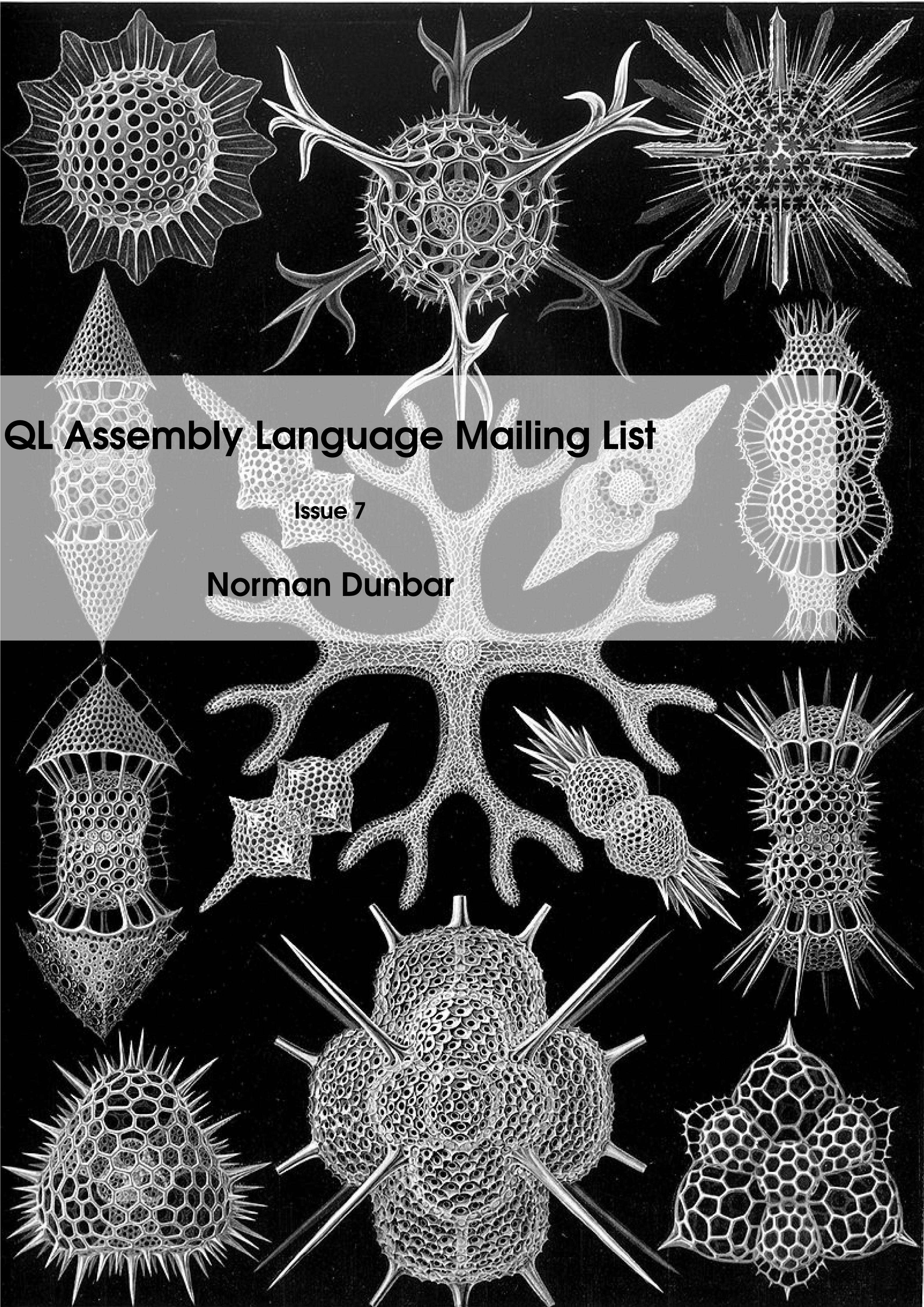


QL Assembly Language Mailing List

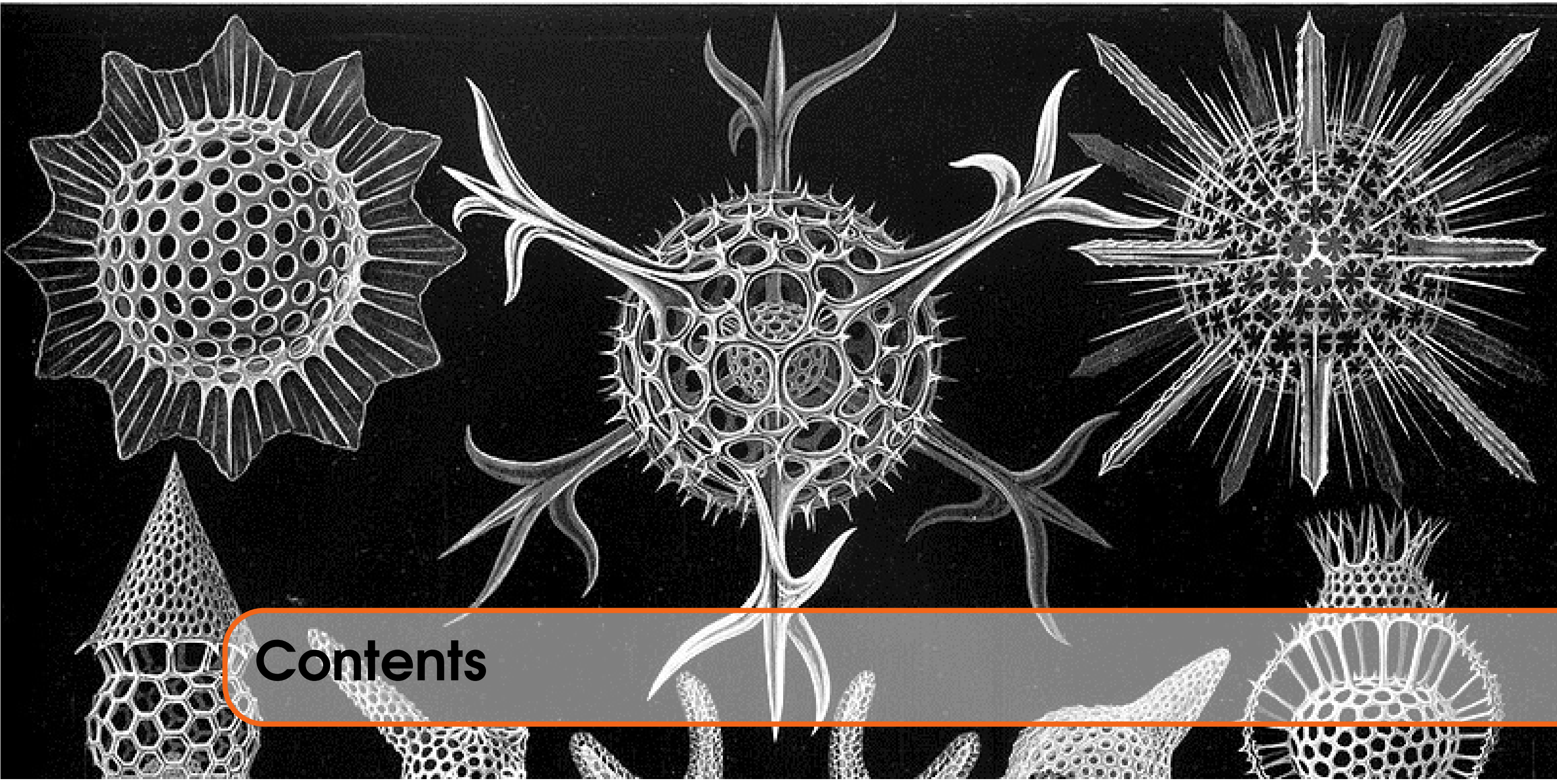
Issue 7

Norman Dunbar

PUBLISHED BY MEMYSELFEYE PUBLISHING ;-)

Download from:
https://github.com/NormanDunbar/QLAssemblyLanguageMagazine/blob/Issue_007/Issue_007/Assembly_Language_007.pdf

This pdf document was created on *27/9/2019* at *16:38*.



Contents

Listings

1. Preface

1.1 Feedback

Please send all feedback to assembly@qdosmsq.dunbar-it.co.uk. You may also send articles to this address, however, please note that anything sent to this email address may be used in a future issue of the eMagazine. Please mark your email clearly if you do not wish this to happen.

This eMagazine is created in LaTeXsource format, aka plain text with a few formatting commands thrown in for good measure, so I can cope with almost any format you might want to send me. As long as I can get plain text out of it, I can convert it to a suitable source format with reasonable ease.

I use a Linux system to generate this eMagazine so I can read most, if not all, Word or MS Office documents, Quill, Plain text, email etc formats. Text87 might be a problem though!

1.2 Subscribing to The Mailing List

This eMagazine is available by subscribing to the mailing list. You do this by sending your favourite browser to http://qdosmsq.dunbar-it.co.uk/mailinglist and clicking on the link “Subscribe to our Newsletters”.

On the next screen, you are invited to enter your email address *twice*, and your name. If you wish to receive emails from the mailing list in HTML format then tick the box that offers you that option. Click the Subscribe button.

An email will be sent to you with a link that you must click on to confirm your subscription. Once done, that is all you need to do. The rest is up to me!

1.3 Contacting The Mailing List

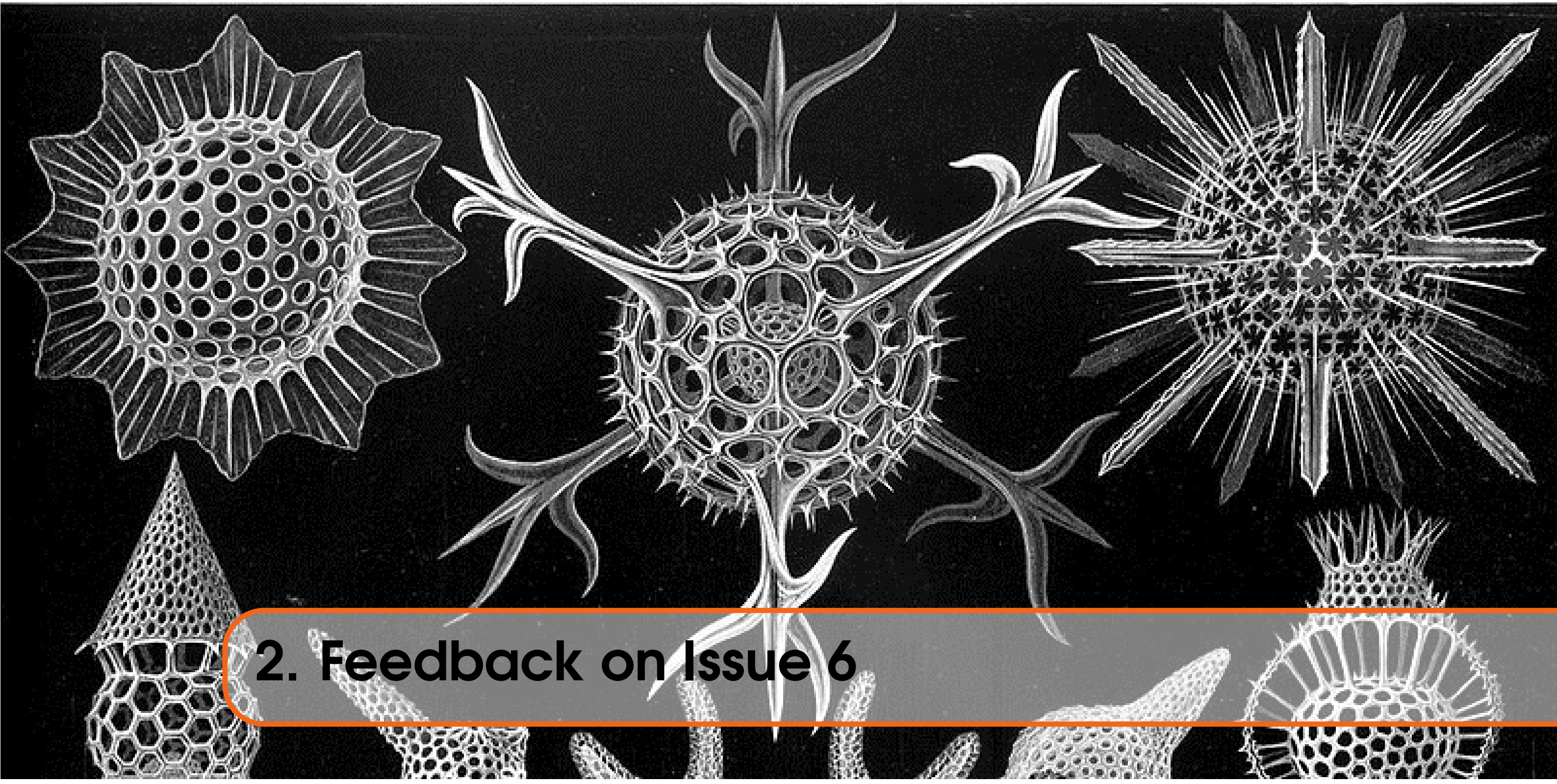
I'm rather hoping that this mailing list will not be a one-way affair, like QL Today appeared to be. I'm very open to suggestions, opinions, articles etc from my readers, otherwise how do I know what I'm doing is right or wrong?

I suspect George will continue to keep me correct on matters where I get stuff completely wrong, as before, and I know George did ask if the list would be contactable, so I've set up an email address for the list, so that you can make comments etc as you wish. The email address is:

`assembly@qdosmsq.dunbar-it.co.uk`

Any emails sent there will eventually find me. Please note, anything sent to that email address will be considered for publication, so I would appreciate your name at the very least if you intend to send something. If you do not wish your email to be considered for publication, please mark it clearly as such, thanks. I look forward to hearing from you all, from time to time.

If you do have an article to contribute, I'll happily accept it in almost any format - email, text, Word, Libre/Open Office odt, Quill, PC Quill, etc etc. Ideally, a LaTeXsource document is the best format, because I can simply include those directly, but I doubt I'll be getting many of those! But not to worry, if you have something, I'll hopefully manage to include it.



2. Feedback on Issue 6

2.1 No Feedback so far!

3. The Fastest Scrolling in the West

I'm very grateful to ***Tobias Fröschle*** who submitted this article for publication.

It concerns the various ways in which the Q68 can move memory around. It appears that the Q68 has a lot of memory, and doing simple things like scrolling the screen around can take quite some time.

I hope you enjoy the following.

3.1 Messing Around with the Q68

While Norman tends to write his stuff in GWASS, my favourite assembler is QMac. The choice is mainly a matter of taste – GWASS overall has similar features to QMac. So bear with me, the code examples here will be in QMac lingo.

In the passing time between Christmas and back to work called "between years" in Germany, there was a bit of time to mess around with the Q68 and the trusty QMac Assembler. I was always a bit concerned how the Q68 can handle the massive amounts of memory that need to be shoved around in order to handle a high-colour screen.

My favourite resolution on the Q68 is the high colour mode with 512 by 384 pixels. One pixel takes 16 bits in this resolution, a 68000 word. That makes 1kBytes per scan-line, all in all 384kBytes for the whole screen. Scrolling this screen to the left by one pixel, for example, requires moving 384 x (1024 - 2) bytes of memory, scrolling the whole screen to the left by 512 pixels with a one-pixel increment to create smooth animation requires 384kBytes * 512 times to be moved – a whooping 192Mbytes of memory shoved around. (In a game, for example, you would, however, scroll in larger increments to speed up things, normally.)

All the below experiments will work on the Q68 or on QPC2 (provided you set the screen resolution to 512 x 384 and 16-bit colour.). The screen start address will be different, though. (You can find out with the SCR_BASE S*BASIC command).

To put things in perspective: This action results in roughly 12 times more memory to shove around than an original Black Box would need to do for the same action. Granted, on the Q68 we don't need to shift the screen words themselves to scroll horizontally, which makes matters a bit simpler (thus faster), but it is still a huge task. I just wanted to see how the Q68 would cope with this.

3.2 The Straight-Forward Approach

Let's start simple (or, should I call that naïve?): Two nested loops, the innermost moves one scan-line one pixel to the left using two address registers, the outermost iterates over all scan-lines. Call that routine 512 times and we're done:

```
; Scrolls the screen one pixel to the left
Lscroll
        movem.l  a0-a1,-(sp)
        lea      screen_start,a0
        lea      2(a0),a1
        move.w   #384-1,d1          ; 384 scan-lines

lineLoop
        move.w   #512-1,d0          ; 512 pixels to move

cpy_loop
        move.w   (a1)+,(a0)+
        dbf      d0,cpy_loop
        dbf      d1,lineLoop
```

Listing 3.1: Scrolling one pixel leftwards

We're at 90 seconds now to scroll a screen across the whole screen width and the scrolling looks, admittedly, pretty lame (remember, that is moving 192 megabytes of memory...). The first improvement that comes to mind is a long-word move in cpy_loop which would allow us to save half of the inner loop iterations. Should be like 30-50% faster on a real 68000. On a Q68, it unfortunately isn't for some reason. In fact, it is only a few seconds faster and not really a significant improvement. Time to look for some more drastic means to speed things up:

3.3 Unrolling loops (or: How to waste Precious Amounts of Memory)

What slows the straightforward approach down quite a bit are the two nested loops (one per width of screen, one per height of the screen). If we could get rid of these, or at least one of them, we should achieve a significant improvement. And, in fact, we can. The Q68 has so much memory that we can put that to good use: Instead of looping around one single longword move, we can write all the 256 iterations in a row into our source code, voilà, the inner loop is gone. Because programmers are lazy and writing 256 identical statements is a bit boring, it is now time to show the interested (?) reader what the "Mac" in QMac is good for: Time for some macro trickery.

```
REPT     MACRO num,args
         LOCAL count,pIndex,pCount
Count    SETNUM 1
Lp       MACLAB
pCount   SETNUM [.NPARAMS] - 1
pIndex   SETNUM 2
pLoop:   MACLAB
         EXPAND
```

```
           [.PARAM([pIndex])]
           NOEXPAND
pCount     SETNUM  [pCount]-1
pIndex     SETNUM  [pIndex]+1
           IFNUM   [pCount] >= 0 GOTO pLoop
Count      SETNUM  [count] + 1
           IFNUM   [count] <= [num] GOTO lp
           ENDM
```

Listing 3.2: The REPT Macro

If this is all Chinese for you, the whole macro simply repeats the text you give it as second to last argument(s) the amount of times you give as the first, like

```
NotUseful
     REPT      256,{    nop},{    clr.l d0}
```

Listing 3.3: A simple REPT example

Will expand to 256 NOP and CLR.L D0 instructions in your code. The GOTO directives don't do anything in your finished program, but rather have the assembler running in circles producing source code for you (nice, isn't it?). The outer loop starting at *Lp* iterates over the parameter list the amount of times you give as first parameter, the inner loop at *pLoop* over the parameter list. Ideal stuff for lazy programmers.

Note

The macro would look a bit different when written in GWASS which uses a similar, but slightly different macro syntax (That I don't happen to be familiar with, unfortunately (and I should really work on my writing style – That looks like a programmer's))).

Now back to our screen scrolling problem: We wanted to unroll the inner loop which iterates over the pixels in one single scan-line to get rid of the inner loop. So, let's place that macro invocation (incantation?) in place of that inner loop, replacing it with 256 long word move instructions:

```
; Scrolls the screen one pixel to the left
Lscroll
     movem.l  a0-a1,-(sp)
     lea      screen_start,a0
     lea      2(a0),a1
     move.w   #384-1,d1          ; 384 scan-lines

lineLoop
     REPT     256,{    move.l (a1)+,(a0)+}
     dbf      d1,lineLoop
```

Listing 3.4: Unrolling the innner loop

The REPT invocation looks unremarkable, but if you have a look at the produced assembly listing, you will find that the assembler has just expanded the macro to 256 lines of code, effectively replacing that inner loop (this also blew our code for that loop from xxx to yyy bytes. But after all, we are on a Q68 or QPC and have plenty of memory to trade for).

If you run the above code, you will find it runs about three times faster than the previous version, so we have bought execution speed for memory. Want to drive this a bit further by unrolling the outer loop as well? Try something like

```
screenLongs EQU 512*384*2/4
    REPT     [screenLongs],{      move.l  (a1)+,(a0)+}
```

Listing 3.5: Unrolling the outer loop

But that might be a little ridiculous, so I have left this as exercise to the reader (Ha! I always wanted to use this sentence somewhere).

Can we still do better? Sure.

3.4 MOVEM.L Can Work in Other Places Other Than the Stack

There is one instruction in the 68k instruction set that can shove memory about in large chunks – The MOVEM instruction. You would normally use it to save and restore registers to and from the stack in subroutines, but its use is not restricted to that. In cases where you have many registers to spare, you can also use it to implement large block moves.

There's just one single caveat: The MOVEM instruction does not work with a "post-increment" we would need to do a block move, so a simple

```
    movem.l  (a0)+,REGSET
    movem.l REGSET,(a1)+        ; this instruction does not exist
```

Listing 3.6: MOVEM restrictions

will unfortunately not work, so, in order to repair this, we need to increment the target register with a separate instruction.

So, let's assume you can spare (or free up) registers d3-d7 and address registers a2-a6 in our scrolling routine, we can move a whoopy 40 bytes per instruction like in (note the backslash in a macro invocation is understood as a line continuation character in QMac)

```
    REPT       25,{     movem.l  (a1)+,d3-d7/a2-a7},\
                  {     movem.l  d3-d7,a2-a6,(a0)},\
                  {     adda.l  \#40,a0}
```

Listing 3.7: Improving the REPT macro

This time our macro receives 4 arguments, the repetition count and the three lines to repeat. The macro magic will repeat these three lines 25 times in an unrolled loop, creating copy commands for 250 longwords. Oops, 6 missing to a complete scan-line, so add a

```
    REPT       6,{     move.l  (a1)+,(a0)+}
```

Listing 3.8: Scrolling one pixel leftwards

after it to create code to move the last 6 long words of a scan-line.

This is only marginally faster as the above unrolled loop on a Q68, but saves a significant amount of code space with an even (slightly) better runtime speed. I was actually expecting a bit more speedup, but Q68 instruction timings seem to differ from the original 68k.

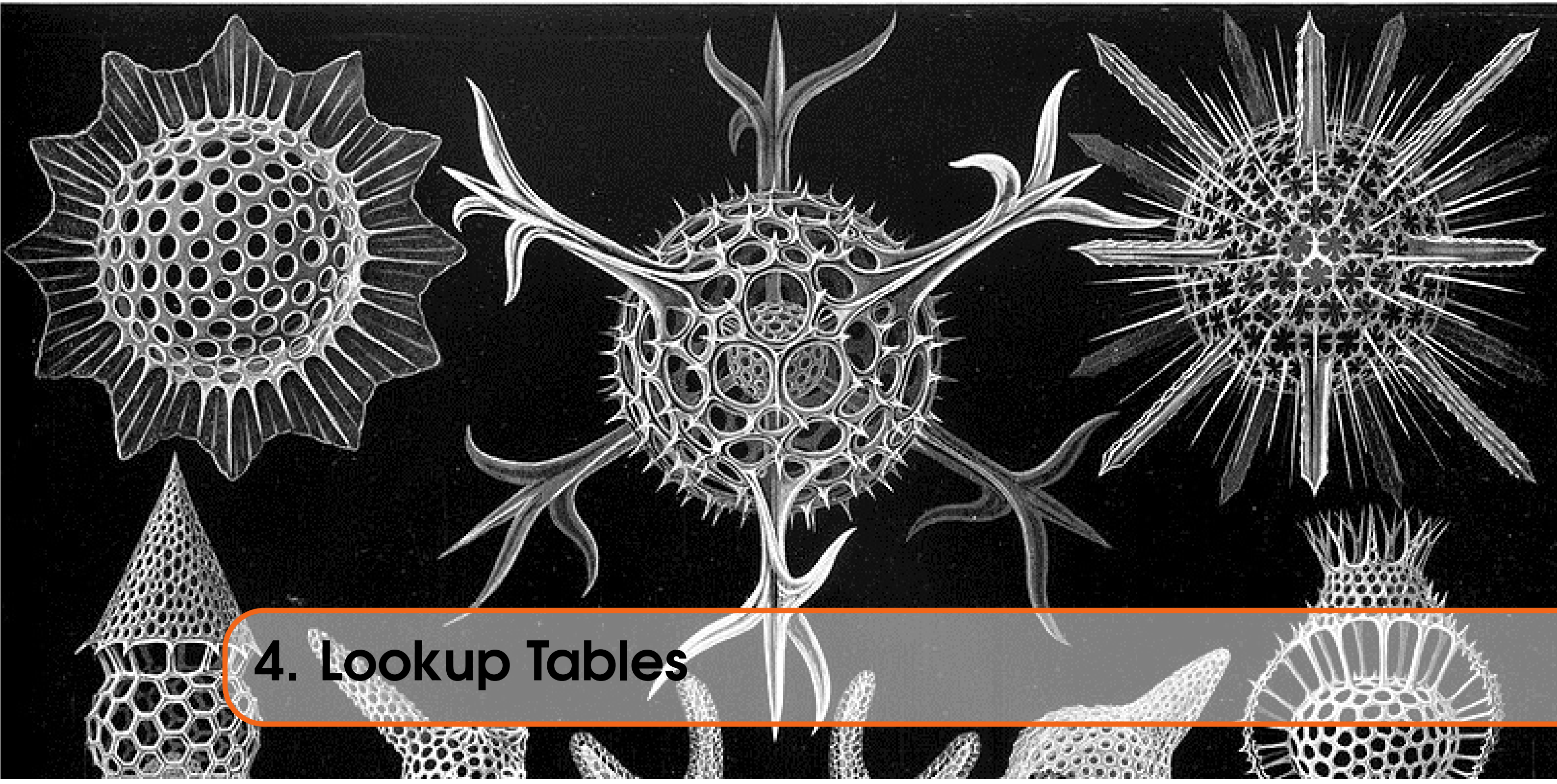
The MOVEM block move is the fastest way to move large chunks of memory around using a 68000 CPU (In case you happen to know anything faster, I'd like to hear from you), so, we're at the end here. Really? No, not quite:

3.5 If Software Can't Cope, Use Hardware

If you want to speed up the scrolling even further, you can use the SD memory in the Q68. This is a small (read: scarce, about 12k) amount of very fast memory that can be used for time-critical routines.

Code like the above (that mainly accesses "slow" memory) can be expected to run about three to four times faster in Q68 SD RAM than in the normal DRAM areas. As the amount of space available in fast memory is limited (some of it is already used by SMSQ/E as well), you might want to keep the usage of fast memory as low as possible. Also note that, just like the RESPR area, it is not possible to release space in fast memory once it has been allocated. A game, for example, could however easily argue that you would reset the computer anyway after finishing.

My tests resulted in about a three-fold speed increase once the above routines were copied to fast memory and executed from there.



4. Lookup Tables

Lookup tables are useful. Remember when you were at school and had to find the logarithm of a number? You didn't have to calculate it every time it was needed, someone else did it for you and put the details in a booklet[1]. When writing code it's sometimes useful to use lookup tables rather than doing a possibly resource intensive calculation each and every time.

The rest of this section shows a couple of uses for lookup tables.

4.1 Bits and Bobs

Here's a sequence of 10 numbers, they are all integers:

```
0, 1, 1, 2, 1, 2, 2, 3, 1, 2 ...
```

Q1: Do you know what the next value in the sequence will be?

Q2: Do you know what the above sequence represents?

Would it help if I told you that the formula to calculate the value for number 'n' in the sequence is given by:

```
Value(n) = (value(int(n/2)) + (n and 1)
```

For example, to find the value of the number 10, the 11th number in the sequence as we start from 0, and which just happens to be the answer to Q1 above, we must take value(5) and add on bit 0 of 10. Of course, we need then to find the answer to Value(2) and add on bit 0 of 5 and so on. Recursion anyone? This works out as the following sequence of calculations:

[1]Ok, I'm *probably* showing my age here - calculators were not invented/easily available until after I was in secondary school! We had a booklet of log tables to look up.

```
Value(10) = (value(5) + (10 and 1)
Value(5)  = (value(2) + (5 and 1)
Value(2)  = (value(1) + (2 and 1)
Value(1)  = (value(0) + (1 and 1)
Value(0)  = 0
```

This gives us, working backwards up the above sequence of calculations:

```
Value(0)  = 0
Value(1)  = 0 + 1 = 1
Value(2)  = 1 + 0 = 1
Value(5)  = 1 + 1 = 2
Value(10) = 2 + 0 = 2
```

So, the 11th number in the sequence, aka value(10), is 2. That answers Q1, Q2 will be answered soon, I promise.

Assuming you need to know these numbers in a program you happen to be writing in assembly language, you could work them out each time. The formula does tend to imply recursion is required and the following brief section of code will do exactly that.

```
; On Entry (to Value routine) :
;   D0.B = Required value for 'n'.
;
; On Exit:
;   D1.B = Answer (Value(n)).
;
; All registers are preserved except D1 and D0.
;
; Enter at start for a demo with N = 10. Enter at
; Value, with D0 holding the required byte value, to
; calculate the result for that value.
;
Start   moveq #10,d0          ; N = 10
        bsr.s Value           ; Get recursive

; Result is now in D1.B.

Back    moveq #0,d0     ; No errors
        rts

Value   tst.b d0              ; N = 0 yet?
        bne.s More            ; Not yet
        moveq #0,d1           ; Yes Value(0) = 0
        rts

More    move.w d0,-(a7)       ; Save current N
        lsr.b #1,d0           ; INT(N/2)
        bsr.s Value           ; Recurse

; On return to here, D1.B holds the Value(N/2) result.

rtnHere move.w (a7)+,d0       ; Current N again
        btst #0,d0            ; Anything to add in bit 0?
        beq.s  Done             ; No, even number.
        addq.b #1,d1            ; Yes, add bit 0 of N

Done    rts
```

Listing 4.1: Calculating values with recursion

So, what happens in the above when we use 10 as the required value?

1. At the label Value, D0 = 10 and the stack contains the return address of label Back, and the return to SuperBasic address. The stack looks like this:

   ```
   SuperBasic
   Back
   ```

2. As D0 is not yet zero, we end up at label More where we stack D0, shift it right to get 5, and call Value again. At Value, the stack looks like this:

   ```
   SuperBasic
   Back
   10
   rtnHere
   ```

3. As D0 is not yet zero, we end up at label More where we stack D0, shift it right to get 2, and call Value again. At Value, the stack looks like this:

   ```
   SuperBasic
   Back
   10
   rtnHere
   5
   rtnHere
   ```

4. As D0 is not yet zero, we end up at label More where we stack D0, shift it right to get 1, and call Value again. At Value, the stack looks like this:

   ```
   SuperBasic
   Back
   10
   rtnHere
   5
   rtnHere
   2
   rtnHere
   ```

5. As D0 is not yet zero, we end up at label More where we stack D0, shift it right to get 0, and call Value again. At Value, the stack looks like this:

   ```
   SuperBasic
   Back
   10
   rtnHere
   5
   rtnHere
   ```

```
2
rtnHere
1
rtnHere
```

6. At Value, D0 is now zero, so we store zero in D1 and return to rtnHere.
7. At rtnHere, we unstack 1 into D0. The stack now looks like:

```
SuperBasic
Back
10
rtnHere
5
rtnHere
2
rtnHere
```

   As D0 is odd, we add 1 to D1. The running total is now 1. Then we execute an RTS instruction and end up back at rtnHere.
8. At rtnHere, we unstack 2 into D0. The stack now looks like:

```
SuperBasic
Back
10
rtnHere
5
rtnHere
```

   As D0 is even, we don't add 1 to D1. The running total is still 1. Then we execute an RTS instruction and end up back at rtnHere.
9. At rtnHere, we unstack 5 into D0. The stack now looks like:

```
SuperBasic
Back
10
rtnHere
```

   As D0 is odd, we add 1 to D1. The running total is now 2. Then we execute an RTS instruction and end up back at rtnHere.
10. At rtnHere, we unstack 10 into D0. The stack now looks like:

```
SuperBasic
Back
```

   As D0 is even, we don't add 1 to D1. The running total is still 2. Then we execute an RTS instruction and end up back at Back.
11. At Back we clear D0 and return to SuperBasic. The value in D1 is 2, which is the correct value for the 11th number in the sequence.

The test code above is fine if you only need one or two values, but if your code needs lots, then a lookup table would be a good trade off between memory usage - you need extra space for the table

- and CPU resources - if you have to do lots of calculations each time. The following code sets up a lookup table for all values from 0 to 255 - so that's a good reason for having a single byte for each value.

```
; Lookup Table initialisation.
;
; Register Usage:
; D0.B = 'N' counter (0 - 255).
; D2.B = INT(n/2), value(N).
; A2.L = Pointer to start of lookup table.

Entry   bra Start               ; Skip the lookup table

Lookup  ds.b 256          ; Lookup table

Start   moveq #0,d0             ; Value(0)
        lea Lookup,a2           ; Guess!
        move.b d0,(a2)          ; Save value(0) in table

Loop    addq.b #1,d0            ; Next 'n'
        bcs.s Done              ; Bale out at 256
        move.w d0,d2            ; Copy 'n' to D2
        lsr.w #1,d2             ; INT(n/2)
        move.b (a2,d2.w),d2     ; Value(INT(n/2))
        btst #0,d0              ; Anything to add?
        beq.s Store             ; No, just store value(n)
        addq.b #1,d2            ; Yes, add bit 1

Store   move.b d2,(a2,d0.w)     ; Store Value(n)
        bra.s Loop              ; Not done yet

Done    moveq #0,d0             ; No errors
        rts
```

Listing 4.2: Initialising the lookup table

If the program initialises the lookup table during startup, then any time it needs to extract a value, it's as simple as:

```
        ...
        move.w #n,d0      ; D0 must be 0 - 255
        lea Lookup,a2
        move.b (a2,d0.w),d0 ; Value(d0.b)
        ...
```

Listing 4.3: Using the lookup table to find a value

At this point, D0.B holds the result of Value(n). Keep in mind that the lookup table only gives values between 0 and 255, but D0 is a word in the above for ease of indexing the table.

So, what's it all about I hear you ask? It's simple, the sequence I gave you way back at the beginning is the number of '1' bits in any byte value.

Taking 10 as an example, it is $0000\ 1010_{bin}$while 5, half of 10, is $0000\ 0101_{bin}$- the same number of bits. So, that works for even numbers, how about odd ones? Well, half of 5 is 2.5 bit as we are rounding down, that's 2. Two is $0000\ 0010_{bin}$ Doubling 2 gives 4 or $0000\ 0100_{bin}$and 5 is just 4 plus 1. So, the number of bits in an odd number is still the same as the number in half of it, plus bit 0. Simples[2].

4.2 Character Characteristics

Another useful lookup table would be one which, again, covers 256 byte entries. However, instead of values, these bytes contain up to 8 bits of ‘flag’ information. In the C/C++ programming languages, there are numerous functions (and also, macros with the same name) which can be used to determine if a character is a digit, upper case, lower case, printable etc. This is done with a lookup table of bit flags.

Each character class (numeric, alphabetic etc) has one or more bits set in the table entry to indicate if this character is indeed a digit, upper case etc. In C68 (look in the header file `ctypes.h`) we have a number of bit masks defined, as follows, although I am using better names than the C68 code!

```
UPPERCASE   equ 1      ; Bit 0 = A - Z
LOWERCASE   equ 2      ; Bit 1 = a - z
DIGIT       equ 4      ; Bit 2 = 0 - 9
SPACE       equ 8      ; Bit 3 = space, tab, linefeed
PUNCTUATION equ 16     ; Bit 4 = .,;: etc
CONTROL     equ 32     ; Bit 5 = Codes < 32
BLANK       equ 64     ; Bit 6 = space, tab
HEXDIGIT    equ 128    ; Bit 7 = A - F, a - f
```

Listing 4.4: Character attribute bit masks

So, in the lookup table for the English language, every entry between CODE(’A’) and CODE(’Z’) will have the UPPERCASE flag, bit 0, set. They will also have the HEXDIGIT flag, bit 7, set for ‘A’ through ‘F’.

Now, I don’t know about you, but I really don’t fancy typing in 256 entries in a table, with the possibility of getting it wrong, somewhere. That’s a nightmare scenario, so the QL can do it for me (you, on the other hand, can simply download the code for this issue and get it for free!) I wrote the following, simple, C68 code to generate the file I needed for assembly routines, using my own constant values as listed above.

The following is the listing of the C68 program, `characters_c`:

```
#include <stdio.h>
#include <ctype.h>

int main(int argc, char *argv[]) {
    int x;

    printf("UPPERCASE   equ 1     ; Bit 0 = A - Z\n");
    printf("LOWERCASE   equ 2     ; Bit 1 = a - z\n");
    printf("DIGIT       equ 4     ; Bit 2 = 0 - 9\n");
    printf("SPACE       equ 8     ; Bit 3 = space tab etc\n");
    printf("PUNCTUATION equ 16    ; Bit 4 = .,;: etc\n");
    printf("CONTROL     equ 32    ; Bit 5 = Various\n");
    printf("BLANK       equ 64    ; Bit 6 = space tab\n");
    printf("HEXDIGIT    equ 128   ; Bit 7 = 0 - 9 a - f A - F\n");
    printf("ALPHABETIC  equ UPPERCASE + LOWERCASE\n");
    printf("ALPHANUMERIC equ ALPHABETIC + DIGIT\n");
```

[2] As the odd, occasional, passing meerkat has been know to utter!

```
    printf("PRINTABLE      equ BLANK + PUNCTUATION + ALPHABETIC + DIGIT\n
⟹ ");
    printf("GRAPHIC        equ PUNCTUATION + ALPHABETIC + DIGIT\n");

    printf("\n\nchartab     ");
    for (x = 0; x < 256; x++) {
        printf("dc.b 0 ");
        if (iscntrl(x)) printf("+ CONTROL ");
        if (isupper(x)) printf("+ UPPERCASE ");
        if (islower(x)) printf("+ LOWERCASE ");
        if (isdigit(x)) printf("+ DIGIT ");
        if (isxdigit(x)) printf("+ HEXDIGIT ");
        if (ispunct(x)) printf("+ PUNCTUATION ");
        if (isspace(x)) printf("+ SPACE ");
        if (x == 9 || x == 32) printf("+ BLANK ");
        printf("      ; CHR$(%d) = '%c'\n              ", x,
               isprint(x) ? x : '.');
    }
        return 0;
}
```

Listing 4.5: C68 utility: characters_c

The code above, compiled to `characters_exe`, generates a file that I can use in my assembly code. It does it much faster than I can, and more accurately to boot.

Note that C68 on the QL doesn't have the function isblank, so I've hard coded the only two values that that function applies to, tab (9) and space (32). C68 gives the following character attributes:

UpperCase 65 through 90, 'A' through 'Z';
LowerCase 97 through 122, 'a' through 'z';
Digit 48 through 57, '0' through '9';
Hex Digit 48 through 57, 65 through 70, 97 through 102, '0' through '9', 'A' through 'F', 'a' through 'f';
WhiteSpace 9 through 13, 32, Tab through Carriage Return, Space;
Blank 9 and 32, Tab and Space;
Control 33 through 47, 58 through 64, 91 through 96, 123 through 126, 128 through 191.
Puntuation 33 through 47, 58 through 64, 91 through 96, 123 through 126, 128 through 191;

The top of the generated file, which I named `characters_asm_in`, resembles the following:

```
UPPERCASE     equ 1     ; Bit 0 = A - Z
LOWERCASE     equ 2     ; Bit 1 = a - z
DIGIT         equ 4     ; Bit 2 = 0 - 9
SPACE         equ 8     ; Bit 3 = space tab etc
PUNCTUATION   equ 16    ; Bit 4 = .,;: etc
CONTROL       equ 32    ; Bit 5 = Various
BLANK         equ 64    ; Bit 6 = space tab
HEXDIGIT      equ 128   ; Bit 7 = 0 - 9 a - f A - F
ALPHABETIC    equ UPPERCASE + LOWERCASE
ALPHANUMERIC  equ ALPHABETIC + DIGIT
PRINTABLE     equ BLANK + PUNCTUATION + ALPHABETIC + DIGIT
GRAPHIC       equ PUNCTUATION + ALPHABETIC + DIGIT

chartab     dc.b 0 + CONTROL             ; CHR$(0) = '.'
            dc.b 0 + CONTROL             ; CHR$(1) = '.'
            dc.b 0 + CONTROL             ; CHR$(2) = '.'
```

```
                dc.b 0 + CONTROL                ; CHR$(3) = '.'
                dc.b 0 + CONTROL                ; CHR$(4) = '.'
                dc.b 0 + CONTROL                ; CHR$(5) = '.'
                dc.b 0 + CONTROL                ; CHR$(6) = '.'
                dc.b 0 + CONTROL                ; CHR$(7) = '.'
                dc.b 0 + CONTROL                ; CHR$(8) = '.'
                dc.b 0 + CONTROL + SPACE + BLANK ; CHR$(9) = '.'
                dc.b 0 + CONTROL + SPACE         ; CHR$(10) = '.'
                dc.b 0 + CONTROL + SPACE         ; CHR$(11) = '.'
                dc.b 0 + CONTROL + SPACE         ; CHR$(12) = '.'
                dc.b 0 + CONTROL + SPACE         ; CHR$(13) = '.'
                ...
```

Listing 4.6: Extract of the generated file `characters_asm_in`

Beware, however, if you view the generated file in an operating system that is not QDOSMSQ because some of the QL character codes represent “invalid” characters in some character sets, on PCs or Linux, for example.

So, now that the table has been created, we need some assembly code to call when we want to check if, for example, a character code is a digit. Those character attribute functions would look like the following. My file is named `charAttr_asm_in`:

```
; All these functions require a character code in D0.B and will
; return D0 = 0 if the character is invalid, otherwise, D0.B will be
; a relatively random non-zero value.
;
; ENTRY Registers:
;   D0.B Character code to be tested
;
; EXIT Registers:
;   D0.B Zero - Character test failed. (Z flag set)
;        non-zero - Character test passed.

    in win1_source_characters_asm_in


; Given a character code in D0.B, extract the character attributes
; bitmap from chartab into D0.B.
;
; Mask the attribute bitmap with the desired attribute mask to get
; the validation result.
;
; Return the result in D0.B with Z set if the test FAILED.

; On the stack we have D1.W.
; D1.B = required mask
; D0.B = character code
isanything move.l a2,-(a7)              ; Save the worker
           lea chartab,a2               ; Character attributes table
           ext.w d0                     ; D0 must be a word wide
           move.b (a2,d0.w),d0          ; Attributes bitmap byte
           and.b d1,d0                  ; Do attributes match?
           move.l (a7)+,a2              ; Restore worker
           move.w (a7)+,d1              ; Restore the other worker
           tst.b d0                     ; Z = test failed
           rts
```

```


; These just set up the mask we want in D1.W, and jump off to the
; common code above. The unstacking of D1.W and return to caller
; is done above.
isdigit     move.w d1,-(a7)              ; Save the first worker
            move.b #DIGIT,d1             ; Required attribute mask
            bra.s isanything             ; Never return here!

isalpha     move.w d1,-(a7)              ; Save the first worker
            move.b #ALPHABETIC,d1        ; Required attribute mask
            bra.s isanything             ; Never return here!

isalnum     move.w d1,-(a7)              ; Save the first worker
            move.b #ALPHANUMERIC,d1      ; Required attribute mask
            bra.s isanything             ; Never return here!

isupper     move.w d1,-(a7)              ; Save the first worker
            move.b #UPPERCASE,d1         ; Required attribute mask
            bra.s isanything             ; Never return here!

islower     move.w d1,-(a7)              ; Save the first worker
            move.w #LOWERCASE,d1         ; Required attribute mask
            bra.s isanything             ; Never return here!

isxdigit    move.w d1,-(a7)              ; Save the first worker
            move.b #HEXDIGIT,d1          ; Required attribute mask
            bra.s isanything             ; Never return here!

ispunct     move.w d1,-(a7)              ; Save the first worker
            move.b #PUNCTUATION,d1       ; Required attribute mask
            bra.s isanything             ; Never return here!

iscntrl     move.w d1,-(a7)              ; Save the first worker
            move.b #CONTROL,d1           ; Required attribute mask
            bra.s isanything             ; Never return here!

isgraph     move.w d1,-(a7)              ; Save the first worker
            move.b #GRAPHIC,d1           ; Required attribute mask
            bra.s isanything             ; Never return here!

isprint     move.w d1,-(a7)              ; Save the first worker
            move.b #PRINTABLE,d1         ; Required attribute mask
            bra.s isanything             ; Never return here!

isspace     move.w d1,-(a7)              ; Save the first worker
            move.b #SPACE,d1             ; Required attribute mask
            bra.s isanything             ; Never return here!

isblank     move.w d1,-(a7)              ; Save the first worker
            move.b #BLANK,d1             ; Required attribute mask
            bra.s isanything             ; Never return here!
```

Listing 4.7: Character attributes library - `charAttr_asm_in`

How these work is pretty simple:

- We enter with the character code to be tested in D0.B, as we will be about to trash it, we save D1.W on the stack prior to loading its low byte with the required attribute mask that we need for the current test.
- A branch is then made to the common code which saves A2.L as we will be using it. The character's attribute bitmap is then extracted from the table. This bitmap is appropriate to the character code originally in D0.B but which we have now extended to word sized to index into the attribute bitmap table.
- The attribute bitmap is ANDed with the desired attribute mask and the result in D0.B will be zero if there are no common bits in the two masks - the test has failed, or non-zero if at least one pair of common bits matched.
- The stack is then tidied and we return to the caller with the Z flag set to indicate a *failure*, unusually, or unset to show that the character code in D0.B was a character which belonged to the attribute set we were interested in - a digit, an upper case letter etc.

In your code, this can be used as follows:

```
        in charAttr_asm_in

        ...
        move.b(a2),d0           ; Get character code from buffer
        bsr isalnum             ; Is it a letter or digit?
        beq.s notAlnum          ; No, it's not
        ...
```

Listing 4.8: Using the `charAttr_asm_in` routines

This code is useful when writing something like a lexer (part of a compiler, assembler etc) or where you are processing text for some reason. It can save you having to check that the character in D0.B is less than or equal to 'Z' and greater or equal to 'A' or less than or equal to 'z' or greater than or equal to 'a' - and so on. (Yes, I know, the 68020 has the CMP2 instruction which makes this easier.)

4.2.1 A Final Thought

If necessary, the 256 byte table of attributes could be created, then saved as a binary file and binary included into your application's code, using the appropriate command for your assembler. On GWASS this is the LIB or the INCBIN command.

For homework, you could convert the character attribute functions to be SuperBASIC extensions? If you feel the need? Maybe?

5. UTF8 and the QL

UTF8 is a character set much loved, perhaps, by Linux, MacOS and increasingly, Windows computers. As it happens, most of the HTML pages, as well as almost all XML files, are themselves in UTF8 format. What is it and how does it affect the QL?

I spend more time editing files, at least to get a first draft, in Linux. When I copy the files up to my QPC session and open them in QD, a couple of things happen:

- QD converts all my runs of 4 spaces to a tab character, even though I've repeatedly asked it not to. I'm rapidly losing patience with QD!
- Some of the QL characters, happily typed on Linux, are shown as weird blobs in QD. The UK Pound sign, for example, or the Euro are blobs in QD when they were fine on Linux. Why?
- Writing QL files back to, say, DOS1_, then opening them in a Linux editor shows many characters as the UTF8 character with Code Point U+0000, the black blob with a question mark in it. Oops! Don't even try opening a QL file with the arrow characters within, you don't want to go there!

5.1 UTF8 Encoding

UTF8 is an encoding standard for plain text. It is a multi-byte character set which simply means that some characters in the set, take up more than one byte when viewed "in the raw" (or with a hex dump). UTF8 has a big enough encoding method that all (I am led to believe) the characters in all the languages of the world, plus all their punctuations, numbers and so on, can be represented.

UTF8 characters can be 1, 2,3 or 4 bytes long. The UK Pound sign, for example, is two bytes - $C2A3, the Euro symbol is three bytes - $E282AC, while the humble digit seven remains as a single byte - $37.

The rules are simple:

- Each character has what is known as a "code point" and is represented by the expression "U+nnnn" where the "nnnn" part may be two, three or four hex pairs. Single byte characters, like the digits, are shows also as "U+nnnn" but the first two digits are zeros - "U+0037" for our digit seven.
- ASCII characters, below 128, are represented in UTF8 by a single byte, exactly the same as the current ASCII byte. Handy! Not on a QL of course! Code points U+0000 through U+007F are represented here.
- ASCII characters above 128 are split into three groups.
  - Code points from U+0080 through U+07FF are all two bytes long.
  - Code points from U+0800 through U+FFFF are all three bytes long.
  - Code points from U+10000 through U+10FFFF are all 4 bytes long.

So, how do we encode an ASCII character onto one, two, three or four bytes of UTF8? Easy!

- In ASCII, all characters with the top bit (bit 7) clear will have their UTF8 code point value, encoded into the lower 7 bits of a single byte. In other words, 0xxxxxxx, allowing 7 bits to encode the code point.
- Two byte UTF8 characters have the layout 110xxxxx 10xxxxxx, and this allows for 11 bits to encode the code point within the two bytes.
- Three byte UTF8 characters have the layout 1110xxxx 10xxxxxx 10xxxxxx, allowing for 16 bits of code point information.
- Finally, four byte UTF8 characters have the layout 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx allowing for a massive 21 bits of code point values.

So, how does that work for our examples, the digit seven, UK Pound and the Euro symbol?

The digit seven is a single byte, and is simply the current ASCII value, $37, because that already has the top bit clear and the remaining bits holding the ASCII character, or the UTF8 code point as it is now known.

The UK Pound, has code point U+00A3. This is higher than the highest single byte character, U+007F, but lower than the highest for two byte characters, so it is a two byte character.

A two byte character is of the format 110xxxxx 10xxxxxx where the most significant bits of the code point value is encoded into the bits marked with an 'x'. As the code point is simply a hexadecimal number, U+00A3 is just 00000000 10100011 in binary, so those 8 bits get encoded onto the 'x' bits, giving 110xxx10 10110011. As we cannot have any spare 'x' bits left over, those that remain are cleared to zero, giving 11000010 10110011 which is, $C2 A3 - and that's the character code for a Pound Sign in UTF8.

Taking the Euro next, it has code point U+20AC which puts it into the three byte set of characters. Those are in the format 1110xxxx 10xxxxxx 10xxxxxx. Once again, we take the code point in binary and mask it onto the 'x'bits, filling with leading zeros as appropriate.

Code point U+20AC is 00100000 10101100 which is 16 bits as a three byte character allows for up to 16 bits, it fits nicely without any spare 'x' bits. The result is 11100010 10000010 10101100 or $E2 82 AC and that's the three bytes we use for the Euro symbol.

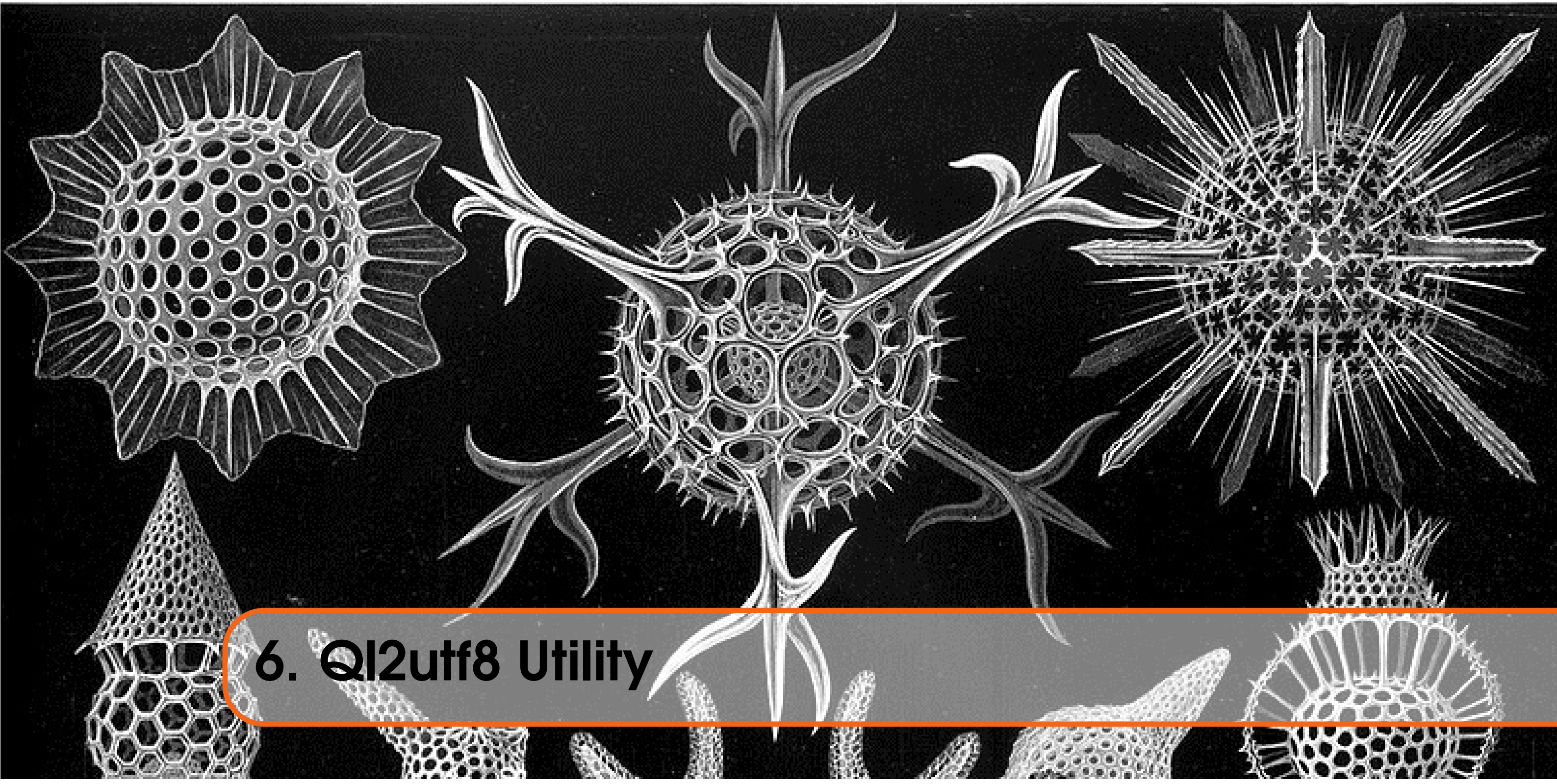
5.2 The QL Character Set

As ever, nothing is straight forward in the QL world. Sir Clive has done his best to unstandardise things. However, I suppose he had only 256 characters to fit ASCII and a few "foreign" characters that might be needed in Europe. America seems to get by on only 7 bits ASCII anyway! So, what's broken in the QL's character set?

- The UK Pound symbol is character 96 ($60) on the QL, but in ASCII it is character 163 ($A3).
- The copyright symbol is character 127 ($7F) on the QL but is 169 ($A9) in ASCII.
- The Euro, which came a long time after the QL, doesn't exist in the BBQL character set, but under SMSQ, it is at character 181 ($B5)
- Characters above 128 ($80) are a mess on the QL. Many are simply missing, especially some of the, I assume, lesser used accented characters.

So while my Linux editor can open files created on the QL, and the QL can open (most) files created on the Linux side of things, it's not completely the same. A conversion is required, one to go from the QL to Linux (MacOS, Windows etc) and one to come back again.

I guess we need some assembly code then? Read on.



6. Ql2utf8 Utility

This utility is what I would need to use when I've saved a file on the QL, or in QPC, and I need to transfer it down to the Linux box for some processing - say, for example, to get the finished and tested source code into an article like this one!

The utility is an example of a QL program which are collectively becoming known as a "YAF".[1]

The utility reads a QL created text file, where the content is any of the QL character set up to but not above, character 191 ($BF) which is the down arrow. Anything above that is a control character and is unprintable - undefined results may occur if any are present in the QL file.

It is executed in the usual manner:

```
ex ram1_ql2utf8_bin , ram1_ql_txt , ram1_utf8_txt
```

The input file, ram1_ql_txt will be read in, and each byte converted to the appropriate UTF8 byte sequence, and written out to the ram1_utf8_txt file. The latter file will be used on my Linux box, but Windows and MacOS users can also take advantage.

Right, enough waffle, on with the code.

6.1 The Code

As ever, my code starts with an introductory header and some equates. This utility is no different as you can see below.

```
;——————————————————————————————————————————————————————————————————————
; QL2UTF8:
;
; This filter converts QL text files to UTF8 for use on Linux, Mac or
```

[1] Yet Another Filter!

```
; Windows where most modern editors etc, default to UTF8.
;
;
; EX ql2utf8_bin, input_file, output_file_or_channel
;
;------------------------------------------------------------------------
; 26/09/2019 NDunbar Created for QDOSMSQ Assembly Mailing List
;------------------------------------------------------------------------
; (c) Norman Dunbar, 2019. Permission granted for unlimited use
; or abuse, without attribution being required. Just enjoy!
;------------------------------------------------------------------------

; How many channels do I want?
numchans    equ     2       ; How many channels required?


; Stack stuff.
sourceId    equ     $02     ; Offset(A7) to input file id
destId      equ     $06     ; Offset(A7) to output file id

; Other Variables
pound       equ     96      ; UK Pound sign.
copyright   equ     127     ; (c) sign.
grave       equ     159     ; Backtick/Grave accent.
euro        equ     181     ; Euro symbol
err_bp      equ     -15
err_eof     equ     -10
me          equ     -1
timeout     equ     -1
```

The main entry point for the program is next. This section of code contains the usual QDOS Job header and a few checks to ensure that we only get a pair of channel IDs on the stack. If the user decided to pass over a command string as well, it would be ignored.

```
;========================================================================
; Here begins the code.
;------------------------------------------------------------------------
; Stack on entry:
;
; $06(a7) = Output file channel id.
; $02(a7) = Source file channel id.
; $00(a7) = How many channels? Should be $02.
;========================================================================
start       bra.s   checkStack

            dc.l    $00
            dc.w    $4afb
name        dc.w    name_end-name-2
            dc.b    'QL2UTF8'
name_end    equ     *

version     dc.w    vers_end-version-2
            dc.b    'Version 1.00'
vers_end    equ     *

```

```

bad_parameter
    moveq   #err_bp,d0       ; Guess!
    bra     errorExit        ; Die horribly


;------------------------------------------------------------------------
; Check the stack on entry. We only require NUMCHAN channels - any
; thing other than NUMCHANS will result in a BAD PARAMETER error on
; exit from EW (but not from EX).
;------------------------------------------------------------------------
checkStack
    cmpi.w  #numchans,(a7)   ; Two channels is a must
    bne.s   bad_parameter    ; Oops
```

Next up is some initialisation. In this short section of code, a couple of registers are set to values which will be used throughout the entire utility.

```
;------------------------------------------------------------------------
; Initialise a couple of registers that will keep their values all
; through the rest of the code.
;------------------------------------------------------------------------
q12utf8
    lea     utf8,a2          ; Preserved throughout
    moveq   #timeout,d3      ; Timeout, also Preserved
```

And now we have the top of the main loop for the program. We start here by initialising the various registers to be able to read a single byte from the input channel. The ID for that file is on the stack at offset 2 from the current value in register A7.

Once a byte has been read we check the error code in D0, and if it shows no errors, we can get on with the translation. If D0 is showing an error, and it happens to be End Of File, we bale out of the program and return success to SuperBASIC, Other errors will return the appropriate error code to SuperBASIC, but that will only be seen if the utility was executed with EXEC_W or EW, or equivalent.

```
;------------------------------------------------------------------------
; The main loop starts here. Read a single byte, check for EOF etc.
;------------------------------------------------------------------------
readLoop
    moveq   #io_fbyte,d0     ; Fetch one byte
    move.l  sourceID(a7),a0  ; Channel to readLoop
    trap    #3               ; Do input
    tst.l   d0               ; OK?
    beq.s   testBit7         ; Yes
    cmpi.l  #ERR_EOF,d0      ; All done?
    beq     allDone          ; Yes.
    bra     errorExit        ; Oops!
```

The first check is to test it bit 7 of the character just read, is set or not. It it is set then the chances are that it is a multi-byte character. If it is clear, then we continue processing.

```
testBit7
     btst     #7,d1            ; Bit 7 set?
     bne.s    twoBytes         ; Multi Byte character if so
```

Right then, at this point the top bit must be clear, so we are looking at a single byte character, or are we? The QL has a few little exceptions to the rule as it uses different character codes to standard (if there is such a thing) ASCII.

The first exception is the UK Pound sign, which is a two byte character in UTF8. The code below checks and processes a Pound sign, if one is found. After writing out the UTF8 codes, it loops back to the start of the main loop, ready for the next character.

```
;------------------------------------------------------------------
; The UK Pound and copyright signs are exceptions to the "bytes
; less than $80 are the same in UTF8 as they are in ASCII" rule as
; Sir Clive didn't follow ASCII 100%. Both characters are multi-byte
; in UTF8.
;------------------------------------------------------------------
testPound
     btst     #7,d1            ; Potential multi-byte character?
     bne.s    twoBytes         ; Yes
     cmpi.b   #pound,d1        ; Got a UK Pound sign?
     bne.s    testCopyright    ; No.

gotPound
     move.b   #$c2,d1          ; Pound is $C2A3 in UTF8.
     bsr.s    writeByte        ; Write first byte
     move.b   #$a3,d1
     bsr.s    writeByte        ; Write second byte
     bra.s    readLoop
```

The next exception is the copyright symbol. It too is a multi byte character in UTF8 so the code below checks for it and deals with it appropriately.

```
;------------------------------------------------------------------
; Here we repeat the same check as above, in case we have the
; copyright sign.
;------------------------------------------------------------------
testCopyright
     cmpi.b   #copyright,d1    ; Got a copyright sign?
     bne.s    oneByte          ; No.

gotCopyright
     move.b   #$c2,d1          ; Copyright is $C2A9 in UTF8.
     bsr.s    writeByte        ; Write first byte
     move.b   #$a9,d1
     bsr.s    writeByte        ; Write second byte
     bra.s    readLoop
```

That's all the QL characters that are exceptions to the “ASCII characters below code 128 are single byte in UTF8” rule. The remaining QL characters less than code 128 are dealt with by simply calling the routine to write a single byte and then heading back to the top of the main loop. Job done.

```
;------------------------------------------------------------------------------
; All other ASCII characters, below $80, are single byte in UTF8 and
; are the same code as in ASCII.
;------------------------------------------------------------------------------
oneByte
     bsr.s    writeByte        ; Single byte required in UTF8
     bra.s    readLoop
```

Speaking of writing a single byte, the following code does exactly that. It fetches the channel ID for the output channel from the stack. Normally, this would be at offset “destId” on from A7, but as this code is always called as a subroutine, there is an extra 4 bytes on the stack for the calling code’s return address, so that has to be considered.

All the following snippet has to do is set up the registers to enable the trap call, IO_SBYTE, to be called. D3, the timeout, is already set to -1, and will be preserved on return, as will D2, which is being used elsewhere in the code to safely hold a value during processing.

```
;------------------------------------------------------------------------------
; A small but perfectly formed subroutine to send the byte in D1 to
; the output channel.
; BEWARE: This is called with an extra 4 bytes on the stack!
;------------------------------------------------------------------------------
writeByte
     moveq    #io_sbyte,d0      ; Send one byte
     move.l   4+destId(a7),a0   ; Output channel id
     trap     #3
     tst.l    d0                ; OK?
     bne.s    errorExit         ; Oops!
     rts
```

As mentioned above, we have processed all the QL characters that are a single byte in UTF8, so now we need to think about those characters with codes above 127, the majority of these are accented characters and as the QL doesn’t cover all the “standard” ones, there is some “furkling about”[2] to be done.

The QL wouldn’t be the QL we know and love if there were not a couple of exceptions to the rule that “ASCII characters above code 128 are always multi-byte”. The grave (no, not somewhere you bury people, the accent much loved by the French I believe) aka the backtick (at least on Unix, Linux etc) is actually a single byte character in UTF8, so that is dealt with first.

We arrive at the following code whenever a character is read in which has the top bit, bit 7, set.

The code begins by checking for and processing a grave character.

```
;------------------------------------------------------------------------------
; ASCII codes from $80 upwards require multiple bytes in UTF8. In the
; case of the QL, these are mostly 2 bytes long. I could use IO_SSTRG
; here, I know.
; However, as ever, there are exceptions. The grave accent (backtick)
```

[2]That would be a technical term!

```
; is a single byte on output, while the 4 arrow keys are three bytes.
; The bytes to be sent are read from a table because, again, the QL
; is not using the full set of accented characters - so there is
; mucking about to be done.
;------------------------------------------------------------------------
twoBytes
    cmpi.b   #grave,d1        ; Backtick/Grave accent?
    bne.s    testEuro         ; No.

;------------------------------------------------------------------------
; We are dealing with a backtick character (aka Grave accent)?
;------------------------------------------------------------------------
gotGrave
    move.b   #pound,d1        ; Grave in = pound out!
    bsr.s    writeByte        ; Single byte required
    bra      readLoop         ; Do the rest
```

From here on in we should be dealing with all the two byte characters for UTF8, however, those exceptions are popping up again. The first is the Euro symbol. This is missing from the original 128Kb QLs of old, as the Euro didn't even exist when they were conceived, however, in SMSQ, they have been allocated character 181 - which, when you look at it in Pennel or similar, is a seriously weird character which I've never seen used, si I think the SMSQ authors chose well!

In UTF8 the Euro needs three characters, $E282AC, so the following section of code does the necessary checking and handling of a Euro character.

```
;------------------------------------------------------------------------
; Here we repeat the same check as above, in case we have the
; Euro sign.
;------------------------------------------------------------------------
testEuro
    cmpi.b   #euro,d1         ; Got a Euro sign?
    bne.s    testArrows       ; No.

gotEuro
    move.b   #$e2,d1          ; Euro is $E282AC in UTF8.
    bsr.s    writeByte        ; Write first byte
    move.b   #$82,d1
    bsr.s    writeByte        ; Write second byte
    move.b   #$ac,d1
    bsr.s    writeByte        ; Write third byte
    bra.s    readLoop
```

Finally, in our exception handling code, the 4 arrow keys. These too are three bytes long in UTF8, $E2869x, where the 'x' nibble is 0, 1, 2 or 3 depending on the arrow's direction. Just to be awkward, the QL's arrow order is different to UTF8 - on the QL the ascending character codes are for the Left, Right, Up, Down arrows, but in UTF8 they are ordered Left, Up, Right, Down.

The code snippet below handles the arrow keys.

```
;------------------------------------------------------------------------
; The arrows are $BC, $BD, $BE and $BF (left, right, up, down). These
; are three bytes in UTF8, $E2 $86 $9x where 'x' is 0, 2, 1 or 3.
```

```
;------------------------------------------------------------------
testArrows
    move.b  d1,d2           ; Copy character code
    subi.b  #$bc,d2         ; Anything lower = C set
    bcs.s   notArrows       ; And is not an arrow
    subi.b  #4,d2           ; Arrows = 0-3. C clear is bad
    bcc.s   notArrows       ; Still not an arrow.

gotArrows
    subi.b  #$bc,d1         ; D1 = 0 to 3
    lea     arrows,a3       ; Arrow table
    move.b  d1,d2           ; Save index into table
    ext.w   d2              ; Need word not byte

    move.b  #$e2,d1         ; First byte
    bsr.s   writeByte
    move.b  #$86,d1         ; Second byte
    bsr.s   writeByte
    move.b  0(a3,d2.w),d1   ; Third byte
    bsr.s   writeByte
    bra     readLoop        ; Go around again.
```

The arrow key's third byte is located in the following tiny table which has the correct third byte for the appropriate arrow's code on the QL.

```
;------------------------------------------------------------------
; We need this as arrows in the QL are Left, Right, Up, Down but in
; UTF8 they are Left, Up, Right, Down. Sigh.
;------------------------------------------------------------------
arrows
    dc.b    $90,$92,$91,$93   ; Awkward byte order!
```

That is now, all the two byte exceptions catered for. The remainder of the higher ASCII characters are all two bytes in size. Obviously, being the QL, these are not in the same order as the originating ASCII codes would be, had Sir Clive done the decent thing and used a standard ASCII code page! Instead he chose to omit some characters and rearrange the others into a non-standard order.[3]

The following code simply copies the character code from D1 to D2 and then manipulates D2 to go from an index into the table, to an offset into the table where a pair of bytes can be found that represent the UTF8 code for the current character.

As we are dealing with character codes from 128 ($80) onwards, we start by subtracting $80 from the character code. This gives the correct index into the table. As each entry in the table is two bytes, we double the index to get the correct offset, then pick up the two bytes there and send them on their way to the output file, before heading back to the start of the main loop.

```
;------------------------------------------------------------------
; Now we are certain, everything is two bytes. Read them from the
; table and write them out.
;------------------------------------------------------------------
notArrows
```

[3]Ok, fair play, there probably wasn't a standard ASCII code page he could use back then.

```
    move.b  d1,d2           ; D2 = byte just read
    subi.b  #$80,d2         ; Adjust for table index
    ext.w   d2              ; Word size needed
    lsl.w   #1,d2           ; Double D2 for Offset
    move.b  0(a2,d2.w),d1   ; First byte
    bsr.s   writeByte       ; Send it output
    addq.b  #1,d2
    move.b  0(a2,d2.w),d1   ; Second byte
    bsr.s   writeByte       ; Send it out too
    bra     readLoop        ; Go around.
```

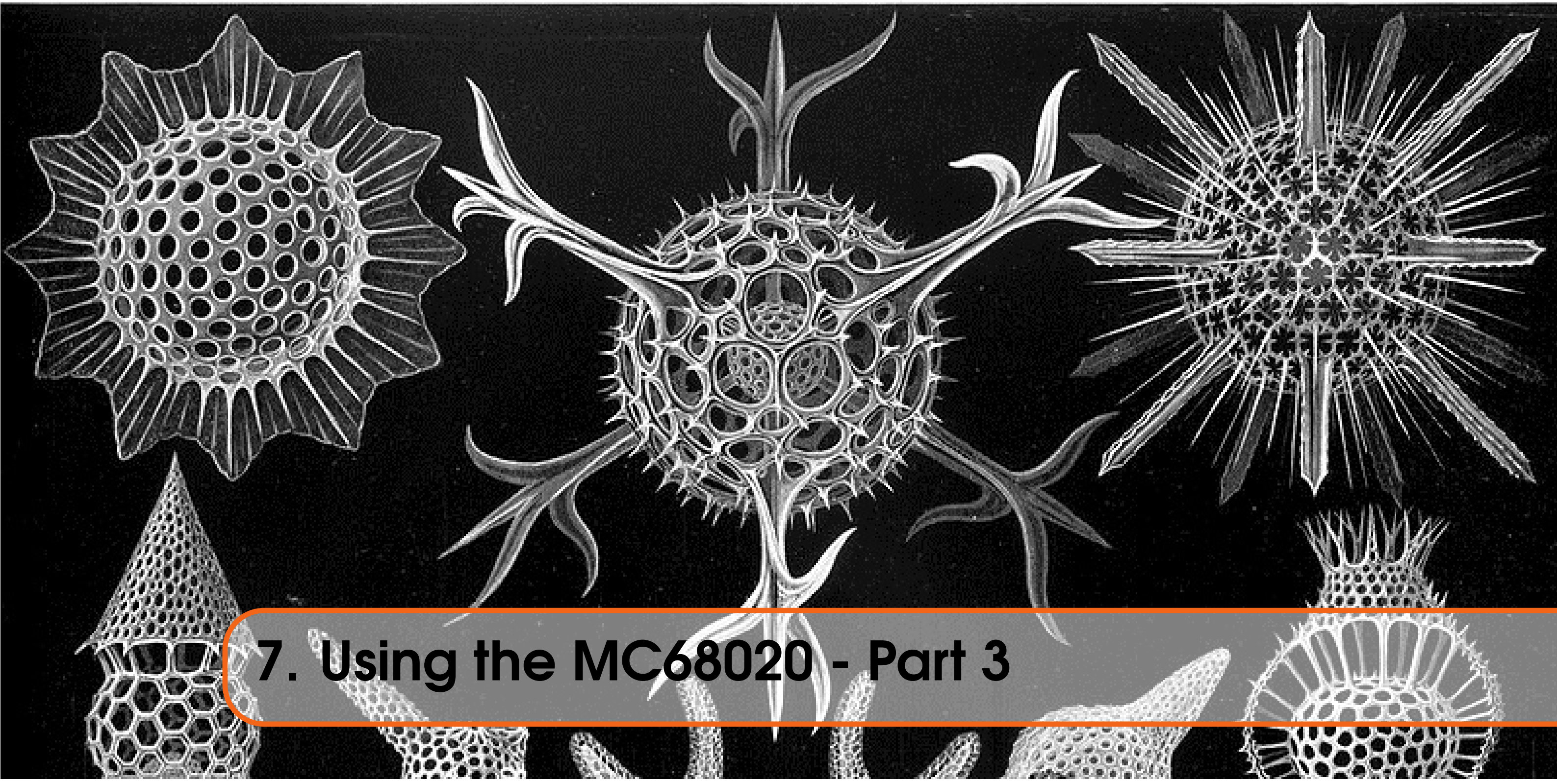
The code below is the usual tidy up and bale out code. It doesn't require much explanation as you will have seen it before, many times.

```
;------------------------------------------------------------------------------
; No errors, exit quietly back to SuperBASIC.
;------------------------------------------------------------------------------
allDone
    moveq   #0,d0

;------------------------------------------------------------------------------
; We have hit an error so we copy the code to D3 then exit via a
; forcible removal of this job. EXEC_W/EW will display the error in
; SuperBASIC, but EXEC/EX will not.
;------------------------------------------------------------------------------
errorExit
    move.l  d0,d3           ; Error code we want to return

;------------------------------------------------------------------------------
; Kill myself when an error was detected, or at EOF.
;------------------------------------------------------------------------------
suicide
    moveq   #mt_frjob,d0    ; This job will die soon
    moveq   #me,d1
    trap    #1
```

Finally, the table of two byte values for the multi-byte characters. Those which have a word of $0000 are exceptions, dealt with elsewhere. And finally, the table only goes as far as character 191 ($BF) as everything that follows is unprintable and unlikely to ever get into a QL text file. This basically means that if you do manage to do this, the output will be “undefined” - as they say!

```
;------------------------------------------------------------------------------
; The following table contains the two byte sequences required for
; QL characters above $80. These are all 2 bytes in UTF8, so quite a
; simple case. (Not when converting UTF8 to QL though!)
;------------------------------------------------------------------------------
utf8
    dc.w    $c3a4           ; a umlaut
    dc.w    $c3a3           ; a tilde
    dc.w    $c3a2           ; a circumflex
    dc.w    $c3a9           ; e acute
    dc.w    $c3b6           ; o umlaut
    dc.w    $c3b5           ; o tilde
```

```
        dc.w    $c3b8           ; o slash
        dc.w    $c3bc           ; u umlaut
        dc.w    $c3a7           ; c cedilla
        dc.w    $c3b1           ; n tilde
        dc.w    $c3a6           ; ae ligature
        dc.w    $c593           ; oe ligature
        dc.w    $c3a1           ; a acute
        dc.w    $c3a0           ; a grave
        dc.w    $c3a2           ; a circumflex
        dc.w    $c3ab           ; e umlaut
        dc.w    $c3a8           ; e grave
        dc.w    $c3aa           ; e circumflex
        dc.w    $c3af           ; i umlaut
        dc.w    $c3ad           ; i acute
        dc.w    $c3ac           ; i grave
        dc.w    $c3ae           ; i circumflex
        dc.w    $c3b3           ; o acute
        dc.w    $c3b2           ; o grave
        dc.w    $c3b4           ; o circumflex
        dc.w    $c3ba           ; u acute
        dc.w    $c3b9           ; u grave
        dc.w    $c3bb           ; u circumflex
        dc.w    $ceb2           ; B as in ss (German)
        dc.w    $c2a2           ; Cent
        dc.w    $c2a5           ; Yen
        dc.w    $0000           ; Grave accent - single byte
        dc.w    $c384           ; A umlaut
        dc.w    $c383           ; A tilde
        dc.w    $c385           ; A circle
        dc.w    $c389           ; E acute
        dc.w    $c396           ; O umlaut
        dc.w    $c395           ; O tilde
        dc.w    $c398           ; O slash
        dc.w    $c39c           ; U umlaut
        dc.w    $c387           ; C cedilla
        dc.w    $c391           ; N tilde
        dc.w    $c386           ; AE ligature
        dc.w    $c592           ; OE ligature
        dc.w    $ceb1           ; alpha
        dc.w    $ceb4           ; delta
        dc.w    $ceb8           ; theta
        dc.w    $cebb           ; lambda
        dc.w    $c2b5           ; micro (mu?)
        dc.w    $cf80           ; PI
        dc.w    $cf95           ; o pipe
        dc.w    $c2a1           ; ! upside down
        dc.w    $c2bf           ; ? upside down
        dc.w    $0000           ; Euro
        dc.w    $c2a7           ; Section mark
        dc.w    $c2a4           ; Currency symbol
        dc.w    $c2ab           ; <<
        dc.w    $c2bb           ; >>
        dc.w    $c2ba           ; Degree
        dc.w    $c3b7           ; Divide
```

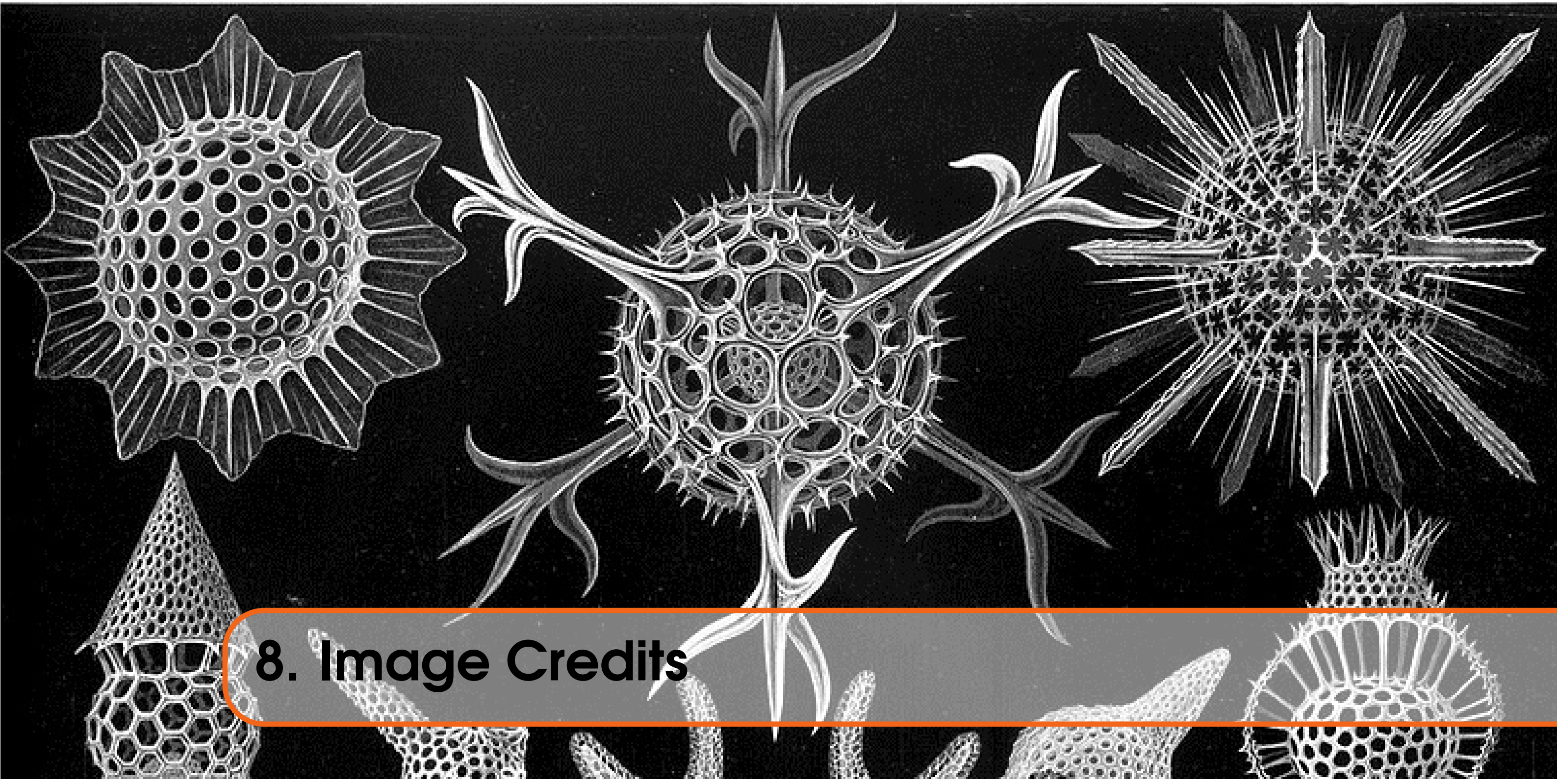


7. Using the MC68020 - Part 3

This article continues our look at the last of the new features of the MC68020, the exception handling.

7.1 MC68020 Exception Handling

7.1.1 Address Exception



8. Image Credits

The front cover image on this ePeriodical is taken from the book *Kunstformen der Natur* by German biologist Ernst Haeckel. The book was published between 1899 and 1904. The image used is of various *Polycystines* which are a specific kind of micro-fossil.

I have also cropped the image for use on each chapter heading page.

You can read about Polycystines on Wikipedia and there is a brief overview of the above book, also on Wikipedia, which shows a number of other images taken from the book. (Some of which I considered before choosing the current one!)

Polycystines have absolutely nothing to do with the QL or computing in general - in fact, I suspect they died out before electricity was invented - but I liked the image, and decided that it would make a good cover for the book and a decent enough chapter heading image too.

Not that I am suggesting, *in any way whatsoever*, that we QL fans are ancient.