I could be anywhere in the job file. Because I have written the code myself, I know what the very first instruction is, it is `BRA.S QLTDIS`. Because I know this, I know that the instruction I am looking at must be the code at label 'QLTDIS'.

If I was debugging some other code that I did not have the original nicely commented source files for, then I would not know where I was in the actual job, or extension, as the first instruction could have sent me off into any location in its own code or even into the ROM.

In this case I have jumped from label 'START' to label 'QLTDIS' and there are quite a few bytes between the two labels. QMON2 is showing me a register dump and the address of, the op-code word and the next instruction to be executed. For the sake of brevity, I've omitted the register dump itself.

```
1A0EB8 6100 BSR.L $1A0EE6
```

So, I'm somewhere in the code for QLTdis, but where. I know I'm at the instruction at address $1A0EB8 but what is the start address of the job itself?

The QMON2 command 'C' will calculate an address and the option 'S' will display the start address of the job.

```
QMON> C S
001A0EA8    1707688
```

This is the Hexadecimal and decimal values for the start of the QLTdis job I'm trying to debug. How can I be sure? Try disassembling the start address for a couple of instructions:

```
QMON> DI S 5
1A0EA8 600E BRA.S $1A0EB8
1A0EAA 0000 ORI.B #0,D0
1A0EAE 4AFB ILLEGAL
1A0EB0 0006 ORI.B #$4C,D6
1A0EB4 5464 ADDQ.W #2,-(A4)
```

The first line is the one to look at, it shows a branch to address $1A0EB8 that QMON2 was showing me originally as the second instruction to be executed. So, the 'S' value does appear to be my label for 'START' and this is what I want.

So, I know that the routine I want to check out is 'DTYPE_24' and that it is at an offset of $0BAC from start, what address is this? Again using the C command to calculate an address, I do this:

```
QMON> C S+$0BAC
001A1A54      1710676
```

I now know where my routine starts, again, to check that it is so, I can disassemble the first few instructions:

```
QMON> DI S+$0BAC 5
1A1A54 0800 BTST #$8,D0
1A1A54 6732 BEQ.S $1A1A8C
1A1A54 3800 MOVE.W D0,D4
1A1A54 0240 ANDI.W $C0,D0
1A1A54 0800 CMPI.W $C0,D0
```

This looks remarkably like the correct code to me, so I can now set a breakpoint at this address and let QMON2 tell me when I get there. Of course, if I was debugging someone else's code, I wouldn't have a handy list of offsets into the program, so I would have to run through it step by step by step until I found out where the code I wanted to check was. Once I'd reached that stage, I would make a note of the address and calculate the offset from the start so that I could easily set a breakpoint there on my next foray into the debugging session. It's much easier when you have the source!

Anyway, I set a breakpoint as follows using the 'B' command.

```
QMON> B S+$0BAC
BRP 1A1A54
```

I could also have simply used the calculated address from earlier by typing 'B $1A1A54' which would have had the same effect. Note that if I set a break point at the same address it will delete the breakpoint at that address. The 'B' command is a toggle.

Again, this is what my code originally looked like when I was debugging the fix for this instruction type:

```
dtype_24    btst    #8,d0           ; If bit 8 is 0, can't be ADDX/SUBX
            beq.s   t24_not_t30     ; Easy bit done
            move.w  d0,d4           ; Need D4 with the op-code

            andi.w  #$00c0,d0       ; Mask bits 7 & 6 of the op-code
            cmpi.w  #$00c0,d0       ; Both set?
            bne.s   t24_not_t30     ; No, skip over type 30 stuff
```

Listing B.5: QLTDis Broken Code

Now I'm ready to go, so I simply type the QMON2 go command which is 'G'.

```
QMON> G
```

The 'G' command means, Go until you hit a breakpoint or finish the program. It causes the program to run at nearly full speed. This means I get all the clear screens and prompts etc that I would normally get when running the program without the debugger. I therefore need to enter a start address and so on to get the disassembler to start working.

I have already loaded a file of assembled `ADDX` and `SUBX` instructions into an area of memory that I allocated with ALCHP and I have its address written down on paper - my own memory is a bit random these days.

After I have typed in the start and end addresses (and the printer device) I return to the QMON prompt with a register dump and the address, hex code and decoded instruction for the next instruction to be executed:

```
At brp SR 0000 --0----- SSP 00028480
D0-D3 0000D300 01BC0924 0000003C FFFFFFFF
D4-D7 0013FFFF 00000000 00000003 0013D300
A0-A3 004C0016 001A1601 001A11DD 001A1130
A4-A7 001A2362 001A11DD 0013A3C8 001A32Fa
1A1A54 0800 BTST #$8,D0
QMON>
```

Taking the above a section at a time, we have this first:

```
At brp SR 0000 --0----- SSP 00028480
```

This is telling me that I'm stopped at a breakpoint - 'at brp' - and the contents of the status register in hex - 0000. Next to that is the interrupt mask value - 0 then 5 dashes showing the current state of the CCR flags. As all are showing dashes, none of the flags are set. Finally, there is the current value of the 'alternative' stack pointer. In this case I'm running in user mode, so I can see the SSP (supervisor stack pointer) value.

Below the status line is a register dump showing the current values of all data and address registers.

```
D0-D3 0000D300 01BC0924 0000003C FFFFFFFF
D4-D7 0013FFFF 00000000 00000003 0013D300
A0-A3 004C0016 001A1601 001A11DD 001A1130
A4-A7 001A2362 001A11DD 0013A3C8 001A32Fa
```

In my case I have the registers split over two lines each for data and address values. This depends on the width of the channel to which QMON2 is writing the register dump.

Below the register dump is the address, the op-code word and the disassembled instruction for the next instruction to be executed. Under that is the QMON2 prompt.

```
1A1A54 0800 BTST #$8,D0
QMON>
```

Back to the debugging session. I want to know what is causing my `ADDX` instructions to be decoded as `ADD`. So, I have my source listing for Type_24 instructions, and because I've hit the breakpoint I set, I know that an `ADDX` is coming through the type_24 decoding routine before jumping into the type_30 decode - or is it? I need to find out.

The register dump shows me the op-code in D0.W and also in D7.W, it is $D300 which is `ADDX D0,D1`. The op-code in binary is as follows, the bit numbers are in HEX above the individual bits themselves:

```
   C    8    4    0
1101 0011 0000 0000
```

Lets trace through the code and see what happens. Remember that the next instruction to be executed is showing just above the QMON prompt, so when I enter the 'T' for Trace command, I will be executing the instruction `BTST #8,D0`. Let's do it.

```
QMON> T
 SR 0000 --0----- SSP 00028480
1A1A58 6732 BEQ.S $1A1A8C
QMON>
```

I'm not showing the register dumps, unless there is anything of interest in the registers.

We have tested bit 8 of D0 and found that it is not zero because the Z flag is not showing in the list of flags. This has to be an `ADDX`, `ADD` or an `ADDA.L` instruction (see the table in my explanation of type_24 decoding above). Let's step again.

```
QMON> T
 SR 0000 --0----- SSP 00028480
1A1A5A 3800 MOVE.W D0,D4
QMON>
```

Nothing of interest here, step again:

```
QMON> T
 SR 0008 --0-N--- SSP 00028480
1A1A5C 0240 ANDI.W #$C0,D0
QMON>
```

Now it's starting to get interesting, the 'N' flag is showing after we moved D0.W to D4.W - this shows that the most significant bit of the new value in D4.W is set and thus the value in D4.W is negative (if using signed arithmetic!). This is how QMON2 displays flags which have been set, the flag letter is displayed on the 'SR' line.

Step again:

```
QMON> T
 SR 0004 --0--Z-- SSP 00028480
D0-D3 00000000 01BC0924 0000003C FFFFFFFF
D4-D7 Ommitted
A0-A3 Ommitted
A4-A7 Ommitted
1A1A60 0C40 CMPI.W #$C0,D0
QMON>
```

So, we have set the Z flag because D0.W is now holding zero (Although D0.L is holding zero, the upper word was already zero only the lower word has changed because the instruction just executed `AND`ed a word value with DO.W.) The next instruction is waiting to be executed so lets do it. Step again:

```
QMON> T
 SR 0009 --0-N--C SSP 00028480
1A1A64 6626 BNE.S $1A1A8C
QMON>
```

It looks like we are going to take the branch as the Zero flag is not set. Lets remind ourselves of what the original source code looked like again:

```
dtype_24    btst     #8,d0          ; If bit 8 is 0, can't be ADDX/SUBX
            beq.s    t24_not_t30    ; Easy bit done
            move.w   d0,d4          ; Need D4 with the op-code word

            andi.w   #$00c0,d0      ; Mask bits 7 & 6 of the op-code
            cmpi.w   #$00c0,d0      ; Both set?
            bne.s    t24_not_t30    ; No, skip over type 30 stuff
```

Listing B.6: QLTdis Broken Code

So you can see where we have single stepped through the above code, and we are just about to jump to label 'T24_NOT_T30' because this instruction is not a type_30. Except, we know that it is an `ADDX` instruction because that is what I was testing, and `ADDX` is a type_30, so what have I done wrong?

I have tested bits 7 and 6 and found them both to be zero (because the Z flag was set after I stepped through the `ANDI.W $C0,D0` instruction. This means that the jump should not be taken to T24_NOT_T30 because I have not yet ascertained that the instruction is not an `ADDX`. With bits 7 and 6 set to 00, I could be looking at `ADDX` or `ADD`. I should not be taking the jump until I have further tested the value in bits 5 and 4 as per my algorithm above.

This could be why the `ADDX` is being decoded as `ADD`, because I have the wrong condition in my test. In order to fix this, I have to change the source code, re-assemble and try my test again. I do this without the QMON2 first of all and if it still fails, I can use QMON2 to try and find out why again. I need to give the current job a 'G' instruction and then I can ESC from the decoding and exit the program.

I shall go do that and report back. Hang on here for a bit ......

Ok, I'm back. I made the change from `BNE.S` to `BEQ.S` and it worked fine. So it looks like I have correctly identified the bug. I need more testing though to make sure I cover all possible op-codes. I have followed up my `ADDX` testing by passing test files which have `ADD`, `ADDA`, `ADDQ` and `ADDI` instructions, along with assorted `SUB` variants and all appears to be working well.

So there you have it, an example of how I manage to get my code wrong and how I can use the tools available to try to sort it out. As I mentioned earlier, QMON2 is available from Jochen for a small fee, but only if you understand German manuals.

Laurence (Lau) Reeves has a different version of QMON2, written by himself, which fixes some bugs but I don't know if this is widely available or if it comes with a manual. Perhaps he could be persuaded to part with it or make it available - who knows. I'm not sure if he ever wrote a manual for it though.

Index

N

P

R

S

T

V