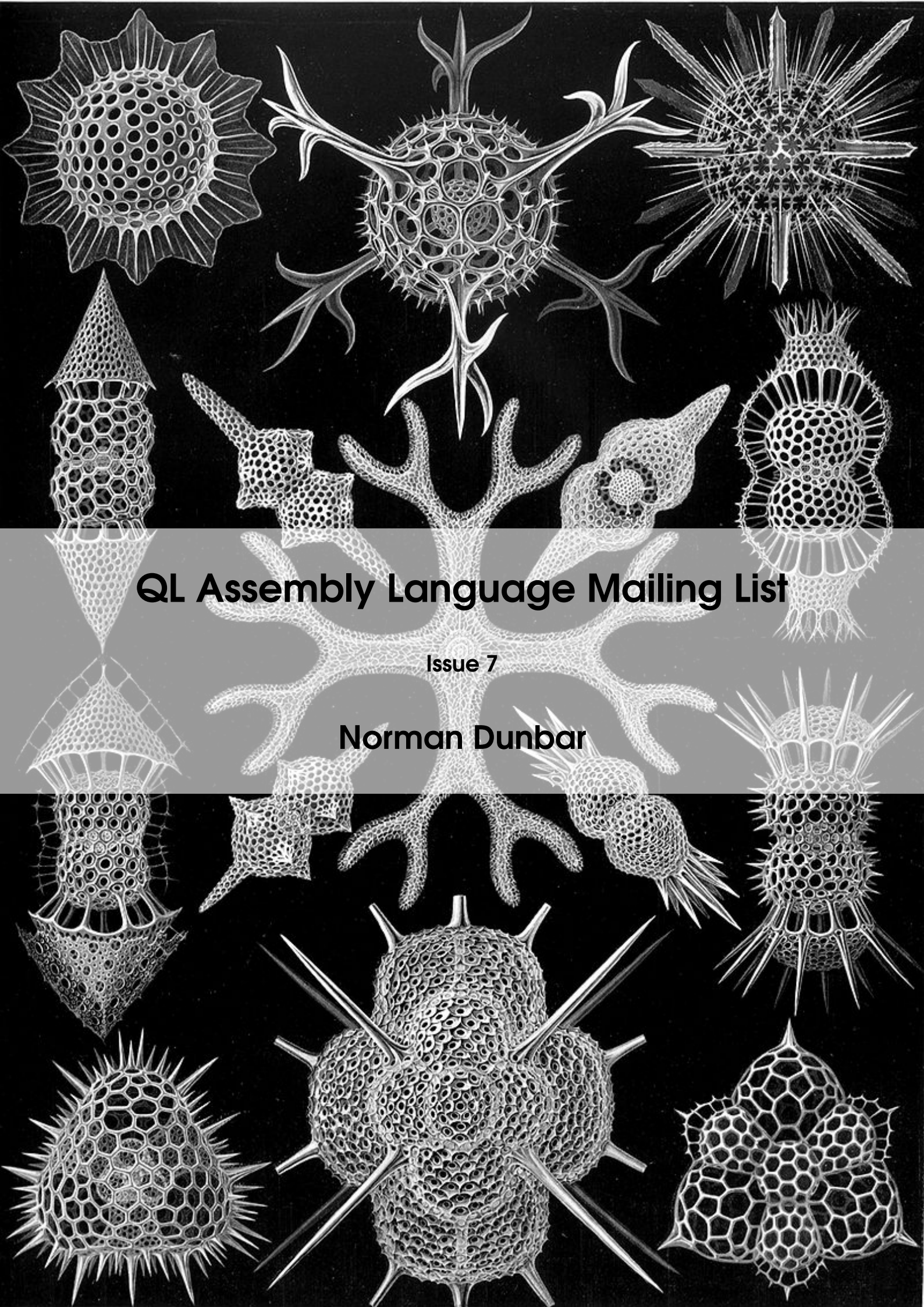


QL Assembly Language Mailing List

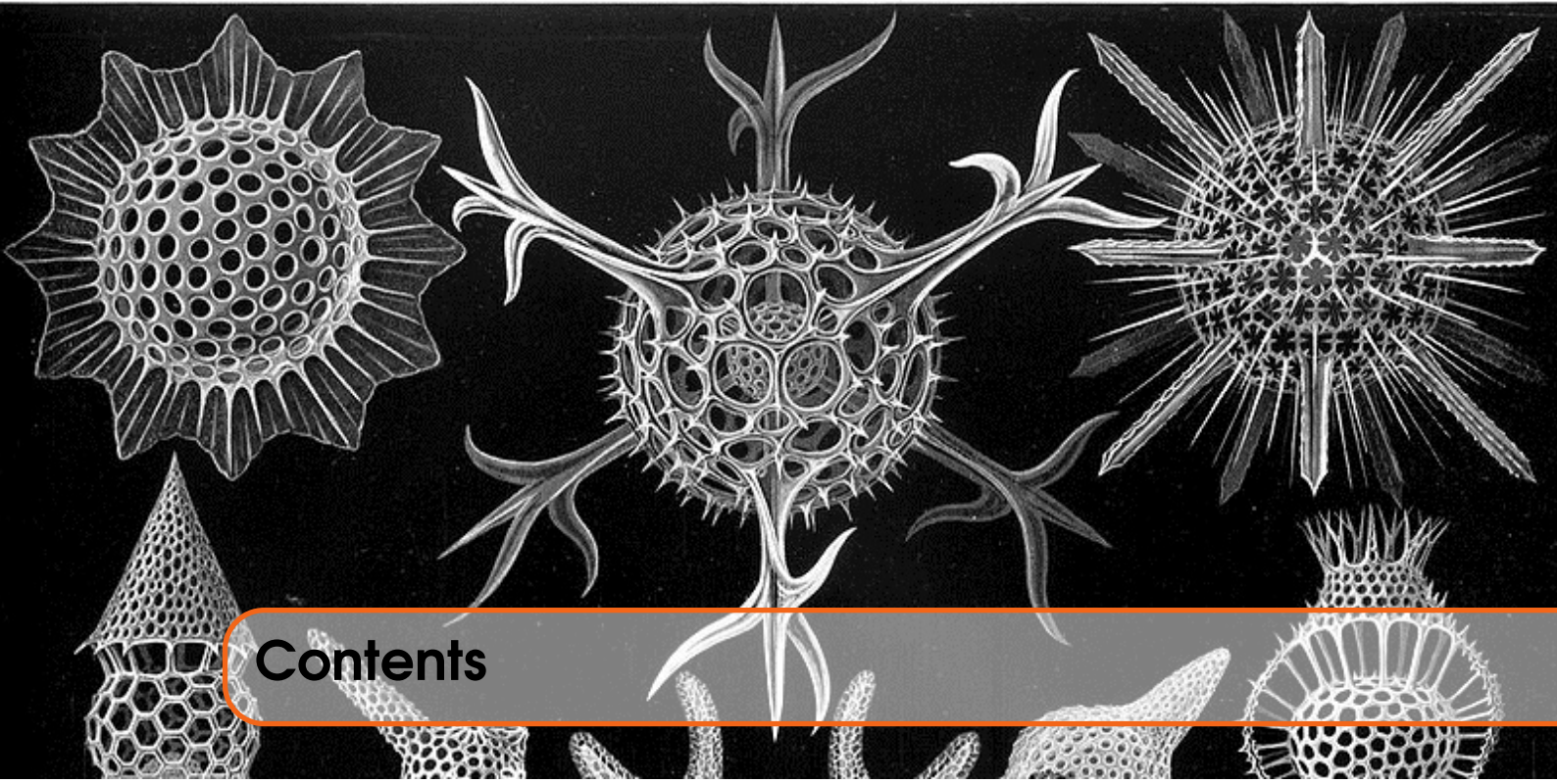
Issue 7

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 *8/1/2019* at *11:14:33*.



Contents

1 Preface 7

1.1 Feedback 7

1.2 Subscribing to The Mailing List 7

1.3 Contacting The Mailing List 8

2 Feedback on Issue 6 9

2.1 No Feedback so far! 9

3 The Fastest Scrolling in the West 11

3.1 The Straight-Forward Approach 12

3.2 Unrolling loops (or: How to waste precious amounts of memory) 12

3.3 MOVEM.L can work in other places than the Stack 14

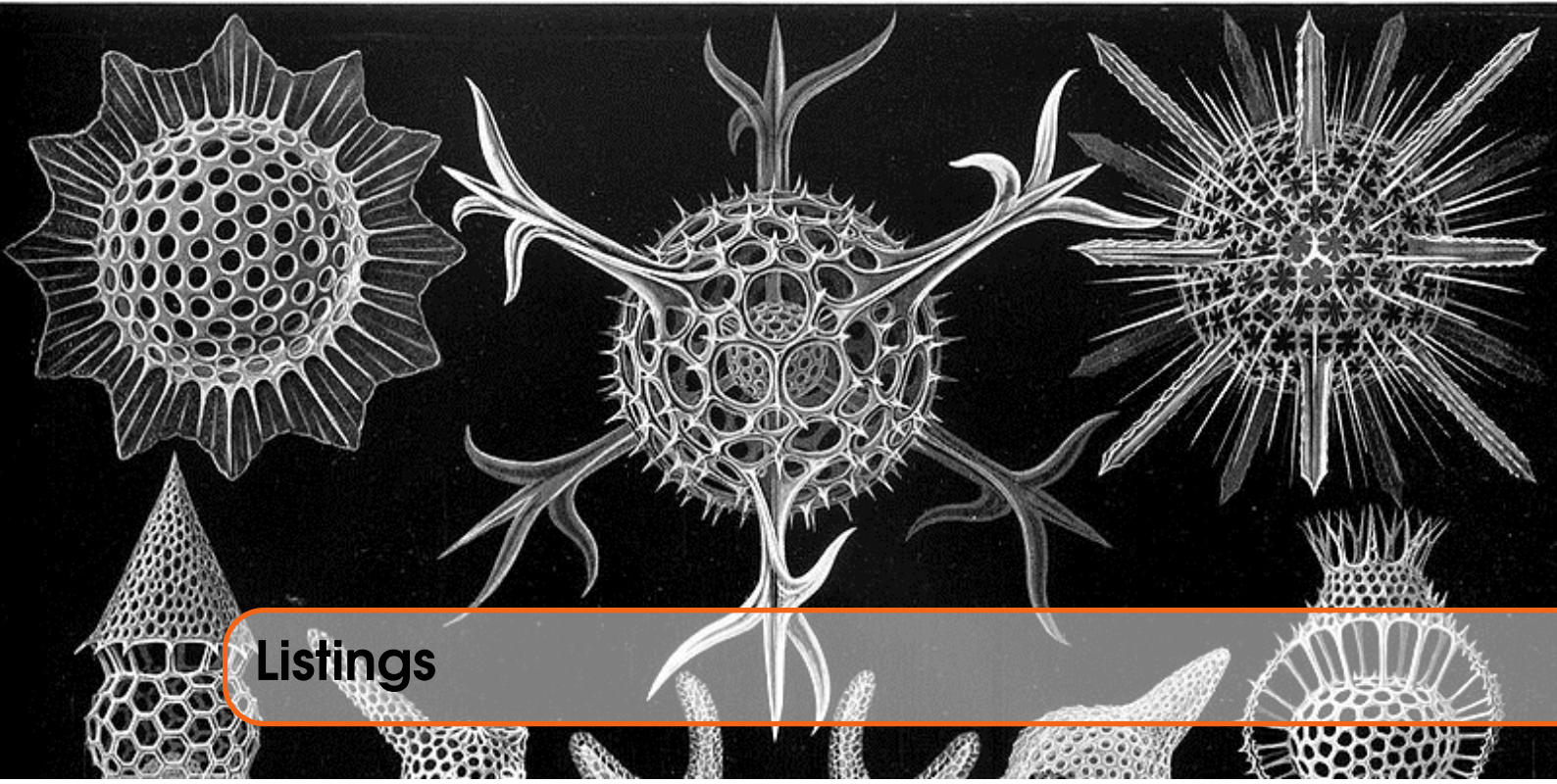
3.4 If Software can't cope, use Hardware 14

4 Using the MC68020 - Part 3 17

4.1 MC68020 Exception Handling 17

4.1.1 Address Exception 17

5 Image Credits 19



Listings

3.1 Scrolling one pixel leftwards 12
3.2 The REPT Macro 12
3.3 A simple REPT example 13
3.4 Unrolling the innner loop 13
3.5 Unrolling the outer loop 13
3.6 MOVEM restrictions 14
3.7 Improving the REPT macro 14
3.8 Scrolling one pixel leftwards 14

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

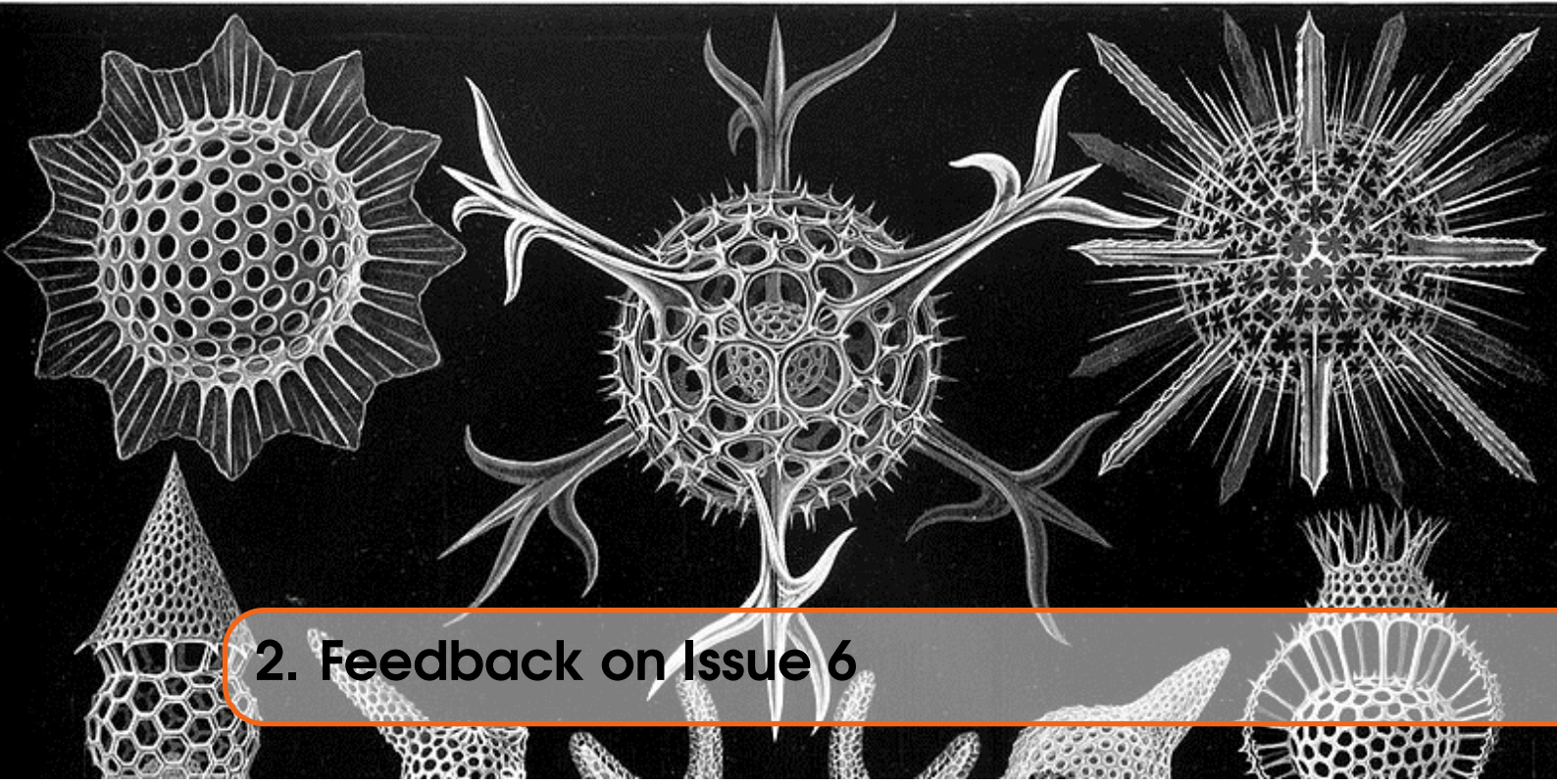
I'm rather hoping that this mailing list will not be a one-way affair, like QL Today appeared to be. I'm very open to suggestions, opinions, articles etc from my readers, otherwise how do I know what I'm doing is right or wrong?

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

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.

3.1 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:

```
1  ; Scrolls the screen one pixel to the left
2  Lscroll
3      movem.l a0-a1,-(sp)
4      lea     screen_start,a0
5      lea     2(a0),a1
6      move.w  #384-1,d1          ; 384 scan-lines
7
8  lineLoop
9      move.w  #512-1,d0          ; 512 pixels to move
10
11 cpy_loop
12     move.w  (a1)+,(a0)+
13     dbf     d0,cpy_loop
14     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:

3.2 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.

```
1  REPT    MACRO num,args
2          LOCAL count,pIndex,pCount
3  Count   SETNUM 1
4  Lp      MACLAB
5  pCount  SETNUM [.NPARAMS] - 1
6  pIndex  SETNUM 2
7  pLoop:  MACLAB
8          EXPAND
9          [.PARAM([pIndex])]
10         NOEXPAND
11 pCount  SETNUM [pCount]-1
12 pIndex  SETNUM [pIndex]+1
13         IFNUM  [pCount] >= 0 GOTO pLoop
14 Count   SETNUM [count] + 1
```

```
15          IFNUM   [count] <= [num] GOTO lp
16          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

```
1 NotUseful
2     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.

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:

```
1  ; Scrolls the screen one pixel to the left
2  Lscroll
3      movem.l a0-a1,-(sp)
4      lea     screen_start,a0
5      lea     2(a0),a1
6      move.w  #384-1,d1           ; 384 scan-lines
7
8  lineLoop
9      REPT    256,{     move.l (a1)+,(a0)+}
10     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

```
1 screenLongs EQU 512*384*2/4
2     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.

3.3 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

```
1    movem.l  (a0)+,REGSET
2    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)

```
1    REPT      25,{    movem.l  (a1)+,d3-d7/a2-a7},\
2                 {    movem.l  d3-d7,a2-a6,(a0)},\
3                 {    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

```
1    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:

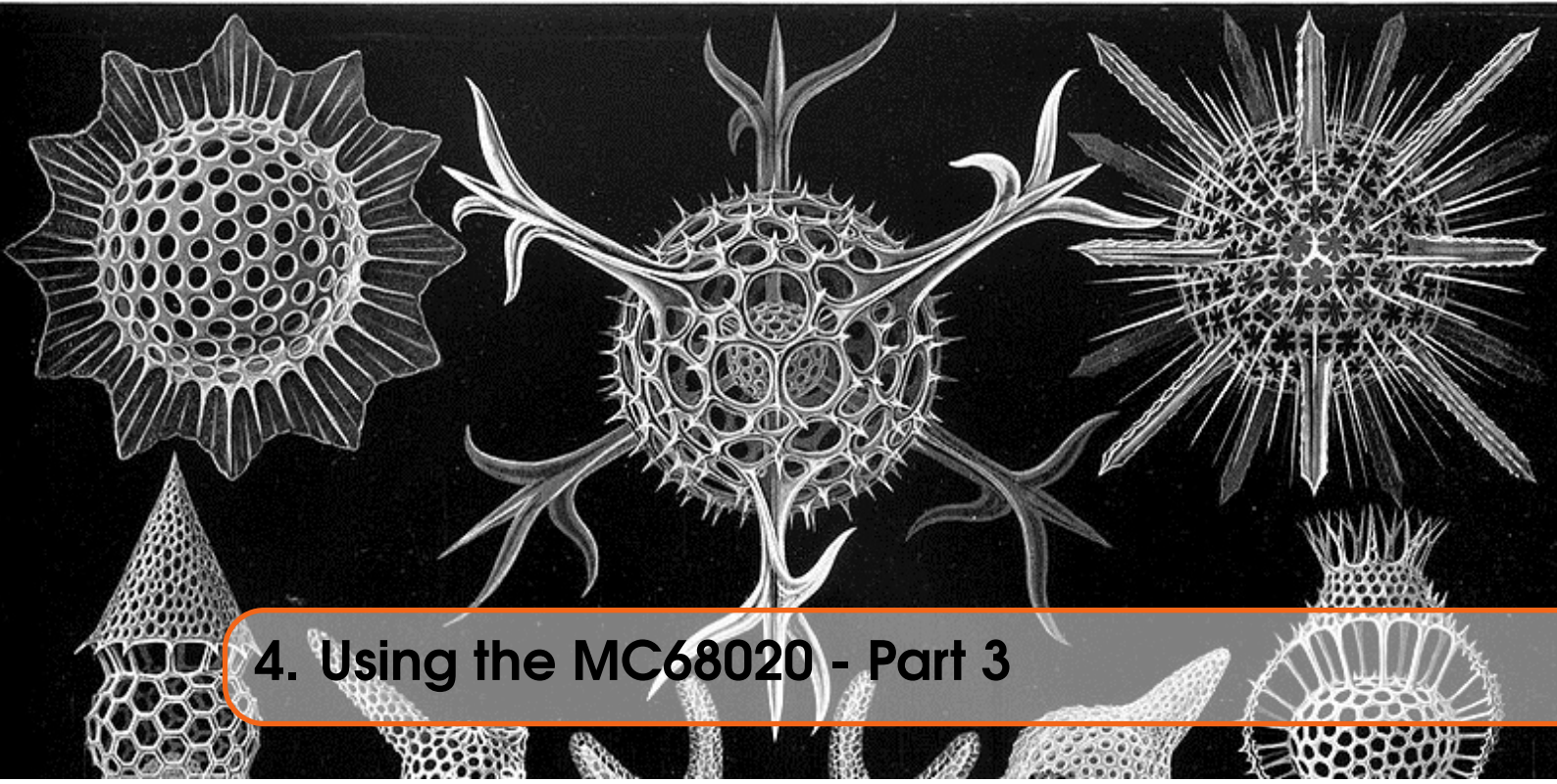
3.4 If Software Can't Cope, Use Hardware

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.

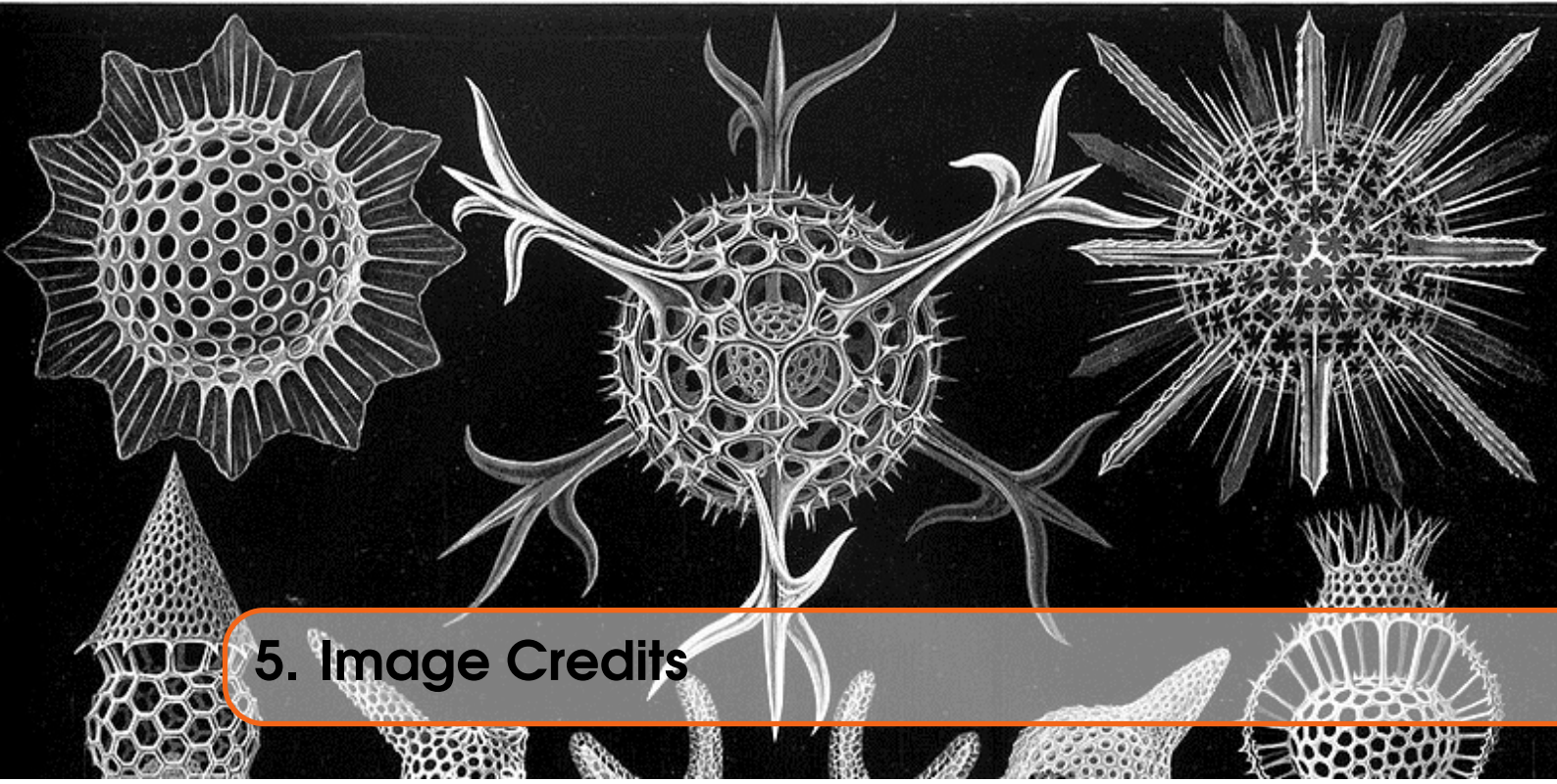


4. Using the MC68020 - Part 3

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

4.1 MC68020 Exception Handling

4.1.1 Address Exception



5. 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.