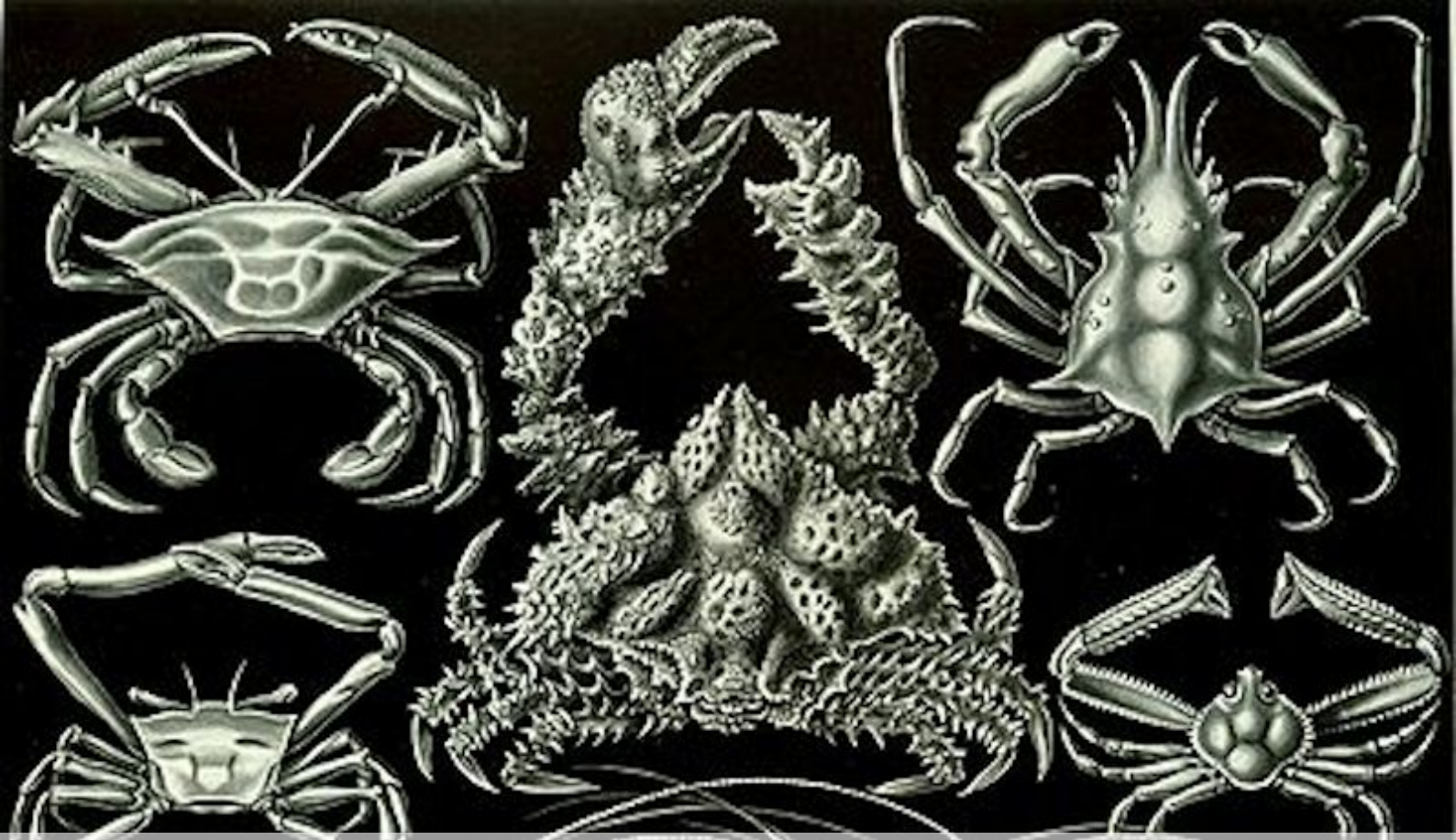


QL Assembly Language Mailing List

Issue 12

Norman Dunbar

PUBLISHED BY MEMYSELFEYE PUBLISHING ;-)

Download from:
https://github.com/NormanDunbar/QLAssemblyLanguageMagazine/releases/tag/Issue_11

Licence:
Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the "License"). You may not use this file except in compliance with the License. You may obtain a copy of the License at http://creativecommons.org/licenses/by-nc/3.0. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

This pdf document was created on *30/6/2022* at *14:45:46*.

Copyright ©2022 Norman Dunbar

Contents

1 Preface 7

1.1 Feedback 7

1.2 Subscribing to The Mailing List 7

1.3 Contacting The Mailing List 8

2 News 9

2.1 QDOS Companion 9

2.2 QL Advanced User Guide 9

3 Beginner's Corner 11

3.1 Introduction 11

3.2 The Numbers 11

3.3 The Flags 12

3.4 The Condition Codes 13

3.5 Condition Codes 13

3.6 Assembling the Code 14

3.6.1 With GWASS 14

3.6.2 With Qmac 14

3.6.3 Executing the Code 14
3.7 Summary 14
3.8 Tools and Manuals 15
4 Quickie Corner 17
5 Maths Stack Update 19
5.1 The Process Outline 20
5.2 The Test Functions 20
5.3 Debugging with QMON2 22
5.4 Results 23
5.5 Summary 24
6 Hash Tables 25
6.1 Oracle Databases 25
6.2 Hash Duplicates 26
6.3 Hash Functions 26
7 Image Credits 29

Listings

5.1 Test Function: FN_0 20
5.2 Test Function: FN_1 21
5.3 Returning results 21
6.1 SDBM hashing function 27
6.2 NDBM hashing function 27
6.3 Ndbm_hash_function.asm 28

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. News

2.1 QDOS Companion

Derek Stewart has scanned Andy Pennell's *QDOS Companion* book and made it available as a PDF file. Andy gave his blessing to this, in case anyone is worried about copyright. There's a thread on the QL Forum, here[1], which you can follow for updates and corrections etc.

Derek has produced a PDF file containing the book, in as near to its original, format as possible, and the source code is also available as a plain Libre Office ODF file, in case you wish to add corrections and regenerate your own PDF file.

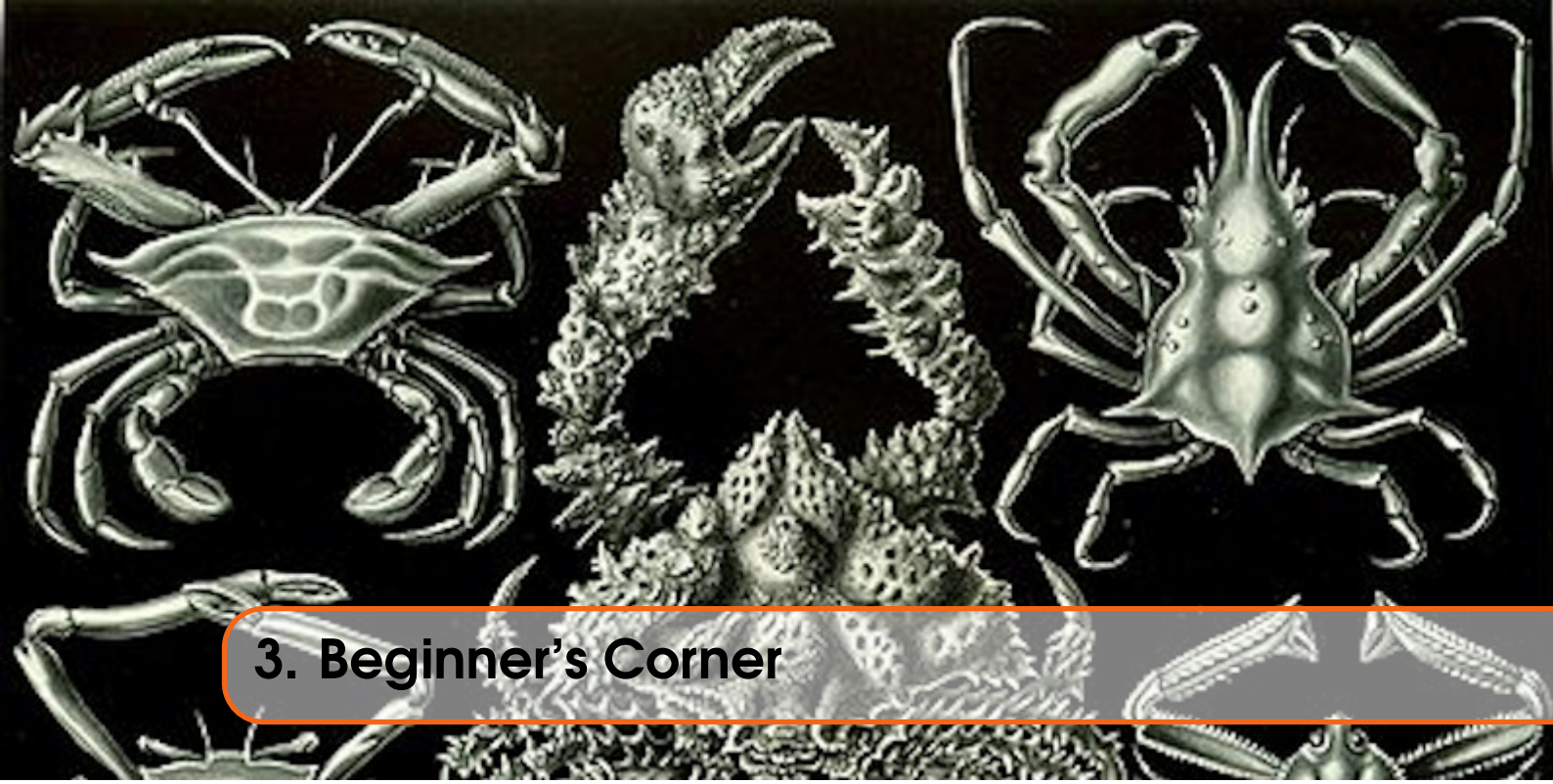
Files can be downloaded from the Sinclair QL account on GitHub, here[2]. To do so, simply click on the file you want to download, then click the "download" button on the screen that appears next. If you are after the PDF, don;t wait for it to display,. just click the button – it doesn't need to display to be downloaded.

A number of corrections have been made to the book already.

2.2 QL Advanced User Guide

Derek has also scanned Adrian Dickens' *QL Advanced User Guide* from 1984. Unfortunately, at the time of writing, Derek's requests to Adder Technologies – as Adder Publications is now known – have so far fallen on deaf ears. Derek is requesting permission to put his scans, and corrections etc, of the book into the public domain. It is not possible for Derek to do this without permission due to copyright issues. Hopefully, Derek's persistence will pay off and the book will soon be made available.

[1] https://qlforum.co.uk/viewtopic.php?f=12&t=4137
[2] https://github.com/SinclairQL/QDOS-Companion



3. Beginner's Corner

3.1 Introduction

In this issue of Beginner's Corner, we are taking a look at Condition codes and, unavoidably, signed and unsigned numbers and tests. I say 'unavoidably' as I often find myself wondering which of the condition codes are signed and which are unsigned – and it all depends on what type of numbers you are dealing with.

3.2 The Numbers

You need to know about numbers first, before we dive into the condition codes.

Question: If a register holds the byte value 255_{Dec}/FF_{Hex}/11111111_{Bin} is it signed or unsigned?

Answer: Yes!

It can be either because it actually depends on what you, the programmer, wants it to be. In many SMSQ/E trap calls, there needs to be a timeout. This is usually a positive value but if infinite timeout is requested, the value -1 is used. 255 in Decimal is the same bit pattern as -1 in Decimal. So how does the system know the difference between -1 and 255?

The leftmost bit of the number, bit 31, 15 or 7, for long, word or byte values, represents the sign of the number.

If the sign bit is a 1, the number is negative, if it is a zero, the number is positive.

This representation of signed numbers is known as 2'*s Compliment*. There is a lot of detail and explanation on Wikipedia[1] – if you are interested.

For unsigned numbers in B bits, the range of values is 0 through $2^B - 1$, while for a signed number, with the same number of bits, the range is $-\frac{2^B}{2}$ through $\frac{2^B}{2} - 1$.

[1] https://en.wikipedia.org/wiki/Two%27s_complement

From this, we can work out that for 8, 16 and 32 bit numbers, byte, word and long, we get:

- 8 bit bytes, range from 0 through 255 unsigned, −128 through 127 signed;
- 16 bit words, range from 0 through 65,535 unsigned, −32,768 through 32,767 signed;
- 32 bit longs, range from 0 through 4,293,967,295 unsigned, −2,147,483,648 through 2,147,483,647 signed.

When you look at a number in binary, it's easy to determine the decimal value simply by adding up all the powers of two where there is a 1 bit present, for example:

10100101 is $2^7 + 2^5 + 2^2 + 2^0 = 165$.

If the number is signed, then it's slightly more complicated as there are at least three different methods.

Option 1: You take the leftmost power of two and subtract all the remaining powers of two where a 1 bit is present. For example:

- 10100101 is $2^7 - \left(2^5 + 2^2 + 2^0\right) = -91$.

Option 2: Flip all the bits, convert to an unsigned value, add 1 and change the sign to make the value negative, for example:

- 10100101 flipped is 01011010;
- Converted to an unsigned value gives $2^6 + 2^4 + 2^3 + 2^1 + 1 = 91$;
- Changing the sign gives −91 as before.

Option 3: Calculate the unsigned value, and subtract from 2^B where B is the number of bits, then change the sign to make the value negative. For example:

- 10100101 is still $2^7 + 2^5 + 2^2 + 2^0 = 165$;
- The number of bits is 8 and we know 2^8 is 256;
- Subtract 165 from 256 to get 91;
- Changing the sign gives −91, yet again!

Pick the version you prefer.

Right, that's enough about numbers, signed or otherwise, for now at least!

3.3 The Flags

So, moving slightly ahead, we now need to think about the Status Register, often known as SR, where the MC680xx CPUs hold their status flags. Some of these are available to the programmer for use in code to determine the outcome of a calculation, a register load, a test and so on.

If you need more information, grab hold of my eBook[2], based on many years of writing Assembly Language articles for the, sadly defunct, *QL Today* magazine. That will explain all.

In the meantime, all you need to know at the moment is that various flags in the SR are set or cleared according to the results of some operation. If the flags are set as the result of some arithmetic operation, or by loading a value into a register, for example, then some flags indicate whether the number is signed or unsigned – in a roundabout way!

Ok, I lied, slightly, sorry. What the flags indicate is whether the number is positive, negative – amongst others – regardless of whether you intended the number to be signed or unsigned. Table

[2] https://github.com/NormanDunbar/QLAssemblyLanguageBook/releases/latest

3.1 shows the full set of flags and Table 3.2 shows what they are set to for certain conditions which can be tested.

Flag	Flag Name	Description
X	Extend	Usually the same as Carry, but used separately. (The manual is no help!)
N	Negative	Indicates if the top bit is set or clear.
Z	Zero	Indicates the last operation resulted in a zero result.
V	Overflow	The last operation caused arithmetic overflow.
C	Carry	The last operation resulted in a carry.

Table 3.1: MC680xx Status Register

3.4 The Condition Codes

Condition	Flags	Description	Signed
CC	C=0	Carry Clear	U
CS	C=1	Carry Set	U
EQ	Z=1	Equal/Zero	U and S
NE	Z=0	Not Equal/Not Zero	U and S
HI	C=0 and Z=0	Higher	U
LS	C=1 or Z=1	Lower or Same	U
PL	N=1	Plus/Positive	S
MI	N=1	Minus/Negative	S
GE	(N=1 and V=1) or (N=0 and V=0)	Greater Than or Equal	S
GT	(N=1 and V=1 and Z=0) or (N=0 and V=0 and Z=0)	Greater Than	S
LE	(Z=1) or (N=1 and V=0) or (N=0 and V=1)	Less Than or Equal	S
LT	(N=1 and V=0) or (N=0 and V=1)	Less Than	S

Table 3.2: MC680xx Condition Codes

3.5 Condition Codes

Signed and unsigned condition codes.

Bcc, DBcc, CMP. Explain.

Signed: EQ, GE, GT, LS, LT, MI, NE, PL,

Unsigned: EQ, HI, LE, NE,

Others: F, T, VC, VS,

3.6 Assembling the Code

Assembling the code requires that you have an assembler installed. See Section 3.8, Tools and Manuals for links to allow you to obtain the assemblers that I frequently use for this eMagazine.

Once you have installed an assembler, assembling the code is simple and is described in the following sections. Pick the section for your own assembler.

3.6.1 With GWASS

- `EXEC gwass60_bin` to start the assembler;
- Select option 1 to start assembling;
- Type in the filename: `ram1_??????????_asm`
- Wait.

3.6.2 With Qmac

To pass the commands directly via the command line:

- `EX qmac;"ram1_??????????_asm -data 2048 -filetype 1 -nolink -bin ram1_??????????_bin"`

Note: The above command should be typed on one line - I've had to split it for the PDF page width.

Alternatively, you can type the command interactively:

- `EX qmac`
- Type the options: `ram1_??????????_asm -data 2048 -filetype 1 -nolink -bin ram1_??????????_bin`
- Wait

What you are doing here, in both cases, is telling the assembler to:

- Assemble the source file `ram1_??????????_asm`;
- Create an executable file (`-filetype 1`), with 2,048 bytes of data space (`-data 2048`);
- Do not invoke the linker as it is not needed because everything is in the same source file (`-nolink`);
- Create the output file named `ram1_??????????_bin` – which will be in uppercase regardless of what you type here!

3.6.3 Executing the Code

After a successful assemble, and regardless of which assembler you used, `ram1_??????????_bin` will be the executable job. To run it:

- `EX ram1_??????????_bin`

On a successful execution, ...

3.7 Summary

That, hopefully, will help make writing structured Assembly Language programs more manageable. Let me know if there's anything you didn't follow or which may need more explanation, and I'll do my best to assist.

3.8 Tools and Manuals

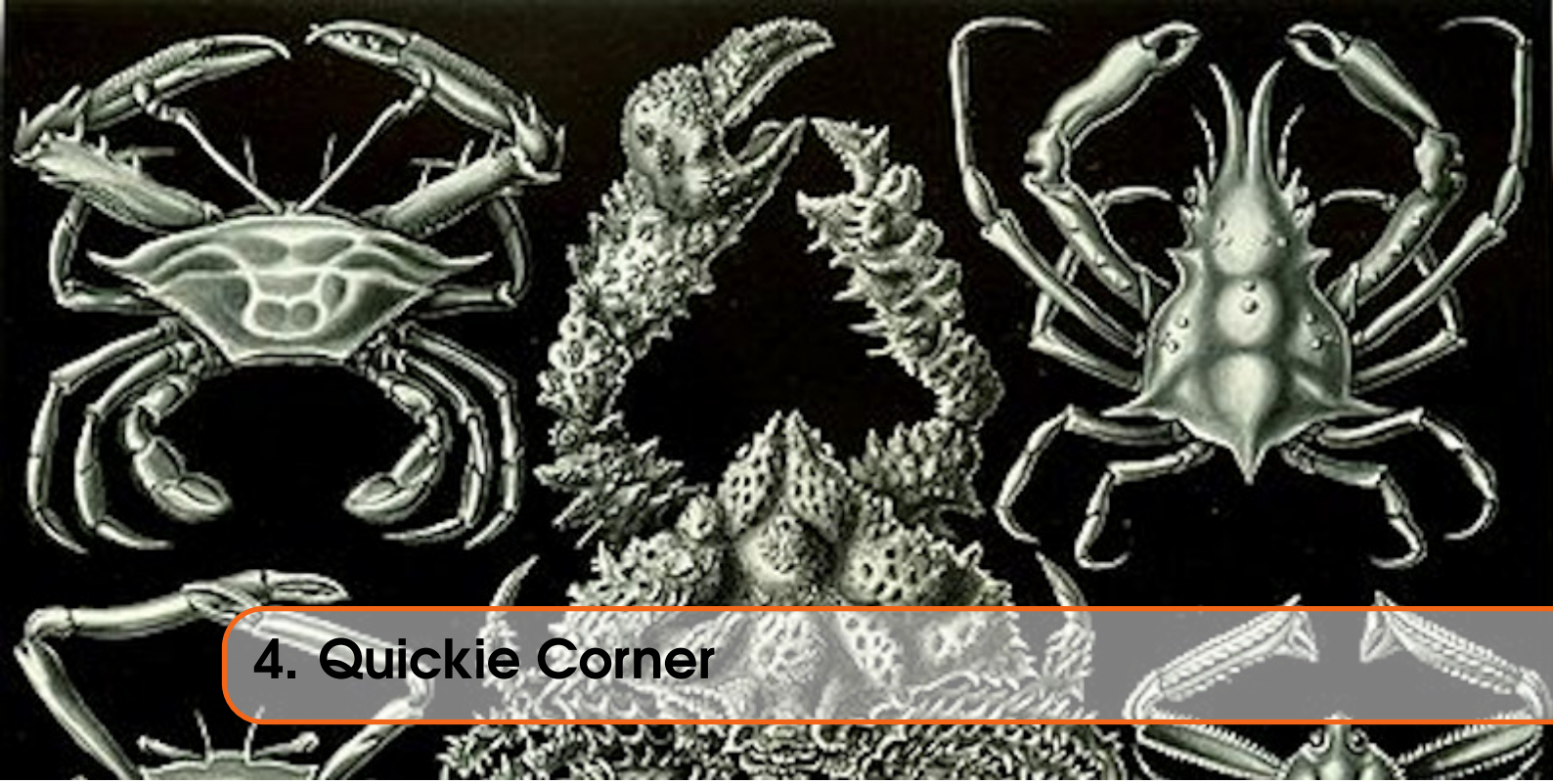
Get hold of the SMSQ/E Reference Manual from Wolfgang Lenerz's web site[3] for the official version. Alternatively, there are copies on Dilwyn's pages:

- Here[4] for the PDF for version 4.5; or
- Here[5] for the ODT (Libre Office) file for version 4.5.
- If you want to ensure that you have the most recent versions of those files, Wolfgang Lenerz's web site[6] is the place to look.

Get hold of GWASS here[7] for 68020 processors.

Download Qmac here[8] for 68008 processors.

[3] https://www.wlenerz.com/qlstuff/#qdosms
[4] http://www.dilwyn.me.uk/docs/manuals/QDOS_SMS%20Reference%20Guide%20v4.5.pdf
[5] http://www.dilwyn.me.uk/docs/manuals/QDOS_SMS%20Reference%20Guide%20v4.5.odt
[6] https://www.wlenerz.com/qlstuff/#qdosms
[7] http://gwiltprogs.info/gwassp22.zip
[8] http://www.dilwyn.me.uk/asm/gst/gstmacroquanta.zip



4. Quickie Corner

5. Maths Stack Update

Years ago I wrote an article for *QL Today* about the Maths Stack in QDOS and how the value in `A1`, on entry to a function or procedure, was *not* pointing to the top of the maths stack, as documented in the various QDOS manuals, and books around at that time. In brief, I discovered that on entry, we have three possibilities:

- If `A1` is *negative*, it means that the function/procedure in question has been called as part of an expression, and `A1` shows how much space has been used on the stack for the expression, so far. The code would resemble this:

```
PRINT 1234 * myFunction(...)
```

- If `A1` is *positive*, it means that the maths stack has some free space available for use without any requirement to allocate more.
- If `A1` is *zero*, it means that the function/procedure in question has been called on it's own, or at the beginning of an expression. `A1` shows how much space has been used on the stack for the expression, so far. The code would resemble this: The code would resemble this:

```
PRINT myFunction(...)
```

or

```
PRINT myFunction(...) * 1234
```

or any expression where `myFunction()` is at the beginning.

This is covered in my eBook which you can download from GitHub[1] – section 7.8 is the location you will need.

[1] https://github.com/NormanDunbar/QLAssemblyLanguageBook/releases/latest

There was a recent exercise by Derek Stewart to scan Andy Pennell's *QDOS Companion* book and make it available as a PDF. Andy did give his blessing to this, in case anyone is worried about copyright. There's a thread on the QL Forum, here[2], which shows the process and updates so far. See this issues News chapter for download details.

As part of the ensuing discussion, it was mentioned that Andy's book didn't correctly document A1 when entering a procedure or function. I responded with the above information and was brought before the select committee[3] to explain:

- *You don't say whether your findings apply to: JM, JS, Minerva, SMSQ/E etc. It could, in other words just be random.*
- *I'm no authority, but I would advise to not rely on this unless it were documented to be valid across the board, and to be there for a particular reason, ie as a result of the way things by necessity are done, or as a particular service to keyword writers. I wont go into reasons as I hope they are self evident.*

Which are excellent points of order.

Let's dive into the maelstrom then!

5.1 The Process Outline

In order to test I will be following these few steps:

- Create a function with no parameters, and another with one parameters, as the test code. A `TRAP #15` instruction will be placed at the very start of the function code. On normal running, this has no effect, but when I do some fiddling in the commands of QMON2, it will immediately jump into QMON2 and I can debug from there.
- When the debugger has control, examine the registers paying close attention to A1.
- The code will be tested on SMSQ/E under QPC2. It has already been tested in QDOS under the JS ROM, back in the day. I don't have Minerva or other ROMS, but I do have emulators which do!
- The functions will be tested stand-alone, and as part of an expression where it occurs at the beginning and another where it appears after the beginning.
- The functions will be tested in integer, float and string (coercion) expressions.

5.2 The Test Functions

Listing 5.1 is the code I'll be using for the test function `FN_0` and Listing 5.2 is the code for function `FN_1`. The function names match the number of parameters they take. I have not shown the code that links the functions into S*BASIC, but this is present in the code download for this issue.

There's nothing special about the two functions, other than the `TRAP #15` instructions which appear at the start of each function. This causes execution to jump into QMON2, if that has been loaded and set up correctly. `FN_0` returns a word integer of zero as it's result, while `FN_1` returns its parameter, incremented by 1, as another word integer.

```
39 ;-------------------------------------------------------------------
40 ; FN_0 takes no parameters and returns zero as an integer word
41 ; of two bytes.
```

[2] https://qlforum.co.uk/viewtopic.php?f=12&t=4137
[3] Per Witte

```
42 ;-------------------------------------------------------------------------
43 fn0
44     trap #15                    ; Jump into QMON2
45     moveq #0,d7                 ; Result in D7
46     moveq #2,d6                 ; How much space do I need?
47     bra.s fnResult              ; Just return a result
```

Listing 5.1: Test Function: FN_0

```
49 ;-------------------------------------------------------------------------
50 ; FN_1 takes one integer word parameter and returns it +1 as an
51 ; integer word of two bytes. No validation here before fetching
52 ; the parameter(s), but we do check for fetching only one.
53 ;-------------------------------------------------------------------------
54 fn1
55     trap #15                    ; Jump into QMON2
56     move.w ca_gtint,a2          ; Fetch word integers only
57     jsr (a2)                    ; Do it
58     tst.l d0                    ; Ok?
59     beq.s fn1Check              ; Yes, carry on
60     rts                         ; No, bale out
61
62 fn1Check
63     cmpi.w #1,d3                ; How many? We need 1
64     beq.s fn1Got1               ; Yes, carry on
65     moveq #err.ipar,d0          ; Bad parameter
66     rts                         ; Error out
67
68 fn1Got1
69     move.w 0(a6,a1.l),d7        ; Fetch the parameter
70     addq.w #1,d7                ; Increment for return
71     moveq #0,d6                 ; No space required
```

Listing 5.2: Test Function: FN_1

Obviously, as `FN_0` takes no parameters, we need to allocate space on the maths stack before we can return a result. We do this by fetching the current maths stack pointer from `SV_ARTHP(A6)` into `A1`, and requesting `D6.W` bytes of space for the result. In this case, `D6.W` is 2 as we are returning a word integer. After the space has been allocated, it is possible that the maths stack, plus its contents, has been moved, so we need to refresh `A1` from `SV_ARTHP(A6)`.

`FN_1`, on the other hand, takes a word integer parameter and as such, has already got enough space on the maths stack to return its result. For `FN_1` then, `D6.W` is zero to indicate that we don't need any space so we simply store the value in `D7.W` into the word pointed to by `(A6,A1.L)`.

Listing 5.3 is the code that handles the returning of results and the allocation of maths stack space, as required, in order to do so.

```
73 ;-------------------------------------------------------------------------
74 ; This code returns the function results. For FN_0 it has to
75 ; allocate two bytes but for FN_1, there's already space as we
76 ; used two bytes for the parameter. The result is in D7 and D6
77 ; holds the space we need to allocate on the stack for the
78 ; result.
79 ;-------------------------------------------------------------------------
80 fnResult
81     tst.w d6                    ; Do I need space allocated?
```

```
82      beq.s rtnFn1                    ; No, use existing space
83      move.l sv_arthp(a6),a1          ; Yes, get the stack pointer
84      move.w d6,d1                    ; Space needed for result
85      move.w qa.resri,a2              ; Allocation vector
86      jsr (a2)                        ; Allocate - will not error out
87      move.l sv_arthp(a6),a1          ; Possible new maths stack
88      subq.l #2,a1                    ; Make room for result
89
90  rtnFn1
91      move.w d7,0(a6,a1.l)            ; Stack the result
92      moveq #3,d4                     ; Signal word integer result
93      move.l a1,sv_arthp(a6)          ; Save top of stack
94      moveq #0,d0                     ; No errors
95      rts                             ; Back to S*BASIC
96
97  ;     end                           ; Uncomment for QMC assembler
```

Listing 5.3: Returning results

5.3 Debugging with QMON2

After loading the QMON2 binary, we need to tell it to execute when a `TRAP #15` instruction is executed. to do this we simply execute QMON2 in S*BASIC channel #1 – so that when it executed, it comes up in the same channel – and then tell it to intercept those `TRAP #15` instructions. This is done as follows:

```
QMON #1

Qmon> TL 14
Qmon> g
```

That's all there is to do. Now whenever a `TRAP #15` instruction is executed, QMON2 will intercept it, and break execution, giving us full control over the system at the instruction immediately after the `TRAP #15` instruction.

The `TL` command in QMON2 will execute QMON2 every time a `TRAP` *higher* than that indicated, is executed.

After setting up QMON2, testing was done by calling `PRINT` with various expressions as its parameters, each using the appropriate test function in one of three places:

- As the only term in the expression.
- As the first term in the expression.
- As the final term in the expression.

My debugging session looked something like this:

```
X=1234
X% = 1234
x$="1234"

PRINT FN_0
PRINT FN_0 * X
```

```
PRINT FN_0 * X%
PRINT FN_0 * X$

PRINT X * FN_0
PRINT X% * FN_0
PRINT X$ * FN_0

PRINT FN_1
PRINT FN_1 * X
PRINT FN_1 * X%
PRINT FN_1 * X$

PRINT X * FN_1
PRINT X% * FN_1
PRINT X$ * FN_1
```

5.4 Results

On SMSQ/E the results are slightly different from that in QDOS with the JS ROM. There was no difference when the function called was `FN_0` or `FN_1`, they both showed the same values in `A1.L` when QMON2 intercepted execution. Table 5.1 shows the results of testing on SMSQ/E.

Test	A1 (Hex)	A1 (Decimal)
Function only	0	0
Function * integer	0	0
Function * float	0	0
Function * string	0	0
Integer * function	FFFFFFFC	-4
Float * function	FFFFFFF8	-8
String * function	FFFFFFFA	-6

Table 5.1: Test function results

I repeated the tests with other arithmetic operators, the results are the same as Table 5.1 regardless of the operator in question.

Looking at the figures, it appears that on SMSQ/E, the value in `A1` at the start of a procedure or function is 2 bytes greater than that used by QDOS under the same conditions.

- When `A1` is zero, the SMSQ/E result is the same as the QDOS: JS result, no space yet used on the maths stack.
- When `A1` is negative, the SMSQ/E result appears to show an extra 2 bytes over the size of the space used on the maths stack so far. So for a word sized integer, 2 bytes, the value in `A1` is -4, for a 6 byte float it's -8 and for a string, which was never tested on QDOS: JS, it's always -6 regardless of the length of the string.
- I was unable to remember[4] how I managed to get a positive value in `A1` back in the QDOS: JS days, and I was unable to get one in SMSQ/E.

[4] I'm getting old. When I learn new stuff these days, something else has to be forgotten to make room for it!

5.5 Summary

Even without testing Minerva ROMs under QDOS, it is obvious that there are indeed differences – at least between QDOS and SMSQ/E. For this reason, it's best to ignore what I've said in the past, and simply treat `A1` on entry to a procedure or function as *not being a suitable value for the top of the maths stack* so it must always be loaded from `BV_RIP(A6)` on QDOS or from `SV_ARTHP(A6)` on SMSQ/E.

- If your function takes no parameters, `A1` *must* be loaded before attempting to request maths stack space to return the result.
- If your function takes any parameters, then until such time as you call the appropriate vector to fetch the parameters, `A1` is not usable as the maths stack top. After fetching the parameters, it will be correctly set.

6. Hash Tables

6.1 Oracle Databases

"Hang on! What does Oracle Databases have to do with hash tables", I hear you think.

Well, in the past, whenever anyone using the database tried to submit an SQL query for processing, the system used to cause a slowdown as the query had to be parsed, then a syntax and semantic validation was carried out. If all that checking worked fine, an execution plan would be constructed and in doing so, a lot of different trial execution plans would be considered until the best one was decided upon. The plan was then executed and the results returned to the user. However, the execution plan was also cached in memory so that, in the event of the same query arriving again, all that parsing and such like could be avoided, and the execution could be started *almost*[1] immediately.

The problem was, there was a latch that had to be taken to be able to run the parsing, and there was only one latch! Some badly written systems would submit similar queries very frequently, causing a lot of parsing and as more and more users attempted to submit queries, there would be a queue build up as sessions waited to take out the parsing latch.

These days things are much more different. Oracle still hashes the SQL query that was submitted – more on this soon – but it now assigns the query to one of 65,536 different slots in a hash table. There is no longer a need for a single parsing latch as we now have 65,536 slots in our hash table so we can be parsing that many different queries at the same time. Much better throughput is the result.

Unfortunately, some queries hash to the same value, and this means that they are either identical, or they are different but have the same hash value. Not good as you cannot have two queries in the one slot in the hash table. What to do?

[1] After hashing and checking the cache for any plans with the same hash value that is.

6.2 Hash Duplicates

The hash table can be configured in a number of ways to resolve the problem of duplicate hashes:

- Reject duplicates.
- Scan forward in the table for the first free slot.
- Use hash chains.

Rejecting duplicates is pretty much a non-starter. If your code can't handle duplicates, then it's not the best code to be using, *unless* there are not supposed to be any duplicates. In which case, rejecting the duplicates is a viable method of operation.

Scanning the table means that the slot that is returned from the hashing function is full, so you start looking forward in the table, with wrap around, until you find a free slot. This is ok for inserts, but with larger tables, it can slow things down an awful lot. The same goes for deleting entries – as you have to find the desired object to delete it and it might not be in the slot that you expect it to be. Furthermore, using up someone else's slot means that their attempts to store an object also has to start wandering through then table looking for a home. Not good.

So, hash chains appears to be the most acceptable option. How does it work?

This is the method used in modern versions of the Oracle Database. When a query is parsed, a hash value is created and used to select a slot in the hash table. As mentioned, there are 65,536 slots available but with potentially millions of queries being processed every second (yes, second!) duplicate hash values are bound to appear. When this happens, they are stored in their chosen slot, and if the slot is already occupied, the new entry is added and linked to the existing entry/entries in a linked list.

This is efficient for storage – the new entry always goes at the head of the list – and also for deletions as only the hash chain for this particular slot is required to be scanned, not the entire library cache. Deletions are also simple for the same reason.

6.3 Hash Functions

A hash function is simply a function, which when given some input, returns a value. The value must be the same if given the same input – it must be *deterministic* in other words. The Oracle Database uses a simply hash function, it adds up the character codes in the query that has been submitted, then does a bit of jiggery-pokery[2] in the background, before ending up with a 126 bit value indicating the hash table slot. If I remember correctly, it doesn't include spaces or tabs in the calculation. The value returned determines the slot in the hash table that this query belongs to. All that is stored in the table is an address, the address in memory for this query's memory area where all the working out etc will be carried out, and results returned from the database engine, to the user, will pass through a cursor store in this area.

Obviously, the values returned from the hash function should spread the values across all the slots in the hash table, there's no point having a table with 65,536 entries if only 10 of them get used! One way to manage this is to have a hash table with the number of slots equal to a power of two. The hash function should calculate "a number" then AND it with that power of two minus 1 which will result in a range of 'n' values where 'n' is the afore mentioned power of two and also, the table size.

Obviously, choosing a decent hash function requires a good knowledge of maths, which I don't

[2] This is a technical term.

have, so I'll come up with a hash function that will hopefully be random enough! Time will tell if I'm successful[3].

According to http://www.cse.yorku.ca/~oz/hash.html, the "SDBM" hash function is a good one, in that it was *created for sdbm (a public-domain reimplementation of ndbm) database library. it was found to do well in scrambling bits, causing better distribution of the keys and fewer splits. it also happens to be a good general hashing function with good distribution. the actual function is hash(i) = hash(i - 1) * 65599 + str[i]; what is included below is the faster version used in gawk. [there is even a faster, duff-device version] the magic constant 65599 was picked out of thin air while experimenting with different constants, and turns out to be a prime. this is one of the algorithms used in berkeley db (see sleepycat) and elsewhere.*

The C code for the sdbm hash function is shown in Listing 6.1.

```
unsigned long sdbm(unsigned char *str) {
    unsigned long hash = 0;
    int c;

    while (c = *str++)
        hash = c + (hash << 6) + (hash << 16) - hash;

    return hash;
}
```

Listing 6.1: SDBM hashing function

Effectively, this is taking a character code, c, and adding it to the existing hash multiplied by 65599 which we get from $(hash << 6) + (hash << 16) - hash$ which is, as we all know, equivalent to $(hash * 64) + (hash * 65536) - hash$ or $hash * 65599$.

Unfortunately for us in the QL World, this function will soon overflow a 32 bit register. In fact, on a QL in S*BASIC, it cannot even get all the way through hashing my name – "Norman Dunbar" before it gives up with "Arithmetic overflow" errors. It would appear that we need 64 bit integers to be able to use the sdbm hash function. However, as I mentioned, I'm not a mathematician, so lets amend it slightly and rename it to "NDBM", Listing 6.2 has the new C code.

```
unsigned long ndbm(uint8_t *str) {
    uint32_t hash = 0;
    uint8_t c;

    while (c = *str++)
        hash = hash + (c * c);

    return hash;
}
```

Listing 6.2: NDBM hashing function

Let's write some Assembly code.

We can easily start with the hash function. Mine has been saved to `ndbm_hash_function.asm`, feel free to name yours accordingly. Listing 6.3 shows the complete hash function – the file on the source code "disc" for this issue has comments describing the use and abuse of the function. Those are not shown here.

The function expects a pointer to an SMSQ string in `A1.L` on entry.

[3] That's its job after all!

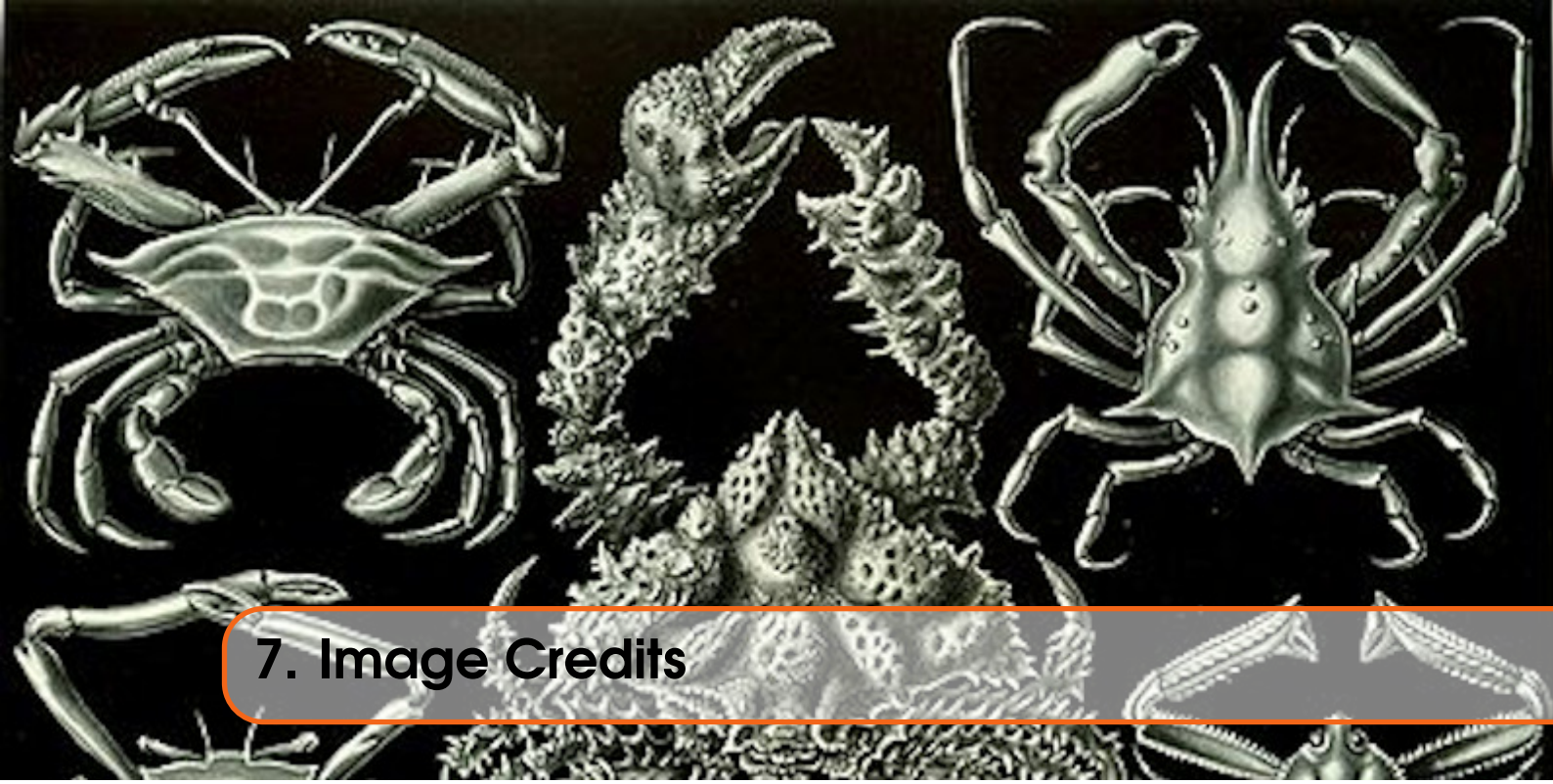
On exit, `A1.L` will be preserved, `D1.L` will be the hash value in 32 bits, `D0.L` will be zero, or `ERR.OVFL` if overflow was detected during the hash calculation.

All other registers are preserved.

Note that overflow is highly unlikely. I tested this in S*BASIC where a string is allowed to be 32,764 characters in length. I filled a string of maximum length with CHR$(255) and managed to get a hash value that didn't overflow, $7EFC87FC or 2,130,479,100.

```
err.ovfl    equ $ee             ; Overflow error code

ndbm
    movem.l d2-d3/a1,-(a7)      ; Preserve working registers
    moveq #0,d1                 ; Current hash value
    move.l d1,d3                ; A "zero" register
    move.w (a1)+,d2             ; String size, can be zero!
    bra.s ndbmEndLoop           ; Skip loop

ndbmLoop
    move.l d3,d0                ; Clear D0 again
    move.b (a1)+,d0             ; Get current character
    mulu d0,d0                  ; Multiply by itself
    add.l d0,d1                 ; Update hash
    bvs.s ndbmError             ; Oops, overflow!

ndbmEndLoop
    dbra d2,ndbmLoop            ; Go around again
    move.l d3,d0                ; No errors
    bra.s ndbmExit              ; Finished

ndbmError
    moveq #err.ovfl,d0          ; Overflow error

ndbmExit
    movem.l (a7)+,d2-d3/a1      ; Restore working registers
    rts
```

Listing 6.3: Ndbm_hash_function.asm



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 *Decapods*. The Decapoda or decapods (literally "ten-footed") are an order of crustaceans within the class Malacostraca, including many familiar groups, such as crabs, lobsters, crayfish, shrimp and prawns. Most decapods are scavengers. The order is estimated to contain nearly 15,000 species in around 2,700 genera, with around 3,300 fossil species.

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

You can read about Decapods 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!)

Decapods have absolutely nothing to do with the QL or computing in general - in fact, I suspect many of them died out before electricity was invented, and the rest probably don't care about electricity or computers! However, 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 10 legged crustaceans.