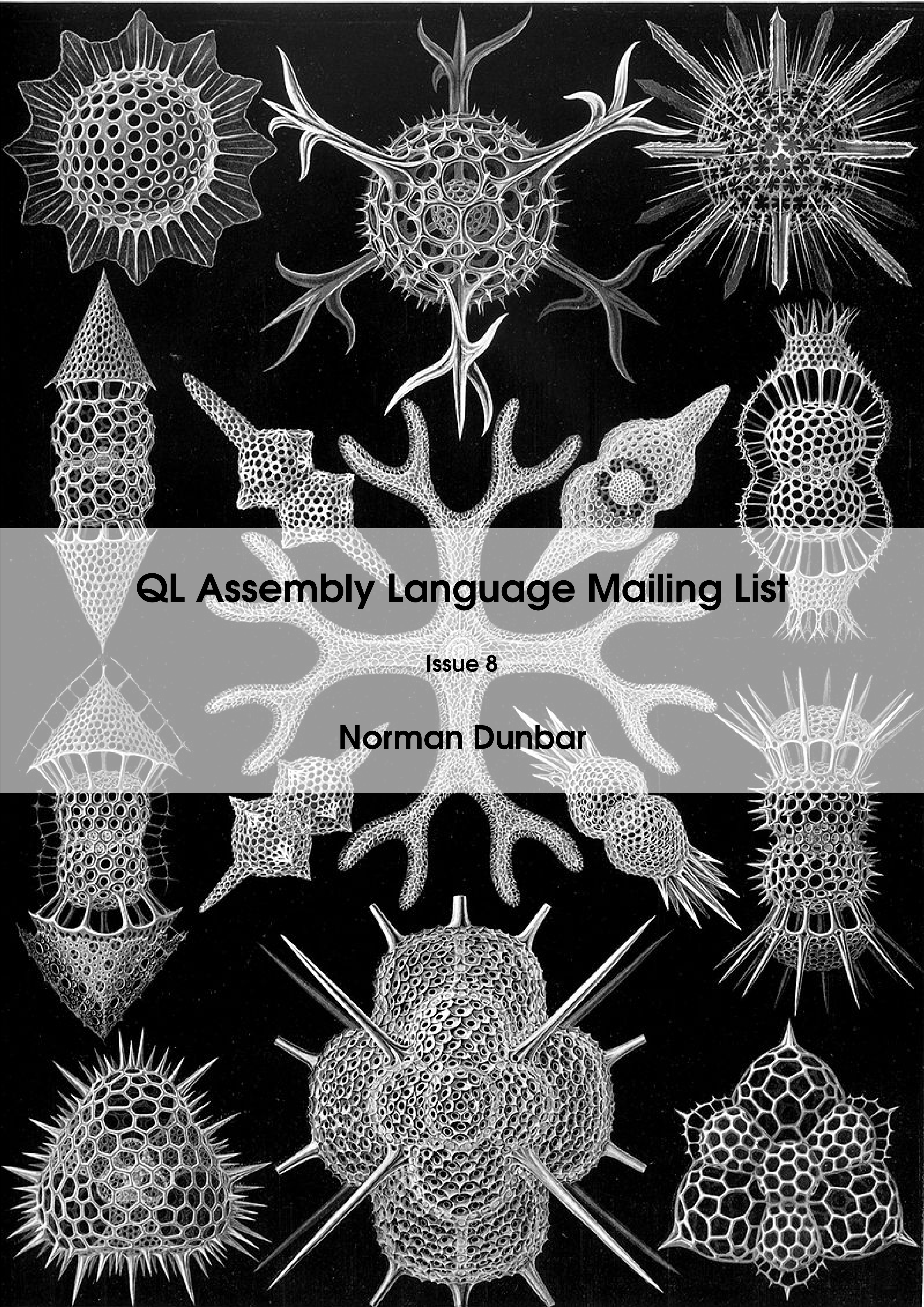


QL Assembly Language Mailing List

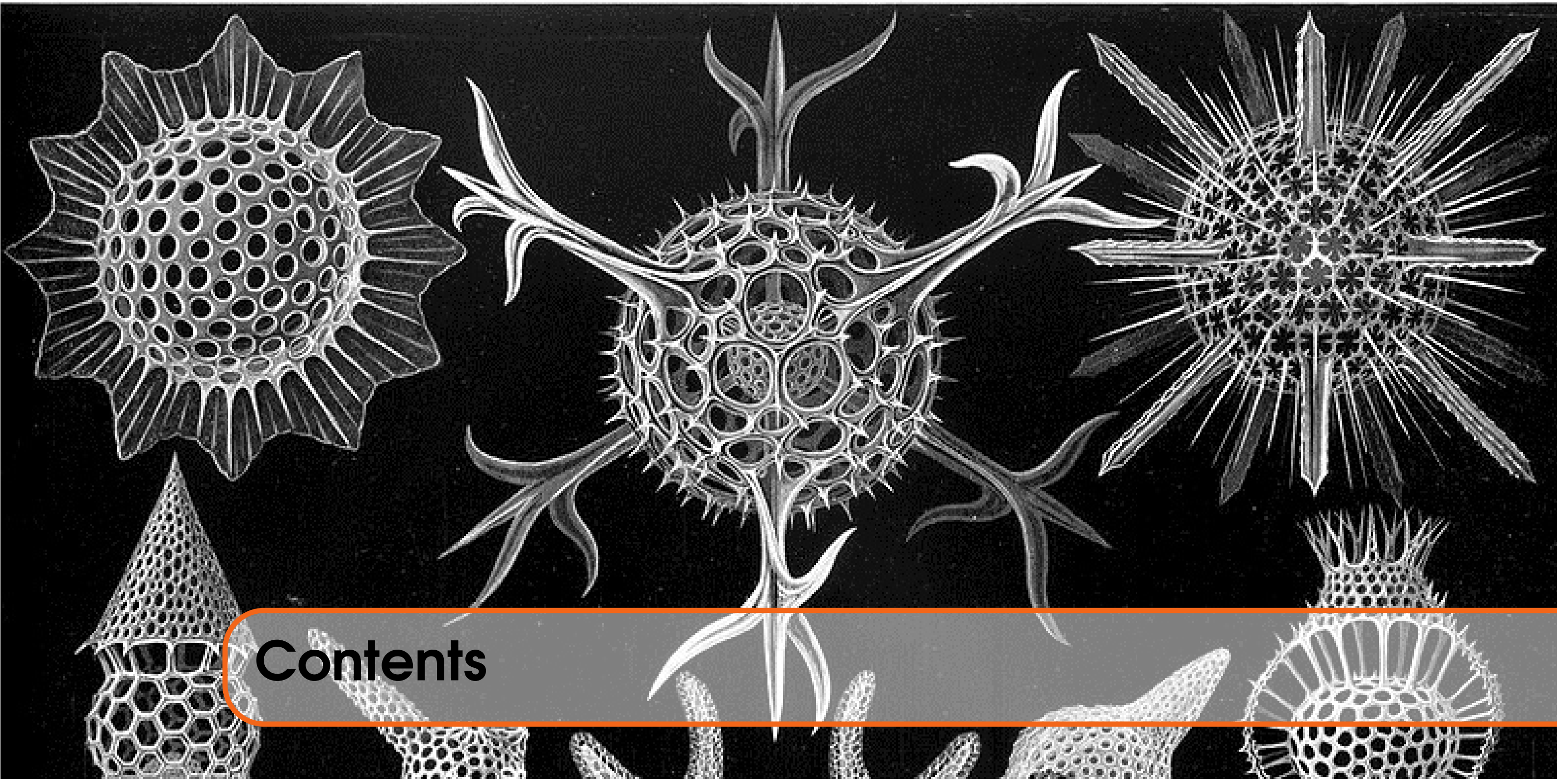
Issue 8

Norman Dunbar

PUBLISHED BY MEMYSELFEYE PUBLISHING ;-)

Download from:
https://github.com/NormanDunbar/QLAssemblyLanguageMagazine/releases/tag/Issue_8

This pdf document was created on *11/12/2020* at *16:46:25*.



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

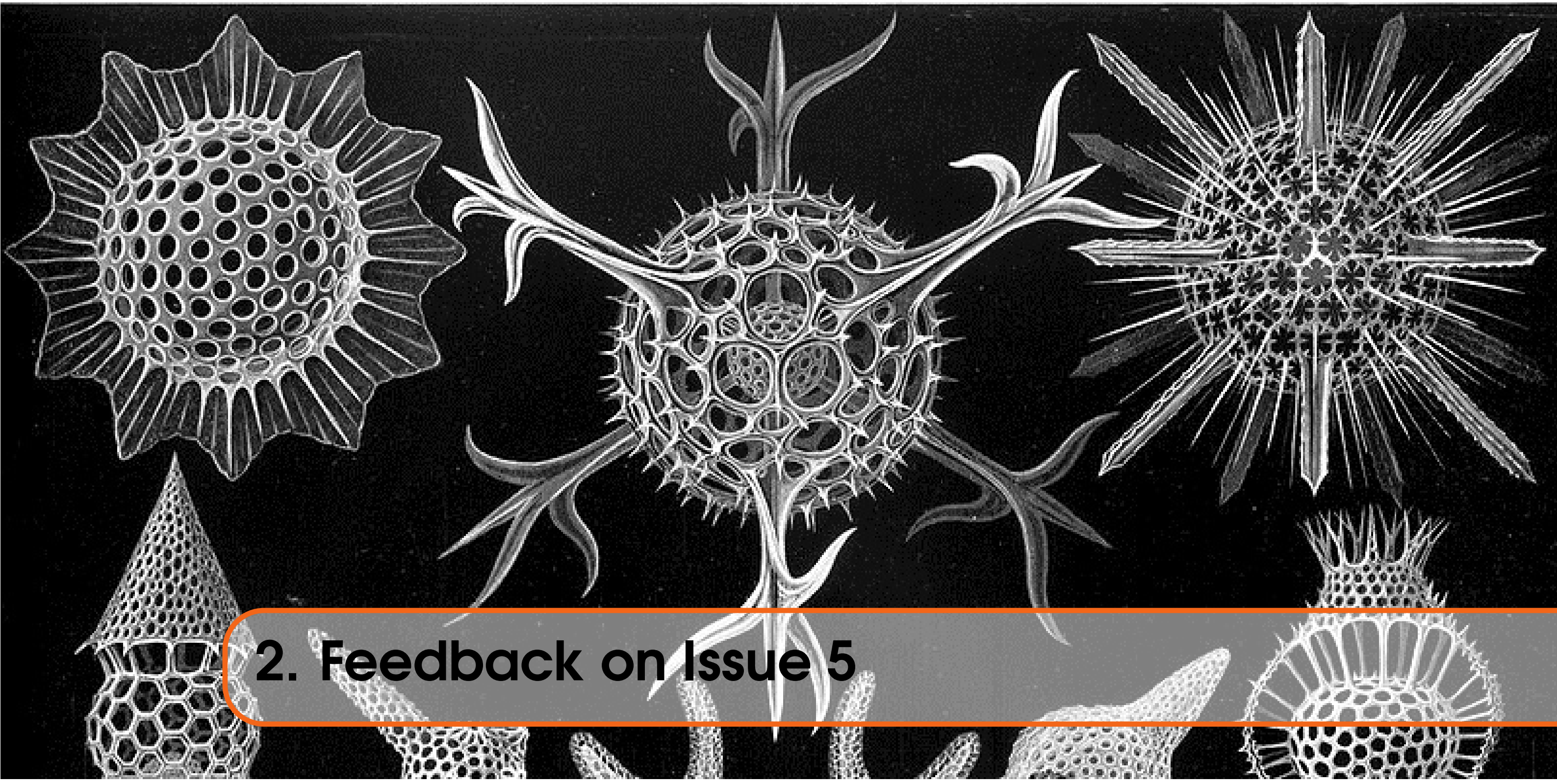
I'm rather hoping that this mailing list will not be a one-way affair, like QL Today appeared to be. I'm very open to suggestions, opinions, articles etc from my readers, otherwise how do I know what I'm doing is right or wrong?

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 5

While looking at something else, I noticed that in Issue 4, Page 28, Section 4.2.1 the following text:

For example, in many opcodes, the size of the operand - .B, .W or .L - is specified in bits 5 and 6 of the opcode instruction word in memory.This is therefore a 2 bit wide bit field, starting at bit $31 - 5 = 26$ *and would be represented as follows:*

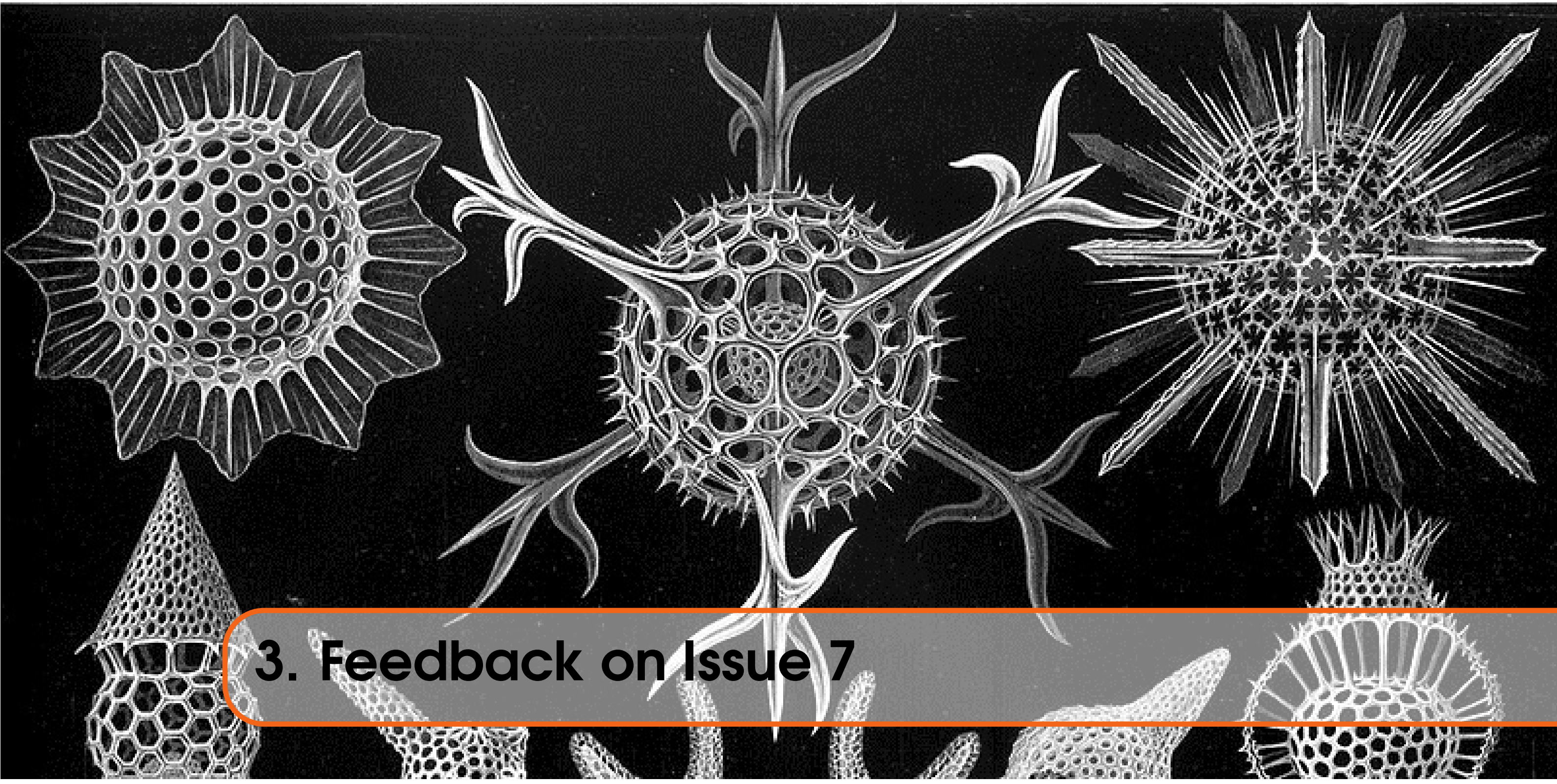
```
{26:2}
```

I thought something was wrong, but didn't know which bit(!) was incorrect, so I worked through it and of course, the two bits in the bit field are wrong! The text should read as follows:

For example, in many opcodes, the size of the operand - .B, .W or .L - is specified in bits 6 and 5 of the opcode instruction word in memory.This is therefore a 2 bit wide bit field, starting at bit $31 - 6 = 25$ and would be represented as follows:

```
{25:2}
```

I had of course specified a bit field for bits 5 and 4 and not for bits 6 and 5. I did mention that there was a certain amount of confusion in bit field specifications!



3. Feedback on Issue 7

Well now, here's a thing. Very quickly after Issue 7 "hit the streets" I got feedback from two different people. Thanks very much to Wolfgang and to Marcel for their input, and their permission to publish.

3.1 Feedback from Wolfgang Lenerz

[WL] Just a little comment: there is a typo on page 16, in the third code extract at line 1: Tobias makes a `MOVEM` to ...`a2-a7` : it should be to ...`a2-a6`.

[ND] Thanks. I don't have a Q68 (yet?) and I really didn't have much to do with Tobias's article to get it into the eComic, so I didn't notice that slight error. I fixed it in the PDF download on 1st October 2019 at around 19:00 BST (UTC + 01:00) - so anyone who downloaded prior to that time might wish to download again to get the correction.

[WL] Also a more general comment, which I offer as constructive criticism: in the `utf82ql` routine, when handling values over 127 (i.e. at least 2 bytes), why check for the special cases first (arrows, pound etc) before getting the values from the table? Wouldn't it be better to leave their place in the table at 0 as well, and every time you hit a 0 in the table you check for the exception?

[ND] Good point, thanks. That would have made more sense as the processing is more likely to be processing valid characters than the exceptions. I thought I was doing well getting the exceptions in what I thought was the most likely order!

[WL] Oh, and this probably doesn't get said often enough : *really enjoy reading your prose!*

[ND] Thanks. It's nice to get feedback, but much nicer to get compliments.

3.2 Feedback from Marcel Kilgus

[MK] As a pedantic ass I have to object so sentences like these:

The UK Pound symbol is character 96 ($60) on the QL, but in ASCII it is character 163 ($A3)" (etc.)

[ND] I like pedants! My wife says I am one, then she corrects me at every available opportunity!

[MK] ASCII is, by definition, 7-bit, so it cannot contain a character with the number 163. The tale of characters 128-255 is one fought in many battles. Linux tended to be "ISO 8859-1" and later "ISO 8859-15" before they adopted UTF-8, on Windows you will mostly find the "Windows-1252" encoding. These are very similar, but differ when it comes to the Euro sign for example (ISO 8859-1 is too old to have a Euro sign and the others have adopted it in different places).

[ND] Technically, I agree, ASCII is indeed 7 bit and 163 is definitely not 7 bit. But let's face it, there have been 8 bit "ASCII" characters for many years, even when I was at college back in the, ahem, early eighties, ASCII was (at least, considered) 8 bit - whether pedantically correct or not. However, true ASCII is 7 bit.

[ND] I remember many occasions, back when `config.txt` was still a thing, trying to set up the correct code page for a system. A nightmare as there was no Google back then to help out, just the manual for whatever system I was installing or working with.

[ND] I am led to understand, however, that ISO/IEC 4873 introduced some extra control codes “characters”, in the $80 to $9F hexadecimal range, as part of extending the 7-bit ASCII encoding to become an 8-bit system.[1] However, I sit corrected on the 7/8 bit point. Thanks.

[MK] But, and that is the important thing, Unicode was made to unify them all. And UTF-8 is a pretty darn cool invention, unfortunately it came too late for Windows, which was a very early adopter of Unicode at a time when everybody thought "65536 characters ought to be enough for everyone!". So Windows started to used 16-bits for every character ("UCS-2" encoding), which makes coding somewhat weird, and then they found out that 65536 characters are not enough after all, so now Windows uses UTF-16, which is UTF-8's big brother, with sometimes 2 bytes per character and sometimes 4. What a mess. But when it comes to data storage UTF-8 is the way to go these days, always!

[ND] It sure is a mess, and yes, UTF-8 is the way to go. As I mentioned XML files depend on it, the web is pretty much full of it in all those HTML files etc. And, once you get your head around the difference between a “code point” and the character's actual bytes, it's pretty easy to understand.

[ND] I'm not so sure that Windows is missing out or behind the times though. At work, my files are all pretty much UTF-8 (I write my documents in `ASCIIDOCTOR`[2] format and convert them to PDF files using `asciidoctor-pdf` - if I need Office flavoured docs, I use `pandoc` to convert to something in DOCX format - but I almost never use those. `Asciidoctor` files are plain text, and very easy to version control! `Notepad++` or `VSCodium` are my text editors of choice and both save in UTF-8 with no problems. Even Notepad itself can read the files - and I suspect Windows 10 will be better, I'm on Windows 7. (Currently)

[ND] Mind you, those damned so-called "smart" quotes that Office documents insist on using mess things up truly. It's the first thing I turn off with my Office stuff, and every slight update or patch seems to turn them back on! So annoying.

[MK] For QPC I already implemented these translations 20 years ago when copying text to/from the clipboard. But well done for bringing UTF-8 to the QL

[1]The Unicode Consortium (October 27, 2006). "Chapter 13: Special Areas and Format Characters" (PDF). In Allen, Julie D. (ed.). The Unicode standard, Version 5.0. Upper Saddle River, New Jersey, US: Addison-Wesley Professional. p. 314. ISBN 978-0-321-48091-0. Retrieved March 13, 2015.

[2]Now that's ironic!

[ND] Well, thanks for the reminder of how old I'm getting! The reason I did the utilities was simple, I had one of those itches to scratch. When I did a bit of work with Jan on his updated QL Monitor, I used a Linux system to do the typing - it's what I'm used to - and those arrow characters caused me no end of grief, as did the copyright and the pound signs. I messed about there using actual, ahem, ASCII codes (sorry!) but now, I don't have to.

[ND] Oh, and *thank you* for `QPC2`, it's my favourite QL program of all time, and it simply "just" works on Linux under `Wine`. I did have some problems recently with it not working, but I traced that to a mix and match installation with bits of `Wine 3` and bits of `Wine 4` living together in sin.

`QPC2` is what has kept me in the QL scene for as long as I can remember - I always got somewhat tired of the QL, the cables, the hard drive, the noise, the length of table I needed with limited space in my flat (apartment) and so on. With `QPC2` it's all on my laptop. Nice and compact.

[ND] And, by the way, *I am a pedant's baddest nightmare!*

3.3 *More* Feedback from Wolfgang Lenerz

[WL] I had a longer look at ql2utf8. I hope you don't mind a few more comments.

[ND] No, I like getting comments - most people do assembly better than I do!

[WL] When I tried to compile the source file, I couldn't, as the different traps weren't defined.

```
io_fbyte        equ 1
io_sbyte        equ 5
mt_frjob        equ 5
```

[ND] Were you using `QMAC` by any chance? I know that's a recurring problem for `QMAC` as `GWASS` and `GWASL` come with the various traps and vectors "automatically" included. If I include them in the source, then they won't assemble for me, I get an error about duplicate definitions.

[WL] Moreover, I got a few errors that some bra.s were out of reach.

[ND] Hmmm, I just recompiled with `GWASS` and got no errors at all. However, I did get one error with `GWASL`. Looking at the listing file, it's complaining that the label `oneByte` is an "illegal instruction" - weird. I remember `GWASL` doing that on a few occasions in the past. I used to edit the sources, rub out the label, and type it in again, that usually worked. I could never trace it to hidden characters etc as a hex dump of the source showed nothing out of the ordinary.

I don't however, get any errors about short branches being out of reach.

[WL] It seems to me that the two lines of code at label `TestBit7` are superfluous: you are doing the exact same test just beneath it, at label `Testpound`.

[ND] That's a typing error. Originally I only had the BVS.S instruction rather than the BTST #7 so in theory, if D1 was loaded with a byte >= $80 the V flag would be set. Unfortunately it didn't work. I traced the code under QMON2 and by the time we get to that point, the V flag is clear, always. I obviously forgot to remove the BVS when I edited the code to add in the BTST #7 instruction. My mistake.

[WL] You could replace the two instructions at Label `OneByte` with the single instruction `PEA readLoop`.

[ND] I see what you mean, if I do that replacement, then instead of branching to `writeByte` and returning and then branching off to `readLoop`, just drop in to `writeByte` and return automagically to the top of the loop. Nice!

[WL] Then you could also just delete the last two instructions of label `gotCopyright` (no need to bsr.s writebyte, you just fall through) and you could also replace, in the different `gotxxxx` routines (e.g. `gotEuro`, `gotGrave` etc) the two instructions:

```
bsr.s writeByte
bra readLoop
```

with a simple:

```
bra.s oneByte
```

and it might be able to use a bra.s rather than a bra somewhere in the changed code.

[ND] Yes, that all makes perfect sense given the above changes to `oneByte`.

[WL] At label `notArrows`, you might want to replace these 4 lines

```
addq.b  #1,d2
move.b  0(a2,d2.w),d1    ; Second byte
bsr     writeByte        ; Send it out
bra     readLoop         ; Go around.
```

with these two

```
move.b  1(a2,d2.w),d1    ; Second byte
bra     oneByte
```

[ND] Yes, that makes perfect sense too. Thanks.

[WL] I would set the output & input channel IDs into two registers (eg A4, A5) and move them into A0 when needed in the byte read/write subroutines, instead of accessing the stack (and thus memory) every time with a LEA.

[ND] I used A4 and A5 for that very purpose in the following chapter, in the `Utf82ql` code, and forgot to go back and fix this code to do the same.

[WL] Finally, I would also include test at label `notArrows` to make sure that the byte in D1 doesn't exceed the max value of your table. I know that values above that are not printable characters, but it *is* possible to include them in a text file. You might want to tell the user that some characters couldn't be translated...

[ND] Yes indeed, that was an oversight. Thanks for pointing it out.

[WL] Hope you don't mind the above.

[ND] Not at all, many thanks indeed.

So, given all those amendments, here for your delectation is the latest version of the `Ql2utf8` code, incorporating all of Wolgang's changes and corrections.

3.4 *Even More* Feedback from Wolfgang Lenerz

[WL] I also had a look through the `utf2ql` code. Some of the comments made for the other routine (`ql2utf8` ND) may also apply here, no need to go through them again. Here I have some more comments on this routine.

[ND] Ok, I'm sitting comfortably ....

[WL] The first one is not really about the code itself, but the way you structured it. Of course, this is much of a personal preference, so please take this with a pinch (or even a spoonful) of salt. Leaving out the exception and read/write routines, your code is structured thus:

```
readLoop
   get byte
   leave if EOF

Multibytes
   is it two bytes?
   yes -> jump to handle_two

NotTwo
   is it three bytes?
   yes -> jump to handle_three

Error
   not three bytes, return error

handle_two
   treat two bytes
   bra     readloop

handle_three
   treat three bytes
   bra     readLoop
```

[WL] For me, you have 6 different blocks of code. I would prefer the following structure with 4 blocks (making the code less "spaghetti"):

```
readLoop
   get byte
   leave if EOF

handle_two
   is it two bytes?
   no-> jump to handle three
   treat two bytes
   bra     readloop

handle_three
   is it 3 bytes?
   no -> jump to error
   treat three bytes
   bra     readLoop

Error
```

```
    not three bytes, return error
```

[ND] Yes, I admit that sometimes my structure leaves a lot to be desired and you are correct in what you say above - I must try harder!

[WL] Leaving out the branches to the loop, basically your way of doing it is:

```
is it something?
  yes, go off, handle it_1
is it something else?
  yes, go off and handle it_2
error
handle_it1
handle_it2
```

[WL] Whereas mine is:

```
is it something?
  no, go off to next check

handle it1
  is it something else?
    no, go to error

handle it2

error
```

[WL] Again, this is a personal preference: There is no functional difference, but I, personally, find the second one easier to read if you want to follow the flow of the code.

[ND] Agreed.

[WL] But in doing so it will allow you to write the code at the `multiBytes` label so:

```
multiBytes
  move.b    d1,d2
  andi.b    #%11100000,d2    ; <-- BUG HERE? [ND]
  cmp.b     #%11000000,d2    ; 2 bytes?
  bne.s     threebytes       ;  ... no->

twoBytes
  (treat 2 bytes including exceptions)

testThree   (no need to copy d1 into d2 again)
  cmp.b     #%11100000,d2    ; 3 bytes?
  bne       invalidUTF8      ;  ... no ->
  ...
```

[ND] Hmm, I think you have a bug there. For three byte characters the top nibble should be 1110, so your mask is missing a '1' bit. I suspect you intended to type the following for `multiBytes`:

```
multiBytes
  move.b     d1,d2
  andi.b     #%11110000,d2
```

Otherwise you are forcing bit 4 of D2 to always be a zero. However, that minor niggle aside, I like your version better than mine as I/we only need a single ANDI instruction which keeps the top nibble, which can then be compared to check for two byte (110x) or three byte (1110) characters. Far more efficient indeed.

[WL] The `scanTable` routine is probably the most time consuming part of the code, so I'd have written it as follows:

```
scanTable
  move.l     a2,a3                  ; point to table
  move.w     #59,d0                 ; there are 60 words to compare

scanLoop
  cmp.w      (a3)+,d2               ; is it a match?
  beq.s      scanDone               ;  ...yes ->
  dbf        d0,scanLoop            ; try all permitted values
  rts                               ; no match found, return NZ from cmp

scanDone
  move.l     a3,d0                  ; where we found it (+2)
  sub.l      a2,d0                  ; index into table
  subq.w     #2,d0                  ; but we overshot by 2 bytes

  lsr.w      #1,d0                  ; offset into index
  add.w      #$80,d0                ; convert to character code
  cmp.w      d0,d0                  ; see below
  rts                               ; the condition code Z is set by the cmp
```

[ND] Curses, I've been found out! My way was easier for me as I didn't have to count up however many two byte characters there were! However, as they say about Unix/Linux, there's more than one way to skin a cat, but again, I prefer your method.

[WL] There are a few more instructions when you find the correct value, but the search loop itself is smaller and will be faster (unless the value searched for is the very first in the table, and even then it'll be a close match). The CMP D0,D0 is there so that the routine returns with the Z flag set, without affecting any other register by zeroing it.

[ND] I wasn't fond of the non-standard way of detecting an error in my version, I have to admit. This is far far better.

[WL] So, coming back from calling the routine at label `doScan`, a simple BNE.S ERROR will do:

```
doScan
  bsr.s     scanTable
  bne.s     invalidUTF8
  (... success in d0 ...)
```

[ND] Agreed, this is better and resembles more a standard error return, zero is good, non-zero is not good.

[WL] At label `twoBytes`, you should be able to write:

```
twoBytes
  lsl.w     #8,d1               ; move byte up
  bsr       readByte            ; get next byte into LSB of D1
```

[WL] You should now have the correct word in D1. Remember, though, as of then to test on D1, not D2, for valid utf, even in the `scanTable` loop.

Note, this presumes that the trap handler does its work correctly and *only* modifies the LSB[3] of D1 to put the returned value in there. (Unlike, e.g. some early versions of SMSQmulator which just reset the *entire* register to 0 and then sets the byte. Ouch!)

[ND] I think that I shall leave the code using D2, just in case it causes problems elsewhere then. Better safe than sorry.

[WL] Most of these comments go a little beyond just checking the code itself, I hope you don't mind.

[ND] No, I don't mind and in fact I welcome comments on anything printed in this ePeriodical. If you have a problem with my writing style, code etc, I'm happy to hear from you. From anyone that is!

3.5 A Better Ql2utf8

```
;------------------------------------------------------------------------------
; QL2UTF8:
;
; This filter converts QL text files to UTF8 for use on Linux, Mac or
; 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
; 07/10/2019 WLenerz Many improvements.
;------------------------------------------------------------------------------
; (c) Norman Dunbar, Wolfgang Lenerz 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
```

[3]LSB = Lowest Significant Byte.

```
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         ; Bad parameter
err_eof      equ      -10         ; End of file
err_or       equ      -4          ; Out of range
me           equ      -1          ; This job's id
timeout      equ      -1          ; Infinty, and beyond!

;-----------------------------------------------------------------------
; Uncomment the following if you are using QMAC as your assembler.
;-----------------------------------------------------------------------
; io_fbyte     equ      1          ; Fetch one byte
; io_sbyte     equ      5          ; Send one byte
; mt_frjob     equ      5          ; Force remove a job
;-----------------------------------------------------------------------

;=======================================================================
; 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 1.01'
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

```

```
;------------------------------------------------------------------------------
; 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
    move.l  sourceID(a7),a4     ; Input channel id
    move.l  destId(a7),a5       ; Output channel id

;------------------------------------------------------------------------------
; The main loop starts here. Read a single byte, check for EOF etc.
;------------------------------------------------------------------------------
readLoop
    moveq   #io_fbyte,d0        ; Fetch one byte
    move.l  a4,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!

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

;------------------------------------------------------------------------------
; 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
    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
    bra.s   oneByte             ; Write out & carry on.

;------------------------------------------------------------------------------
; 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             ; Then drop in to write & carry on

```

```
;------------------------------------------------------------------------------
; All other ASCII characters, below $80, are single byte in UTF8 and
; are the same code as in ASCII. Stack the address of readLoop and
; drop into writeByte. On RTS, we will hit the top of the loop again.
; (Courtesy Wolfgang Lenerz.)
;------------------------------------------------------------------------------
oneByte
    pea     readLoop

;------------------------------------------------------------------------------
; A small but perfectly formed subroutine to send the byte in D1 to
; the output channel.
;------------------------------------------------------------------------------
writeByte
    moveq   #io_sbyte,d0        ; Send one byte
    move.l  a5,a0               ; Output channel id
    trap    #3
    tst.l   d0                  ; OK?
    bne     errorExit           ; Oops!
    rts

;------------------------------------------------------------------------------
; 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)
; 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!
    bra.s   oneByte             ; Write out & carry on

;------------------------------------------------------------------------------
; 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
    bra.s   oneByte             ; Write out and carry on

;------------------------------------------------------------------------
; 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
    subq.b  #4,d2               ; Arrows = 0-3. C clear is bad
    bcc.s   notArrows           ; Still not an arrow.

gotArrows
    subi.b  #$bc,d1             ; Correct arrow code, 0 - 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
    bra.s   oneByte             ; Write it & go around again.

;------------------------------------------------------------------------
; 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!

;------------------------------------------------------------------------
; Now we are certain, everything is two bytes. Read them from the
; table and write them out. However, there are only 60 entries in the
; table - best we check!
;------------------------------------------------------------------------
notArrows
    cmpi.b  #59,d1              ; Are we in range for the table?
    bcc.s   inRange             ; Yes

outOfRange
    moveq   #err_or,d0          ; Out of range
    bra.s   errorExit           ; Oops!

inRange
    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
    move.b  1(a2,d2.w),d1       ; Second byte
    bra     oneByte             ; Write it and go around

```

```

;------------------------------------------------------------------------------
; 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

;------------------------------------------------------------------------------
; 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!) There are 60
; QL characters which convert to two byte UTF8 characters.
;------------------------------------------------------------------------------
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
```

Listing 3.1: Wolfgang's improved ql2utf8 Utility

3.6 A Better Utf82ql

```
;------------------------------------------------------------------------------
; UTF82QL:
;
; This filter converts UTF8 text files from Linux, Mac or Windows to
; to the SMSQ character set.
;
;
; EX utf82ql2_bin, input_file, output_file_or_channel
;
;------------------------------------------------------------------------------
; 28/09/2019 NDunbar Created for QDOSMSQ Assembly Mailing List.
; 07/10/2019 WLenerz Many improvents.
;------------------------------------------------------------------------------
; (c) Norman Dunbar, Wolfgang Lenerz, 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
utf8Pound    equ      $c2a3      ; UTF8 Pound sign
qlPound      equ      96         ; QL Pound sign

utf8Grave    equ      96         ; UTF8 Grave code
qlGrave      equ      159        ; QL Grave code

utf8Copyright equ     $c2a9      ; UTF8 copyright
qlCopyright  equ      127        ; QL copyright sign

qlEuro       equ      181        ; SMSQ Euro symbol

err_exp      equ      -17
err_bp       equ      -15
err_eof      equ      -10
err_or       equ      -4
me           equ      -1
timeout      equ      -1

;------------------------------------------------------------------------
; Uncomment the following if you are using QMAC as your assembler.
;------------------------------------------------------------------------
; io_fbyte     equ      1          ; Fetch one byte
; io_sbyte     equ      5          ; Send one byte
; mt_frjob     equ      5          ; Force remove jobs
;------------------------------------------------------------------------


;========================================================================
; 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     'UTF82QL'
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

;------------------------------------------------------------------------
; Initialise a couple of registers that will keep their values all
; through the rest of the code.
;------------------------------------------------------------------------
ql2utf8
    lea     utf8,a2             ; Preserved throughout
    moveq   #timeout,d3         ; Timeout, also Preserved
    move.l  sourceId(a7),a4     ; Channel ID for UTF8 input file
    move.l  destId(a7),a5       ; Channel ID for QL output file

;------------------------------------------------------------------------
; The main loop starts here. Read a single byte, check for EOF etc.
;------------------------------------------------------------------------
readLoop
    bsr     readByte            ; Read one byte
    beq.s   testBit7            ; No errors is good.
    cmpi.l  #err_eof,d0         ; All done?
    beq     allDone             ; Yes.
    bra     errorExit           ; Oops!

;------------------------------------------------------------------------
; Test the top bit here. If it is zero, we are good for most single
; byte characters, otherwise it is potentially multi-byte.
;------------------------------------------------------------------------
testBit7
    btst    #7,d1               ; Bit 7 set?
    bne.s   multiBytes          ; Multi Byte character if so

;------------------------------------------------------------------------
; In UTF8, the Grave accent (backtick) is a single byte character but
; the byte value doesn't correspond to that on the QL. On UTF8 it is
; $60 (96) but on the QL it is $9F (159) so, this is another Sir
; Clive induced exception!
;------------------------------------------------------------------------
testGrave
    cmpi.b  #utf8Grave,d1       ; Got a grave!
    bne.s   oneByte             ; Must be a single byte if not a pound.

gotGrave
    move.b  #qlGrave,d1         ; Write a grave character
```

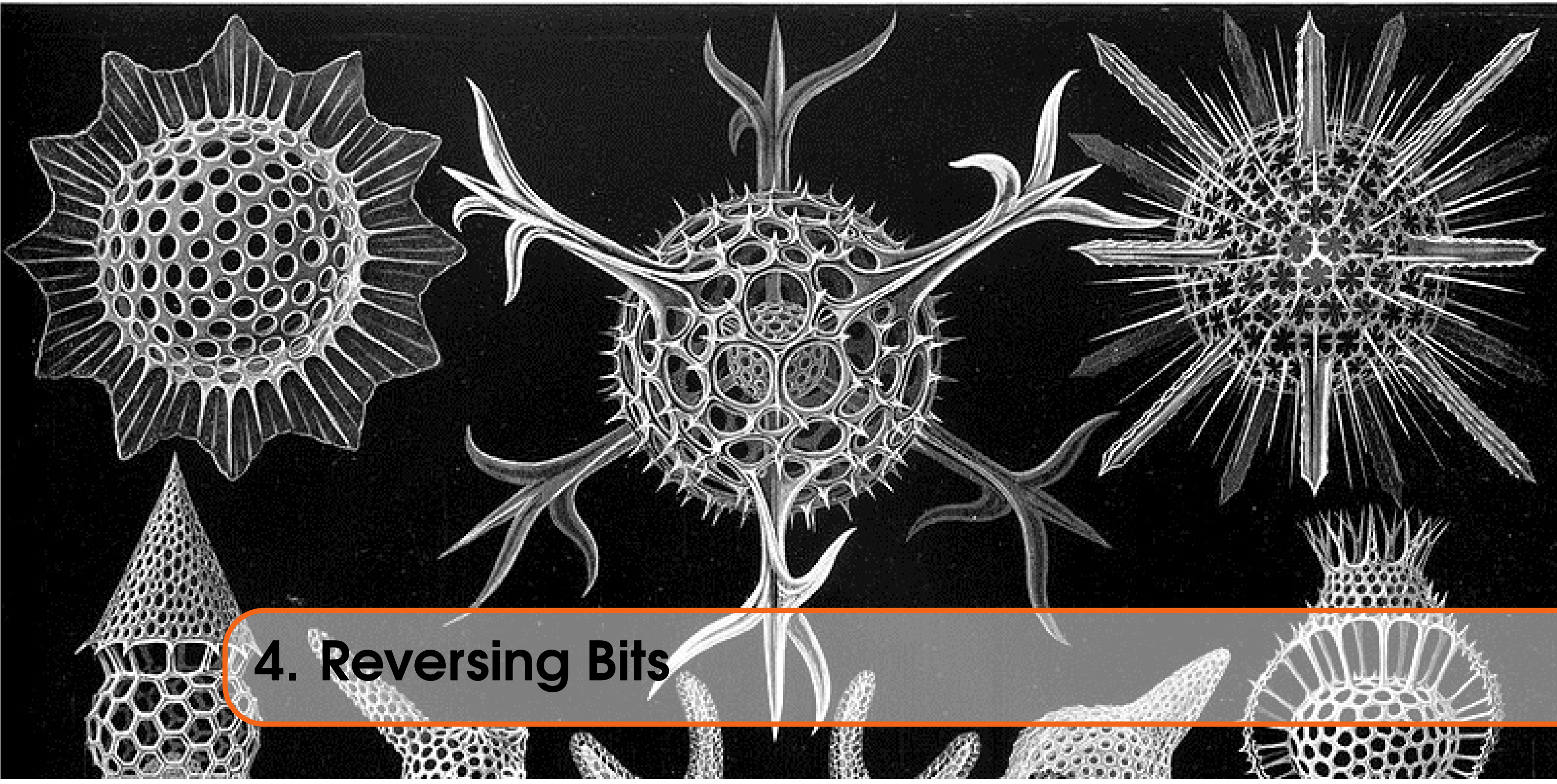
```

;------------------------------------------------------------------------------
; The byte read is a valid single byte character so it has the exact
; same code in the QL's variation of ASCII, just write it out.
;------------------------------------------------------------------------------
oneByte
    bsr     writeByte           ; Write the byte out
    bra.s   readLoop            ; And continue.


;------------------------------------------------------------------------------
; Most of the remaining characters will be two bytes in UTF8 and one
; byte on the QL. There are a few exceptions though - the Euro and
; the four arrow keys are three bytes long in UTF8.
;------------------------------------------------------------------------------
multiBytes
    move.b  d1,d2               ; Copy character code
    andi.b  #%11110000,d2       ; Keep top four bits
    cmpi.b  #%11000000,d2       ; Two bytes?
    bne.s   testThree           ; Yes.

;------------------------------------------------------------------------------
; At this point we should have a UTF8 two byte character but we only
; have the first byte in D1. We need the second byte also, so read it
; and check that it is indeed valid.
;------------------------------------------------------------------------------
twoBytes
    move.b  d1,d2               ; Save the leading byte
    bsr     readByte            ; Read the second byte
    lsl.w   #8,d2               ; Shift first byte upwards
    or.b    d1,d2               ; And add the new byte

;------------------------------------------------------------------------------
; Exception checking. UTF8 codes $C2A3 for the UK Pound and $C2A9 for
; copyright, are not in the table. They are QL codes $60 (96) and $7F
; (127) and are exceptions to the rule that a QL code less than 128
; always has a one byte code in UTF8 - they are both two bytes.
;------------------------------------------------------------------------------
testPound
    cmpi.w  #utf8Pound,d2       ; Got a UK Pound?
    bne.s   testCopyright       ; No

gotPound
    move.b  #qlPound,d1         ; QL Pound code
    bra.s   oneByte             ; Write it out & loop around

testCopyright
    cmpi.w  #utf8Copyright,d2 ; Got a copyright?
    bne.s   doScan              ; No

gotCopyright
    move.b  #qlCopyright,d1
    bra.s   oneByte             ; Write it out & loop around

;------------------------------------------------------------------------------
; Ok, exceptions processed, do the remaining two byte characters.
```

```
;-------------------------------------------------------------------------------
doScan
        bsr       scanTable         ; Is this valid UTF8?
        bne.s     invalidUTF8       ; Nope

validUTF8
        move.b    d0,d1             ; Get the character code
        bsr.s     writeByte         ; Write it out
        bra       readLoop          ; And continue.

invalidUTF8
        moveq     #err_exp,d0       ; Error in expression
        bra       errorExit         ; Bale out.

;-------------------------------------------------------------------------------
; We are interested in a few three byte characters, so we check those
; next. These are identified by the top nibble of the first character
; read in being 1110.
;-------------------------------------------------------------------------------
testThree
        cmpi.b    #%11100000,d2     ; Three bytes?
        bne.s     invalidUTF8       ; No.

;-------------------------------------------------------------------------------
; At this point we should have a UTF8 three byte character but we
; only have the first byte in D1. We need the second byte also, so
; read it and check that it is indeed valid. Then get the third byte.
; All our three byte characters should have $E2 in the first byte.
;
; The Euro is $E282AC.
; The Arrows are $E2869x where 'x' is 0,1,2 or 3.
;-------------------------------------------------------------------------------
threeBytes
        cmpi.b    #$e2,d1           ; Valid three byte?
        bne.s invalidUTF8           ; Looks unlikely.

        move.b    d1,d2             ; Save the first byte
        bsr.s     readByte          ; Get the second byte
        cmpi.b    #$82,d1           ; Euro second byte?
        beq.s     threeValid         ; Yes
        cmpi.b    #$86,d1           ; Arrow second byte?
        bne.s     invalidUTF8       ; Sadly, no, error out.

threeValid
        lsl.w     #8,d2             ; Shift first byte upwards
        or.b      d1,d2             ; And add the new byte
        bsr.s     readByte          ; Get the third byte
        cmpi.w    #$e282,d2         ; Euro possibly?
        bne.s     threeArrows       ; No, try arrows

;-------------------------------------------------------------------------------
; We have read $e282 so if we get $ac next, we have the euro. If not
; it's an error in the UTF8 characters that the QL understands.
;-------------------------------------------------------------------------------
threeEuro
        cmpi.b    #$ac,d1           ; Need this for the Euro
```

```
    bne.s     invalidUTF8        ; No, error out again.
    move.b    #qlEuro,d1         ; QL Euro code
    bsr.s     writeByte          ; Write it out
    bra       readLoop           ; And continue.


;----------------------------------------------------------------
; The QL arrows are $BC, $BD, $BE and $BF (left, right, up, down).
; The UTF8, $E2869x where 'x' is 0, 2, 1 or 3 to correspond with the
; order of the QL arrow codes.
;----------------------------------------------------------------
threeArrows
    cmpi.w    #$e286,d2          ; Got a potential arrow code?
    bne.s     invalidUTF8        ; Fraid not, error out.
    subi.b    #$90,d1            ; D1 is now 0-3 for valid arrows
    bmi.s     invalidUTF8        ; Oops, it went negative
    cmpi.b    #3,d1              ; Highest arrow code
    bhi.s     invalidUTF8        ; Oops, invalid arrow code.
    addi.b    #$bc,d1            ; Convert to QL arrow code.
    bsr.s     writeByte          ; Write it out
    bra       readLoop           ; And continue.


;----------------------------------------------------------------
; A small but perfectly formed subroutine to send the byte in D1 to
; the output QL file.
; On Entry, A0 = input channel ID and A3 = output channel ID.
; On exit, D0 = 0, Z set.
; On error, never returns.
;----------------------------------------------------------------
writeByte
    move.l    a5,a0              ; Get the correct channel ID
    moveq     #io_sbyte,d0       ; Send one byte
    trap      #3
    tst.l     d0                 ; OK?
    bne.s     errorExit          ; Oops!
    rts

;----------------------------------------------------------------
; Another perfectly formed subroutine to read one byte into D1
; from the input UTF8 file.
; On Entry, A0 = output channel ID and A3 = input channel ID.
; On exit, error codes in D0, Z set if no error and D1.B = character
; just read.
;----------------------------------------------------------------
readByte
    move.l    a4,a0              ; Get the correct channel ID
    moveq     #io_fbyte,d0       ; Fetch one byte
    trap      #3                 ; Do input
    tst.l     d0                 ; OK?
    rts

;----------------------------------------------------------------
; Scan the UTF8 table looking for the word in D2. If found, we have
; the table offset in D0 and that is then halved to get the index which
; is still $80 below the correct character code - we add to convert.
; Returns with D0 = the character code, or $FFFF to show the end was
; reached and we appear to have an invalid two byte character. A2
; holds the table address. D7 is a working register.
;------------------------------------------------------------------------
scanTable
    move.l  a2,a3           ; Get start of table
    move.w  #59,d0          ; There are 60 entries in the table

scanLoop
    cmp.w   (a3)+,d2        ; Found it yet?
    beq.s   scanDone        ; Yes
    dbf     d0,scanLoop     ; No, try again
    rts                     ; Not found, Z not set.

scanDone
    move.l  a3,d0           ; Address in table + 2
    sub.l   a2,d0           ; Address now the Offset + 2
    subq.w  #2,d0           ; Adjusted to correct offset
    lsr.w   #1,d0           ; Conver to index
    add.w   #$80,d0         ; Now correct character code
    cmp.w   d0,d0           ; Sets Z flag
    rts

;------------------------------------------------------------------------
; 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

;------------------------------------------------------------------------
; The following table contains the two byte sequences required for
; QL characters from character $80 onwards. Those flagged as $FFFF
; are exceptions, dealt with in the code. There are no entries for
; the arrow keys as they would simply be zero words at the end of the
; table.
;------------------------------------------------------------------------
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    $ffff           ; 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    $ffff           ; 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

        dc.w    $0000               ; End of table
```

Listing 3.2: Wolfgang's improved utf82ql Utility



4. Reversing Bits

Many years ago, I needed a routine to reverse the bits in a register so that bit 0 ended up in bit 31, bit 31 was in bit 0 and so on. I think I asked on the QL Mailing List and the responses I received were pretty similar to the method I knew about - shifting bits right from the input register through the Carry Flag and shifting left into another register. It worked fine but I always thought that there would be a *better* solution. I never found one.

The other day, while doing some embedded (aka Arduino) fiddling, I found a piece of code to reverse the order of bits in an 8 bit register, which is what the Arduino's ATmega328P microcontroller has. I had a look at the code and decided that I could adapt it to reverse all 32 bits on a QL register. This is what I came up with.

Bear in mind that in order to reverse 32 bits through the Carry Flag you only need three registers - the source, the destination and a counter for the 32 shifts for each register. That takes a total of 64 one bit shifts to reverse all the bits.

4.1 Reversing 2 Bits

Might as well start easy. If we start with the value 10 in our two bit register, we can reverse the value to 01 as follows:

- AND 10 *with 10;*
- Shift the result *right* by one bit;
- AND 10 with 01;
- Shift the result *left* by 1 bit;
- OR the results of the two AND operations.

So much for the theory, let's see if it works:

```
10 AND with 10 = 10.
10 >> 1 = 01.
```

```
10 AND with 01 = 00.
00 << 1 = 00.

10 OR 00 = 10.
```

Easy? We started with 10 and finished with 01, job done, we reversed the two bits. So far so good, lets up things a bit and see what happens with 4 bits.

Note: You can, if you wish, shift first then AND, it still works.

4.2 Reversing 4 Bits

To reverse 4 bits, we would do something similar. If we start with the value 1101, then all we have to do is:

- AND 1101 with 1100;
- Shift the result *right* by two bits;
- AND 1101with 0011;
- Shift the result *left* by two bits;
- OR the two results of the AND operations.

Again, let's see if it works:

```
1101 AND 1100 = 1100.
1100 >> 2 = 0011.

1101 AND 0011 = 0001.
0001 << 2 = 0100.

0011 OR 0100 = 0111.
```

Oops! That's not quite right. All we have really done, and indeed, in the first two bit example, is *swap* the top two bits with the bottom two bits, we have not *reversed* them. We need to now swap the two pairs of two bit values in the above result, 0111. Let's continue.

We are currently have 0111 as our intermediate result. This is 2 two bit values, 01 and 11. We know that swapping the 2 bits in a two bit value reverses them. Can we now reverse the pair of two bit values *at the same time*?

```
0111 AND 1010 = 0010.
0010 >> 1 = 0001.

0111 AND 0101 = 0101.
0101 << 1 = 1010.

0001 OR 1010 = 1011.
```

So, we started with 1101, swapped the two pairs of two bit values over, then reversed both bits in each pair to receive the result 1011 which is a complete bit reversal of the original 4 bits.

4.3 Reversing 8 Bit values

In theory then, we should be able to start with 8 bits, swap the two pairs of 4 bits over, then swap the 4 pairs of two bit values over, then reverse the bits in those 4 two bit values.

To swap the 4 bit values we follow the same principle as above:

- AND the value with 11110000;
- Shift the result *right* by 4 bits;
- AND the value with 00001111;
- Shift the result *left* by four bits;
- OR the two results of the AND operations;
- Then carry out the steps for a 4 bit swap but do two at a time;
- Swap/reverse the bits in each of the resulting two bit values.

Does it work? Lets try with $C9 or 11001001:

```
11001001 AND 11110000 = 11000000.
11000000 >> 4 = 00001100.

11001001 AND 00001111 = 00001001.
00001001 << 4 = 10010000.

00001100 OR 10010000 = 10011100.
```

That's the 2 four bit values exchanged, but not yet reversed. We continue with the process to swap the 2 two bit values in each of the 4 bit values:

```
10011100 AND 11001100 = 10001100.
10001100 >> 2 = 00100011.

10011100 AND 00110011 = 00010000.
00010000 << 2 = 01000000.

00100011 OR 01000000 = 01100011.
```

Now we simply reverse the bits in each of the 4 two bit values:

```
01100011 AND 10101010 = 00100010.
00100010 >> 1 = 00010001.

01100011 AND 01010101 = 01000001.
01000001 << 1 = 10000010.

00010001 OR 10000010 = 10010011.
```

And that's working correctly too, 11001001 has been bit reversed to become 10010011.

4.4 Reversing 16 Bit Values

With a 16 bit value, we would:

- Swap the pair of 8 bit values around;
- Swap the 4 four bit values around;
- Swap the 8 two bit values;
- Reverse the 8 two bit values.

Do you see a pattern developing? To swap the two n/2 bit values in an ’n’ bit value:

- AND the value with a mask of n/2 ones and n/2 zeros;
- Shift the value *right* by n/2 bits;
- AND the value with a mask of n/2 zeros and n/2 ones;
- Shift the value *left* by n/2 bits;
- OR the two results to obtain a new value where the two n/2 bit values have been swapped.

This works all the way down until the final processing of two bit values and swapping those over actually reverses the bit in the pairs of bits, giving the final result.

4.5 Reversing 32 Bit Values

You should be able to work out the bit shifts and masks required to swap around the two 16 bit values in a 32 bit value? If you said “Yes, use the SWAP instruction” you would be correct - there is no need to do the *mask and shift dance* to swap them over, we already have a single instruction to do exactly that!

It’s time for some code. Listing 4.1 is the comments at the head of the file which explains how to call the demo code from SuperBASIC or Assembly, and how to extract the result.

```
; A small function to reverse the bits in a long word.
; So, 1111 1111 0000 0000 1100 1100 1010 1010 will become
;     0101 0101 0011 0011 0000 0000 1111 1111
;
; Norman Dunbar
; June 25 2020.
;
; Call this code from SuperBASIC as follows:
;
; CALL address, value
; PRINT bin$(PEEK_L(address + 2), 32)
;
; Where 'address' is the address of the label 'entry'.
;
; To use the code in Assembly:
;
; Call 'reverse32Bits' with D0.L as the value to reverse.
; The code exits with the reversed bits in D0.L.
;
```

Listing 4.1: Reverse32_asm - Header Comments.

Listing 4.2 is the SuperBASIC code entry point. The code should be CALLed and passed a single value to be bit reversed.

```
entry
    bra.s    start

saveD0
    dc.l     1
```

```

start
    move.l  d1,d0
    bsr.s   reverse32Bits
    lea     saveD0,a3
    move.l  d0,(a3)
    moveq   #0,d0
    rts
```

Listing 4.2: Reverse32_asm - SuperBASIC Entry Point.

As you cannot pass a value to register D0 from SuperBASIC, the value in D1.L is copied into D0.L and the bit reversal code in Listing 4.3 is called to reverse the bits. The result is extracted from D0.L and stored in the long word at label saveD0 from where it can be PEEK_L'd by SuperBASIC to retrieve the reversed bit value.

The code for the actual bit reversal is shown in Listing 4.3. This routine starts by saving all the working registers and testing D0 for zero. Zero is already "reversed" so we bale out early if this special case is detected.

If we intend to carry on, the table of mask values is assigned to A0 and we start by swapping over the two 16 bit values in the 32 bit register. That's the simple bit out of the way. The masks we have in the table are those we will use to swap over 16 bit values, then 8, then 4 and finally the 16 pairs of two bit values.

Register D4 holds the number of shifts we need to do at each step in the process.

```
reverse32Bits
    movem.l d1-d4/a0,-(a7)  ; Save the workers
    tst.l   d0              ; Zero?
    beq.s   reverseDone     ; Yes, done
    lea     maskTable,a0    ; Mask table
    swap    d0              ; The easy 16 bits are swapped...
    moveq   #8,d4           ; Shift counter

reverseLoop
    move.l  (a0)+,d1        ; Get first/next mask
    beq.s   reverseDone     ; Finished
    move.l  d1,d2           ; Copy mask
    not.l   d2              ; Invert mask copy
    move.l  d0,d3           ; Copy value
    and.l   d1,d0           ; Mask
    and.l   d2,d3           ; Inverted mask
    lsr.l   d4,d0           ; Shift top down
    lsl.l   d4,d3           ; Shift bottom up
    or.l    d3,d0           ; Combine the bits
    lsr.b   #1,d4           ; Reduce shift count
    bra.s   reverseLoop     ; And again

reverseDone
    movem.l (a7)+,d1-d4/a0  ; Restore workers
    rts

maskTable
    dc.l    $FF00FF00       ; 1111111100000000 1111111100000000
    dc.l    $F0F0F0F0       ; 1111000011110000 1111000011110000
    dc.l    $CCCCCCCC       ; 1100110011001100 1100110011001100
    dc.l    $AAAAAAAA        ; 1010101010101010 1010101010101010
    dc.l    0
```

Listing 4.3: Reverse32_asm - Reverse32Bits Routine.

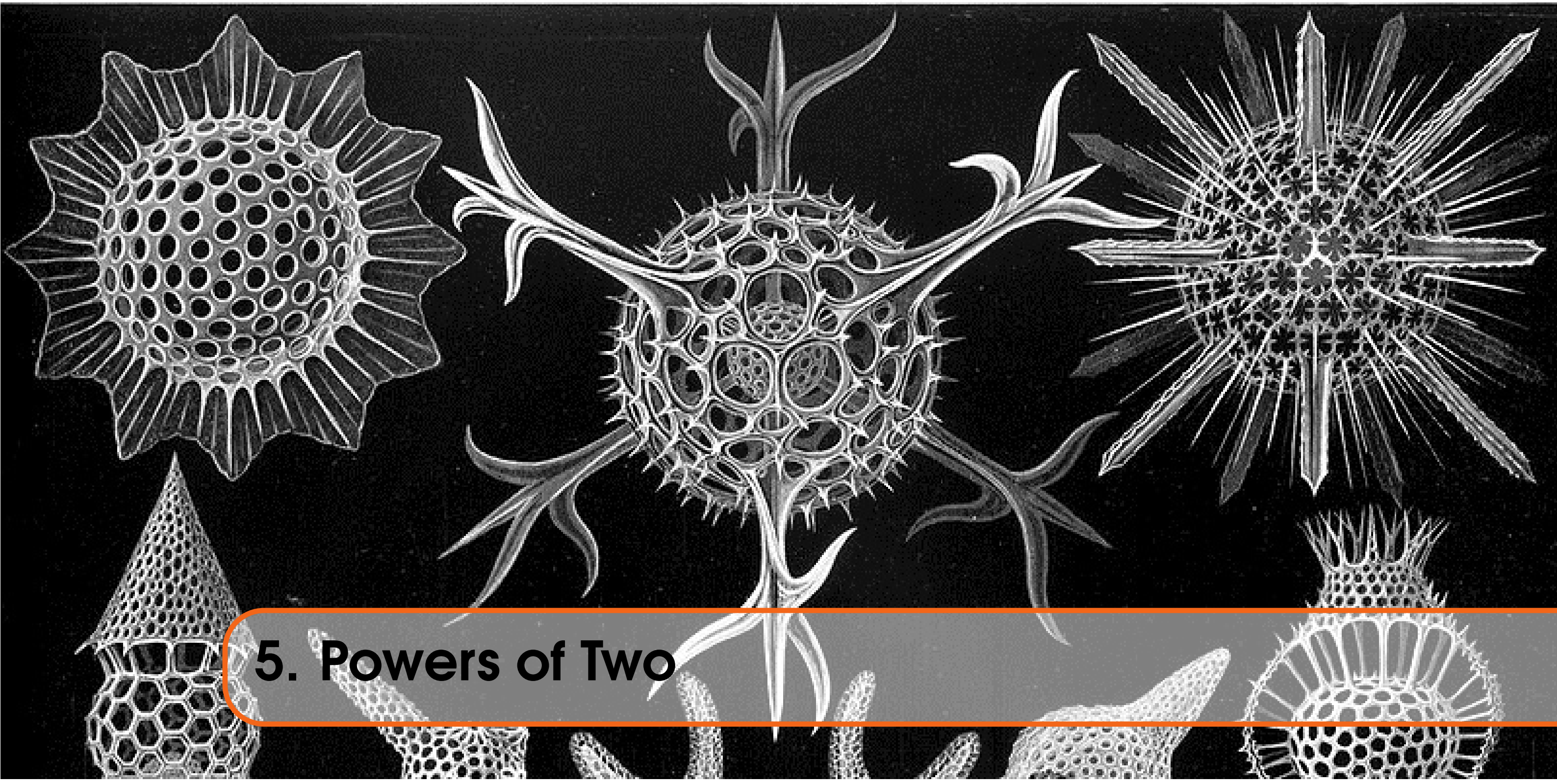
The code at reverseLoop does all the hard work. On entry A0.L points at the first mask we need, so that is loaded into D1.L and copied immediately to register D2.L where the bits are inverted to give the second mask we need. The maskTable only stores one of each pair of mask values. If the mask value is zero, we are done and we exit the loop.

The value to be reversed is copied into D3.L and D0.L is AND.L'd with the mask in D1.L. That gives the first result, prior to the shifts. D3.L is AND.L'd with the inverted mask in D2.L which gives the second result, prior to the shifts.

Both registers are then shifted in the appropriate direction by the number of bits in D4.B before the value in D3.L is OR.L'd into D0.L. All that remains is to divide the shift count in D4.B by two before we jump back into the top of the loop.

When we are all done reversing the bits in D0.L, we restore the working registers and return to the caller with the reversed bits in register D0.L.

The table at maskTable holds 4 masks which are used when swapping over the two n/2 bit values in an n bit value. As you can see only the mask for the first AND.L instruction is stored. This is because the mask used in the second AND.L instruction is the inverted value, and the NOT.L instruction will give us that mask.



5. Powers of Two

Some more messing about with a bit of code I'm writing for my Arduino required a given number to be adjusted to the next power of two, unless that number was already a power of two. So, the value 6 is not a power of two and would result in a new value of 8, while 4 is already a power of two and thus, would not be changed.

I managed to get this task accomplished – it was for a circular buffer which can be set up at any size, but the size must be a power of two, and fit into 8 bits, unsigned – in case you were wondering!

As I'm a bit short on ideas for stuff to write about for this eComic, I wondered how easy it would be to convert a few hundred bytes of C++ code into Assembly Language? With a 68020 processor, or QPC2, and George's GWASS assembler, it was rather simple, and took far less bytes than on my Arduino! It was a little more difficult with a 68008 and GWASL though.

5.1 The Algorithm

The way to determine the next power of 2 value for a number is reasonably simple, but there's a catch, a number might already be a power of 2. This is "easy" to determine as there will be a single set bit in the number, so we could count the set bits to determine if the number is already a power, and return it if so. Too difficult!

- Subtract 1 from the number;
- Find the most significant set bit;
- Work out a value for a number with just that bit set;
- Return double the number.

5.1.1 How it Works

Ok, we know what to do, how does it work? And why subtract 1 at the start? Let us assume 8 bit values, for simplicity, and to stop me typing 32 ones or zeros across the page!

If we take an example of the value 65, this has the binary value 0100 0001. The highest set bit is bit 6 for a value of 64. But as there are other bits set in the number, 65 is obviously greater than 64. The next power of 2 greater than 65 is 128. Even though we didn't do the required subtraction, we would correctly return $2*64$, or 128.

If, on the other foot, the value we started with was 64, it has a binary value of 0100 0000. Returning $2*64$ would be 128, again, but this would be incorrect as 64 is already a power of 2, so the correct answer should be 64.

So, adding in the subtraction this time, we start with 64 – 0100 0000 – and subtract 1 to give 63, or 0011 1111. The highest bit set here is bit 5, for a value of 32. Returning $2*32$ is indeed 64. But does that work with a higher value?

Taking 65 again, we still have a binary value of 0100 0001. When we subtract 1 we get 64 – 0100 0000 – returning $2*64$ does indeed still give the correct result of 128.

The algorithm works. Ok, what about zero? Does that end case work?

Subtracting 1 from zero gives -1, or 1111 1111. The most significant bit set is bit 7 or 128. Returning $2*128$ would be 256, which has the lower 8 bits clear, or zero. The closest 8 bit power of 2 to zero is actually zero. This is incorrect as the closest power of 2 to zero is 2^0 or 1. Hmmm.

In my C++ code, I tested for this corner case, and simply returned zero. However, in the code in Listing 5.1, it actually doesn't need a corner case check as passing zero does correctly result in 1 being returned. Spooky!

5.2 Easy Version for 68020

The code in Listing 5.1 is the entire routine. It is a massive 38 bytes long.

```
; This code finds the value of the "Next Power of Two" for any
; given number.
;
; Call here with one (long) parameter.
; PRINT PEEK_L(start + 2) for the result.

start   bra.s   doit
result  ds.l    1

doit    lea result,a1          ; Result address
        move.l d1,d0           ; Passed parameter
        subq.l #1,d0           ; D0 might be a power of 2
        bfffo d0{0:32},d1      ; Find first 1 bit

; If we find a set bit, D1 has the "offset". Bit 31 = offset 0,
; bit 30 = offset 1 and so on. The bits are numbered from the
; MSB which is not the normal manner. To convert, subtract the
; offset from 31 to get the required bit number.

        neg.l d1               ; D1 = -D1
        add.l #31,d1           ; Same as subtracting!
        addq.l #1,d1           ; Just because!
        moveq #0,d2            ; For the result
        bset d1,d2             ; Set the result bit.
        move.l d2,(a1)         ; Save the result
```

```
done
        clr.l  d0
        rts
```

Listing 5.1: MC60020 - Power2_asm

The value we pass in will end up in register D1. For some reason, I copy that into D0 (I forget why I did that!) but I could have saved a couple of bytes here and there by leaving it alone! Silly me.

Anyway, the next step is to subtract 1 from D0 and then look for the most significant set bit. On the 68020 we have the ability to use bit fields, so that's what the `BFFFO D0{0:32},D1` instruction does, it stands for *Bit Fields Find First One*. It looks in D0, starting at *offset* 0 for 32 bits, for the first set 1 bit. If there are no set bits, the Z flag will be set, and D1 will take on the bit field width, or 32, as it's value.

If there is a set bit, its *offset* will be placed in D1, however, the offset is not the actual bit number. The offset, as the comments indicate, is counted from bit 31 down towards bit 0. Normally we count bits from the least significant end but not in a bit field, they count from the most significant end. Confusing or what. We can easily convert an offset into a bit number simply by subtracting it from 31.

We subtract D1 from 31 in the roundabout way of negating D1 and adding 31 to it as $-D1+31 = 31-D1$.

5.3 Hard Version for 68008

That was the easy case, when using the 68020 processor's useful `BFFFO` instruction, what about the original QL's 68008 processor - it doesn't have this instruction?

Ok, going back to the examples above with 64 – a power of 2 already – first. If we AND a value with the value minus 1, and keep going until we get a zero answer, we have detected the leftmost set bit. For example:

- Value = ??
- Value = Value - 1 (in case it's already a power of 2)
- Repeat loop
- If (value and (value - 1)) = 0, return value * 2
- Else value = (value & (value - 1))
- End repeat loop

For the initial value of 64, 0100 0000, we have:

```
64 - 1 = 63
63  = 0011 1111
62  = 0011 1110
AND = 0011 1110 = 62
61  = 0011 1101
AND = 0011 1100 = 60
59  = 0011 1011
AND = 0011 1000 = 56
55  = 0011 0111
AND = 0011 0000 = 48
47  = 0010 1111
AND = 0010 0000 = 32
31  = 0001 1111
```

```
AND = 0000 0000 = 0.
```

As 32 was the current value when we got zero, we return 64, which is the next power of two to 64. Return $2 * 32 = 64$.

If you look at the binary values above, you will see that we delete one of the lower significant 1s each time we AND with $(value - 1)$. When only a single 1 bit remains, the highest, we are done.

Continuing with the above examples, let's now do 65.

```
65 - 1 = 64
64  = 0100 0000
63  = 0011 1111
AND = 0000 0000 = 0.
```

As before, the current value was 64 when we got a zero from the AND operation, so we exit and return the result of 128. That was quick!

Looking good, what about 1?

```
1 - 1 = 0
0  = 0000 0000
-1 = 1111 1111
AND = 0000 0000 = 0.
```

In this example, the value when we hit zero was zero, so returning $2 * 0$ is *not* the correct answer!

It appears that 1 is a special case which the code in Listing 5.2 must check for at the start. This code assembles to a massive 44 bytes – slightly larger than the 68020 code in Listing 5.1.

```
; This code finds the value of the "Next Power of Two" for any
; given number. The first few results are:
;
; Call here with one (long) parameter.
; PRINT PEEK_L(start + 2) for the result.

start   bra.s   doit
result  ds.l    1

doit    lea result,a1         ; Result address

; Special case. If D1 is 1, we expect 2 as the result. But
; we actually get 0. This is because ANDing D0 with 1-1 = 0.

        move.l d1,d0          ; Passed parameter
        cmpi.l #1,d0          ; Was it 1?
        beq.s done            ; Yes, return result (2)

setup
        subq.l #1,d0          ; D0 might be a power of 2
        move.l d0,d2          ; TEMP is D2

loop    move.l d0,d1          ; D1 = D0
```

```
        subq.l  #1,d1          ; D1 = (D0 - 1)
        and.l  d1,d2           ; TEMP = D0 & (D0 - 1)
        beq.s  done            ; Zero = no more set bits.
        move.l  d2,d0          ; D0 = TEMP
        bne.s  loop            ; Not done yet.

done
        lsl.l  #1,d0           ; D0 = 2 * D0
        move.l  d0,(a1)        ; Save the result
        clr.l  d0
        rts
```

Listing 5.2: MC68008 - Power2_asm

In the code, the comments show the algorithm in use for any non-special values – basically, anything that isn't 1 – and uses D2 as the TEMP register, D0 is Value and D1 is Value - 1.

D0 is loaded from D1 and has 1 subtracted in case it is already a power of 2. It is then copied into D2 ready for the main loop. In the loop, D0 is again copied, this time over to D1, and has 1 subtracted. This is ANDed with D2 and if the result is zero, we exit the loop and return whatever is in D0 * 2.

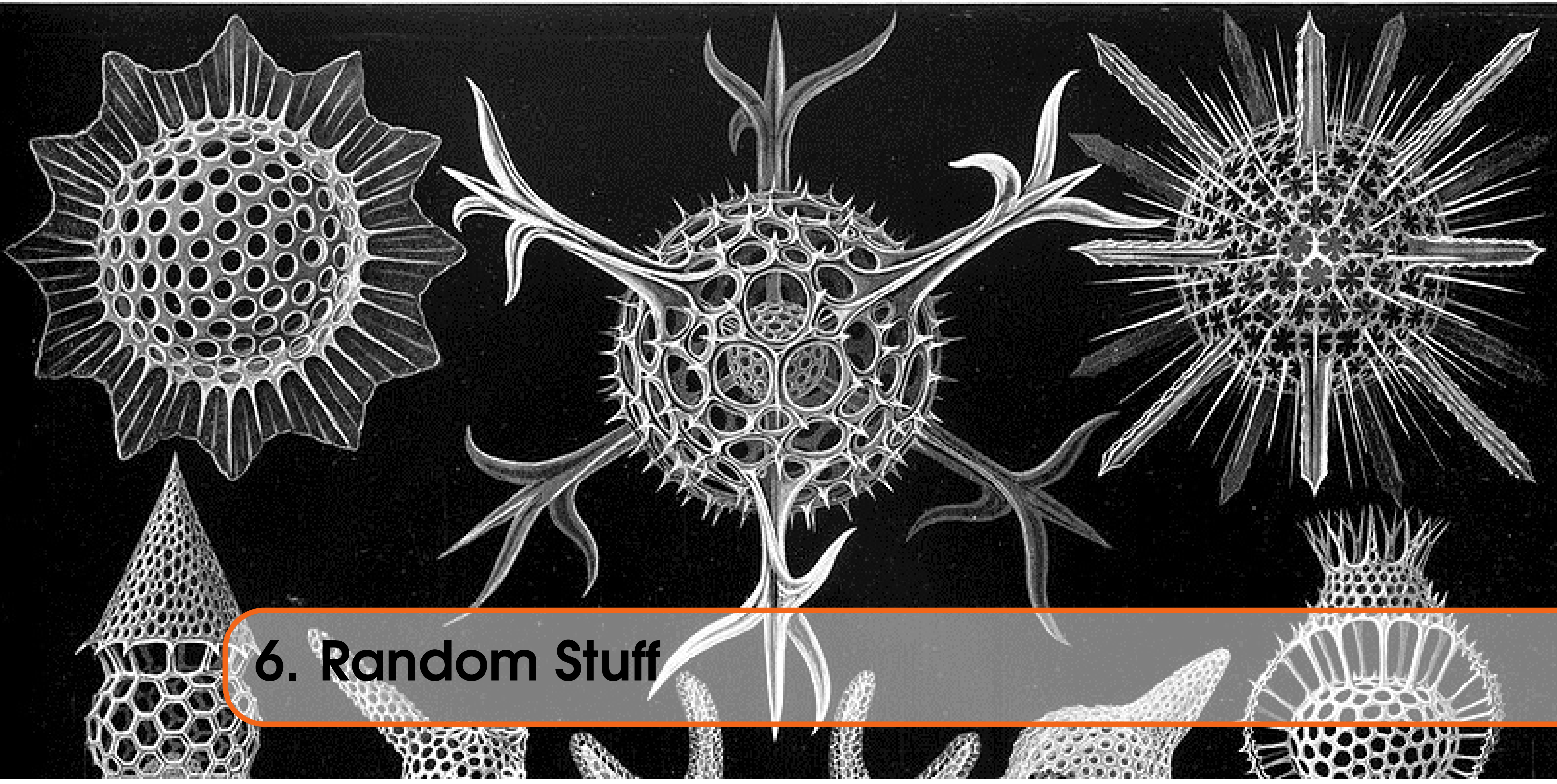
If the result is not yet zero, we copy D2 into D0 as the new value, and try again from the top of the loop. Eventually, we will get a zero result and will bale out with a value to return.

If the value passed was 1, then we copy that into D0 as normal, and test for the special case. If we find it, we skip over the main processing and return $1*2$, which is the correct result.

Make a Procedure?

It shouldn't be too hard to convert one of the two listings above into a working SuperBASIC function:

- Fetch one parameter as a long integer into D1;
- Call the code to do the working out, but grab D0 at the end as opposed to storing it;
- Allocate 2 extra bytes of maths stack for the result – there's 4 on there already;
- Convert D0.L to a float and save on the maths stack;
- Set D3 for a float;
- Clear D0;
- Done.



6. Random Stuff

Ever needed some randomisation in your assembly code? No, neither have I until recently when I suddenly had a need to generate random numbers from 1 to 6 inclusive. How is this possible, given that there are no apparent vectors or traps to grab hold of a random number?

I could have cheated and had a look through my copies of 68000 coding books – but that would have been in breach of copyright, probably, and best avoided. So I did the next best thing, I stole some code from the SMSQ sources!

Almost nothing in the following is my own work, I have stolen it, and only slightly modified it to suit what I needed it for. It does work though, I'm happily generating numbers from 1 to 6 inclusive, and no, it is not a dice (die) that I'm emulating even if the same numbers are involved!

6.1 Random Seed

The code I was working on is to be used in a job, so I have a storage location for my random seed which is an offset from the A6 register. The following code takes that into consideration.

```
; In job code, this is an offset from the A6 register.

myRandSeed   equu     0

; The job code starts here
start        bra.s    myCode
             dc.l     0
             dc.w     $4afb

fname
     dc.w     fname_e-fname-2
     dc.b     "My Job's Name"

fname_e
```

```
    equ     *

myCode
    adda.l  a4,a6                ; A6 points to our data
    clr.l   d1                   ; Randomise the date
    bsr     randomise            ; Do it
    ...                          ; Do stuff
    bsr     rnd                  ; Generate a word 1 to 6 inclusive
    ...                          ; Do more stuff
```

Listing 6.1: The random seed

The purpose of the job code is not relevant here, it will become apparent, I hope, when I get it finished – either in this edition or a future one. Coming soon[1] and all that!

I could have used the system random seed for my own numbers, but I thought about it and didn't want to mess up any other programs that could be running but which depend on a certain set of random numbers based on a starting random seed. Unlikely, perhaps, but I decided to avoid the problem.

6.2 Randomisation

Now that we have the seed variable, we will need a manner of initialising it with a new value. SuperBASIC does this by taking the clock's value in seconds if we don't supply a value ourselves, so that's good enough for me too. You will note from the comments that this code is stolen and only amended in a minor manner for my own needs.

```
;-----------------------------------------------------------------------
; This is effectively ramdomise(date). The code is exactly as
; per the SBasic RANDOMISE function. I stole the code!
; (sbsext/ext/rnd.asm)
;
; Enter with D1.L = 0 to randomise(date) or with D1.L = some
; specified value to randomise(D1).
;
; Preserves all registers.
;-----------------------------------------------------------------------
randomise
    movem.l d0-d2/a0,-(a7)       ; Save workers
    tst.l   d1                   ; D1 passed with a value?
    bne.s   randomise_d1         ; Yes, skip the date
    moveq   #mt_rclck,d0         ; Read clock into D1
    trap    #1                   ; No errors, no need to
;                                ; call doTrap1.

randomise_d1
    move.l  d1,d2                ; D1 = D2 = HHHH LLLL
    swap    d1                   ; D1 = LLLL HHHH
    add.l   d2,d1                ; D1 = (LLLL+HHHH) (HHHH+LLLL)
    move.l  d1,myRandSeed(a6)    ; Save random seed
    movem.l (a7)+,d0-d2/a0       ; restore workers
    rts
```

Listing 6.2: Randomise function

[1] Given my record, for certain values of "soon"!

The code should call the randomise entry point either with D1.L holding zero, or the required starting value for our seed. If we passed zero in D1, then the current date, in seconds, is read from the system and used as a starting point.

Arriving a label randomise_d1, we have a non-zero value in D1.L and can use it to randomise the system. The value is copied into D2.L for safety, then D1 is swapped over to put the low word into the high word and vice versa. If we started with D1.L holding $12345678 we end up with $56781234. This value is then added to the original seed value in D2.L to give, in this example, $68AC68AC.

Whenever randomise is called, we end up with a new random seed which has the high and low words identical. However, as soon as we begin using the seed, that changes. Read on.

6.3 Random Generation

Before we continue, can I just point out that I am by no means a mathematician and while I can look at what the code is doing, I have no idea what formula it is using to do it!

The first problem is to generate a random number from the random seed and to update the seed. We need to do this to avoid generating the same value over and over again!

```
;------------------------------------------------------------------------
; This is effectively RND(1 TO 6). The code does exactly as
; per the SuperBASIC RND() function. I stole the code!
; (sbsext/ext/rnd.asm)
;
; D1 = Bottom of range = 1.
; D2 = Top of range + 1 = 7.
;
; Returns D1.W as RND(1 to 6) and obviously trashes D1.
;------------------------------------------------------------------------
rnd
        movem.l  d0/d2/d4,-(a7)       ; Save workers
        move.l   myRandSeed(a6),d0    ; Get seed value
        move.w   d0,d4                ; Copy low word ???? LLLL
        swap     d0                   ; LLLL HHHH
        mulu     #$c12d,d0            ; D0.L = HHHH * 49453
        mulu     #$712d,d4            ; D4.L = LLLL * 28973
        swap     d0                   ; HHHH LLLL
        clr.w    d0                   ; HHHH 0000
        add.l    d0,d4                ; I have no idea!
        addq.l   #1,d4                ; New seed
        move.l   d4,myRandSeed(a6)    ; Save seed
```

Listing 6.3: Rnd 1 to 6 function - Part 1

After saving the registers we will be working with, except D1 which we use to return the random number later, we grab hold of the random seed and start messing about with it to generate a new seed.

The high word of the seed, in D0, is multiplied by 49,453 and the low word, in D4, by 28,973 neither of which are prime. Both are divisible by 7 if you don't fancy working it out! The resulting long word is D0 is swapped and added to D4 before D4 is incremented. This is our new random seed and is saved accordingly ready for the next call to the rnd routine.

If you want to work through an example:

The original seed, \$68AC68AC, is copied to D0 and D4. After swapping D0, which has no real effect on the first call after a call to randomise, we end up with D0.L = \$68AC68AC and D4.L = \$xxxx68AC. We are going to multiply so the high word of D4 is of no interest.

After the two multiplications, we now have D0.L = \$68AC * \$C12D = \$4EFC123C and D4.L = \$68AC * \$712D = \$2E46523C.

Swapping D0 and clearing the low word gives \$123C0000 which we then add to D4 to get \$4082523C which is then incremented to \$4082523D and used as the next random seed.

After all that, we are now, *finally*, ready to generate the random integer between 1 and 6 that we want.

The code below is designed[2] to only generate a random number between 1 and 6, inclusive, and these two values are hard coded into registers D1 and D2. Should you wish to generalise the following code to pass your own ranges, it should be quite simple.

```
rndOneToSix
        moveq     #1,d1                     ; Bottom of range
        moveq     #6+1,d2                   ; Top of range + 1
        sub.w     d1,d2                     ; Size of range
```

Listing 6.4: Rnd 1 to 6 function - Part 2

First we work out the range of values that we need. This is $(D2+1)-D1$ which in this case, always works out as $(6+1)-1$ giving a range of 6. At this point we have a random

long integer in D4 and a range in D2.

```
        swap      d4                        ; LLLL HHHH of seed
        mulu      d2,d4                     ; D4.HHHH * top
        swap      d4                        ; Take top word
        add.w     d4,d1                     ; Add range bottom
;                                           ; D1 = RND(1 to 6)
        movem.l   (a7)+,d0/d2/d4            ; Restore workers.
        rts
```

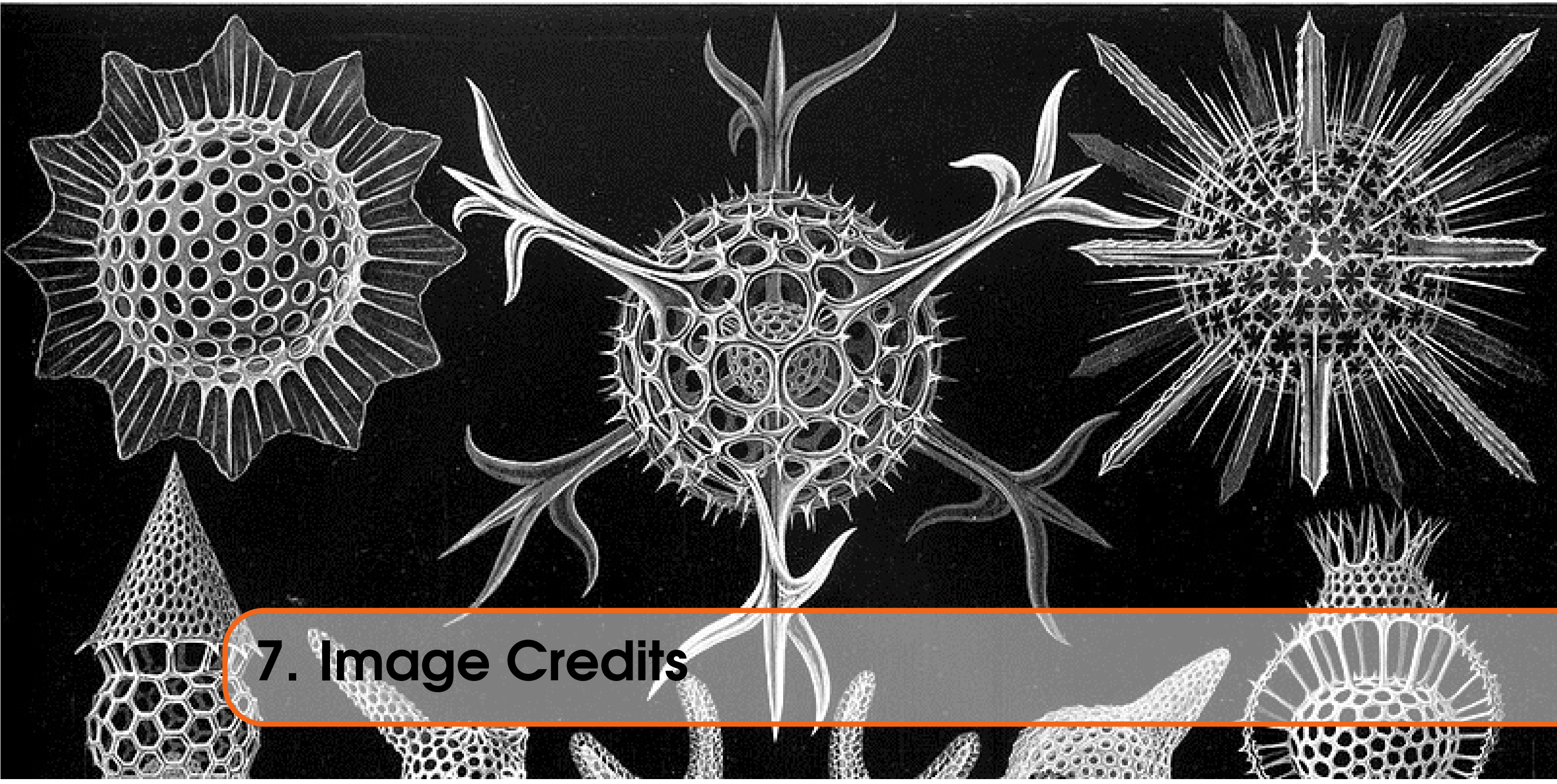
Listing 6.5: Rnd 1 to 6 function - Part 3

Now, for some unknown reason, we swap D4, our new random number and seed, and multiply the previous high word by the range of values we are looking for, and swap it back again. After this, we add the bottom value of the range to the low word of D4 and this is our value between 1 and 6. We then restore the working registers and return the value in D1.W.

Continuing the worked example from previously, D4 starts off as \$2E46523D and we multiply the high word, \$2E46, by the range, 6, to get \$000115A4. After swapping D4 back to get \$15A40001, we add on the base of the range, 1, to get our actual random number, \$0002 in this case.

Any mathematicians out there fancy writing up an explanation of *exactly* what the hell is going on there?

[2]For certain values of “designed”!



7. 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.