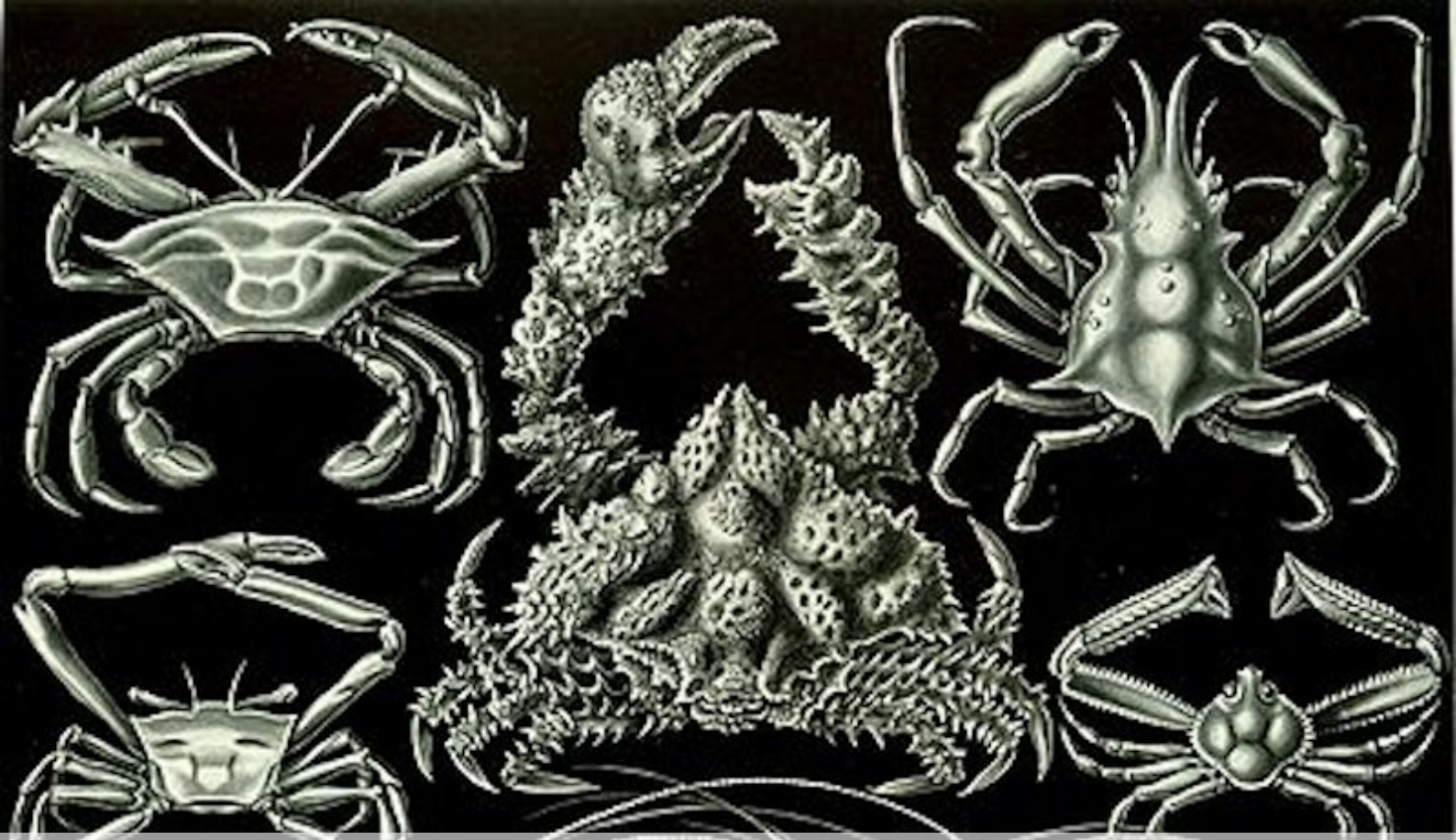


QL Assembly Language Mailing List

Issue 12

Norman Dunbar

PUBLISHED BY MEMYSELFEYE PUBLISHING ;-)

Download from:
https://github.com/NormanDunbar/QLAssemblyLanguageMagazine/releases/tag/Issue_12

This pdf document was created on *21/1/2025* at *16:55:20*.

Contents

Listings

1. Preface

1.1 Feedback

Please send all feedback to assembly@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 watching for updates on the ql-users email list, ql-users@q-v-d.com of by subscribing to the QL Forum at https://qlforum.co.uk and watching out for new messages from me about available issues.

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?

Sadly, George Gwilt has passed away and will no longer continue to keep me correct on matters where I get stuff completely wrong, as before with *QL Today*. I know George did ask, way back when I proposed this eMagazine, if there would be a contact address, 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 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 George Gwilt

Those of you who know of my old scribblings in *QL Today* and what used to be my QDOS Internals web site, will know of George Gwilt. He was an incredible programmer and knew almost everything about Assembly Language for the QL. He maintained the Turbo SuperBASIC compiler for years and assisted Marcel in getting QPC to emulate a 68020 processor. His Assemblers, *gwasl* and *gwass* are tools I have been using for years.

Sadly, in March 2024, George passed away. His son, Richard, posted a message to the QL Forum with the details. George, although he hasn't been heard of for a while, will be sadly missed by many people. I shall certainly miss his input to almost everything I ever wrote about QL Assembly Language.

RIP George.

George's son posted again with details of a hoard of QL "stuff" that they had found in a room at George's house. This was to be taken to the tip if nobody wanted it. Graeme Gregory drove to Edinburgh to rescue everything and is[1] going to catalogue everything and help with disposal/sale/-donations as appropriate.

2.2 Long Delay Since Previous Issue

You may have noticed a long delay since the previous issue? Well, My apologies for that, I've been a little busy. Some of you know that I had a book published in April 2020—*Arduino Software Internals*—well, the publishers requested that I do a second edition to cover changes in the Arduino world since 2020.

Unfortunately, they now only accept manuscripts written with Microsoft Word, or LaTeX but using

[1] Or "was" depending on when you read this!

their own document template. My book was originally written using ASCIIDoctor—a completely different source file format—so had to be converted over to LaTeX. I haven't use Microsoft Word since I was working at Morrisons Supermarket's head office some years back. Word would trash my documents on a regular basis, so I tend to avoid it whenever possible!

Arduino Software Internals 2nd Edition has now been publoished, so I'm available for other things now. However, I'm currently writing my third title—*Arduino Assembly Language*.

My second Arduino book—*Arduino Interrupts*—was published in December 2023. That was actually originally written in LaTeX—hooray!—but using a different template—boo!—so needed some work done to convert it to the correct template. That wasn't as bad as a total rewrite, but still took a while.

Luckily, my third Arduino book—*Arduino Assembly Language*—was only at the beginning the writing stage, so switching templates wasn't too much of a problem.

Unfortunately, all this work, plus the fact that we sold our house and moved back to Scotland in November 2024, has put any QL and eMagazine work on the back burner. Sorry.

2.3 Qdosmsq.dunbar-it.co.uk

A while back, Alison and I closed our IT business and "retired". I had been keeping my hosting payments going even after closing down the business, but in February 2024 I decided that enough was enough and cancelled my hosting contract. My blog, http://dunbar-it.co.uk/blog was moved to a new host on Github. It's now at http://blog.dunbar-it.co.uk[2]. Yes, I know, I'm not "https" secure, but my ISP requires paying for that and I'm a Scotsman!!!

My old http://qdosmsq.dunbar-it.co.uk website is down for the foreseeable future. I have a backup, of course, and something will return, online, "soon". If you try to access it at the moment, you will simply get a message that the site cannot be found. I haven't set anything up yet as I may need to extract the data from the Wiki format that I was using previously.

One inadvertent foul-up of cancelling my hosting plan, is the fact that I have lost my online list of subscribers to this eMagazine. I won't now be able to send out emails like I used to, when a new issue is available. I do have some backups but I need to extract the data from the backend database, which might take a while.

Also, the various contact email addresses—xxxxx@qdosmsq.dunbar-it.co.uk—that I used to use are now also gone. Any feedback on the topics in this and future issues should be directed at the email address in the Feedback section on page 7.

2.4 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[3], 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.

[2] Although, I noticed recently that some images are not displaying so I need to fix that at some stage!
[3] https://qlforum.co.uk/viewtopic.php?f=12&t=4137

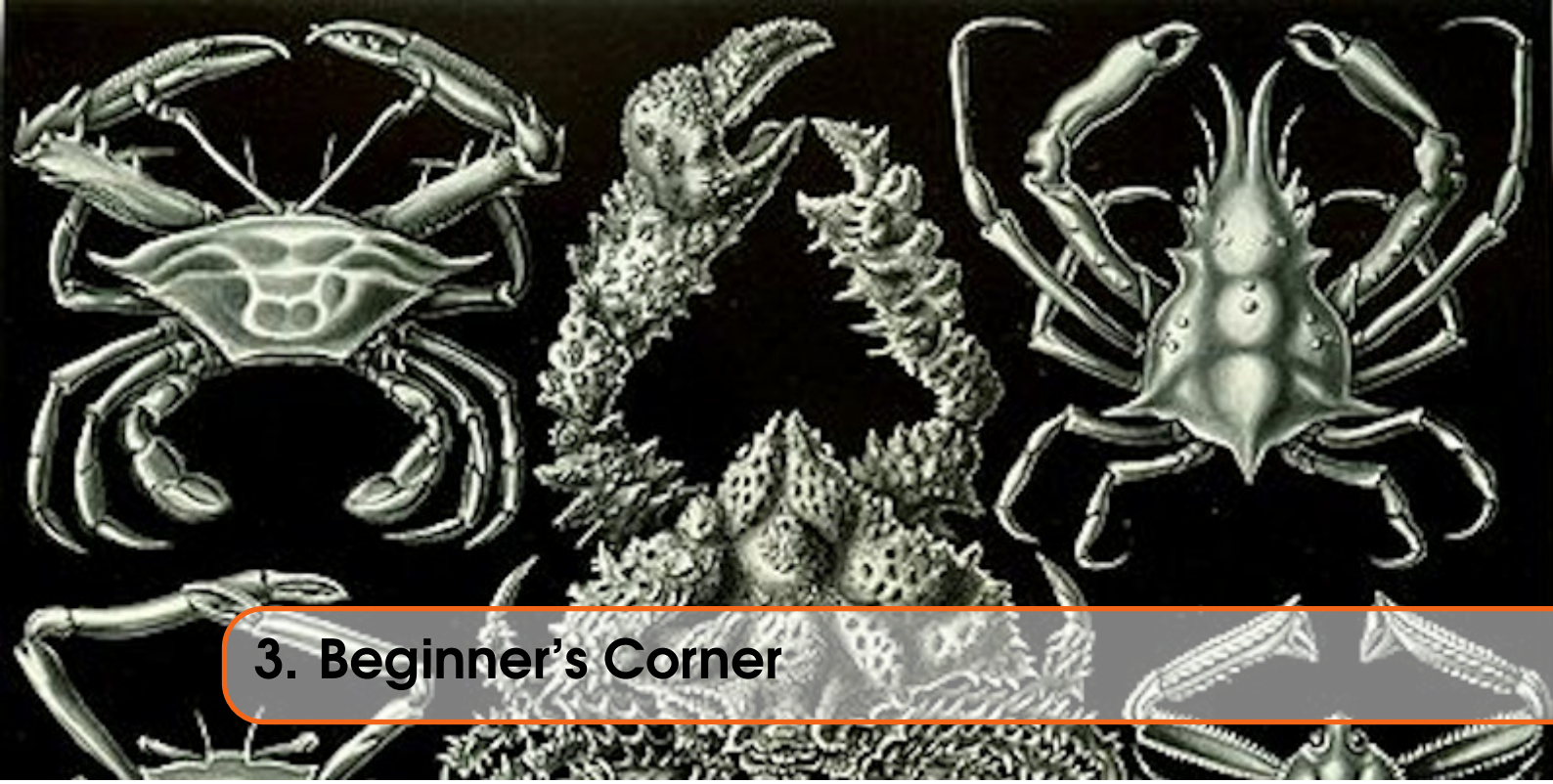
Files can be downloaded from the Sinclair QL account on GitHub, here[4]. To do so, simply click on the file you want to download, then click the "download" button on the screen that appears next. If you are after the PDF, don't wait for it to display, just click the button—it doesn't need to display in full to be downloaded.

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

2.5 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.

[4] 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 is it signed or unsigned?

Answer: Yes!

It can be either because it actually depends on what you, the programmer, want 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 bits: signed values range from 0 through 255; while unsigned range from -128 through 127.
- 16 bits: signed values range from 0 through 65,535; while unsigned range from $-32,768$ through 32,767.
- 32 bits: signed values range from 0 through 4,293,967,295; while unsigned range from $-2,147,483,648$ through 2,147,483,647.

When you look at a number in binary, it's *relatively* 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 negative of the leftmost power of two bit—bit 31; 15; or 7—and add all the remaining powers of two where a 1 bit is present. For example:

- 10100101 is $-2^7 + 2^5 + 2^2 + 2^0 = -91$.

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 and adding 1 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 2^B where B is the number of bits. For example:

- 10100101 is still $2^7 + 2^5 + 2^2 + 2^0 = 165$;
- The number of bits is eight and we know 2^8 is 256;
- Subtract 256 from 165 to get -91;

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 an 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=0	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
VC	V = 0	Overflow Clear	S
VS	V = 1	Overflow Set	S
F	N/A	False	U and S
T	N/A	True	U and S

Table 3.2: MC680xx Condition Codes

3.5 Condition Codes

You can see from Table 3.2 that some condition codes are used with signed operations while others are unsigned. It's your code and your data, so you are the one who decides which to use. If you are performing signed arithmetic, for example, then you should be using the signed condition codes; for unsigned arithmetic, the unsigned codes.

The condition codes can be used to determine if the code will branch after an operation—CMP; SUB; MOVE; etc—using the Bcc or DBcc instructions as appropriate.

The Bcc instruction simply branches if the condition codes are set as desired by the instruction.

Listings 3.1 and 3.2 show a few examples of the use of the Bcc instruction for signed and unsigned values.

```
    ; Signed value in D4.
    tst.l d4          ; Sets the flags according to value of D4.
    beq d4IsZero      ; Branch if D4 is zero. (Signed/Unsigned)
    blt d4IsMinus     ; Branch if D4 is negative. (Signed)
    bgt d4IsPlus      ; Branch if D4 is positive. (Signed)
```

Listing 3.1: Signed condition codes

```
    ; Unsigned value in D4.
    tst.l d4          ; Sets the flags according to value of D4.
    beq d4IsZero      ; Branch if D4 is zero. (Signed/Unsigned)
    bls d4IsMinus     ; Branch if D4 is negative. (Unsigned)
    bhi d4IsPlus      ; Branch if D4 is positive. (Unsigned)
```

Listing 3.2: Unsigned condition codes

You should note that while signed condition codes allow for a test for greater than; greater than or equal; equal; less than; and less than or equal, unsigned values only have higher; equal; lower or same. This means that in Listing 3.2, we had to do the check for equality first because the test for lower would include equal had we done the bls instruction first.

In the above examples, we tested the actual value in D4 but we could have tested after a CMP or SUB instruction, as per the example code in Listing 3.3.

```
    ; Signed value in D4.
    cmp.l d4,d5       ; Sets the flags according to value of D5 minus D4.
    beq d4equals      ; Branch if D4 = D5.
    blt d4IsLess      ; Branch if D4 < D5. (Signed)
    bgt d4IsMore      ; Branch if D4 > D5. (Signed)
```

Listing 3.3: Signed condition codes with CMP

The DBcc family of instructions are similar but have the advantage of decrementing a counter register and branching *until*—not *while*—the condition is true. DBEQ D0, someLabel, for example, means that register D0 will be decremented by 1 and if the Z flag is set, the branch *will not* take place. If Z is clear, the branch will be taken. Confused? I was too when I first started learning MC680xx Assembly.

If you remember that DBcc means "decrement and branch *until* 'cc' is true" then you won't go wrong.

DBF also known as DBRA (decrement and branch always), and DBT need a little extra explanation. DBF/DBRA always branch—because the condition, False, is never, ever, True. These instructions only stop branching when the counter register reaches -1.

DBT is a tad weird to say the least, it means that the branch will never be taken as the test always is true.

3.6 Assembling Code

Note: This section will remain in the Beginners Corner until the end of the universe; or until I stop writing the magazine! Not everyone is au-fait with assembling code, especially as there are quite a few assemblers for the QL.

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 win1_gwass60_bin` to start the assembler;
- Select option 1 to start assembling;
- Type in the filename: `ram1_mycode_asm` for example.
- Wait.

3.6.2 With Qmac

To pass the commands directly via the command line:

- `EX win1_qmac; "ram1_mycode_asm -data 2048 -filetype 1 -nolink -bin ram1_mycode_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 win1_qmac`
- Type the options: `ram1_mycode_asm -data 2048 -filetype 1 -nolink -bin ram1_mycode_bin`
- Wait

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

- Assemble the source file `ram1_mycode_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_mycode_bin`—which will be created with its name in uppercase regardless of what letter case you type here!

3.6.3 Executing the Code

After a successful assemble, and regardless of which assembler you used, `ram1_mycode_bin` will be the executable job. To run it, you have a choice:

- `EXEC ram1_mycode_bin`
- `EXEC_W ram1_mycode_bin`

The first will start the job running and return immediately to S*BASIC. You will never see any error codes that the job returns to S*BASIC. The second will start the job running and wait for it to complete before returning to S*BASIC. In this case, if the job returns an error code, you will see

the error message displayed in channel #0 when the job terminates.

Of course, this assumes that you have written a multitasking job in your code. If the code is `CALL`able from S*BASIC, then that's different.

3.7 Summary

Hopefully, this chapter will have gone some way in helping you understand the various tests and conditions that you might need to cater for in your code. Knowing this will also help in writing structured Assembly Language programs.

The next chapter covers this very topic.

3.8 Tools and Manuals

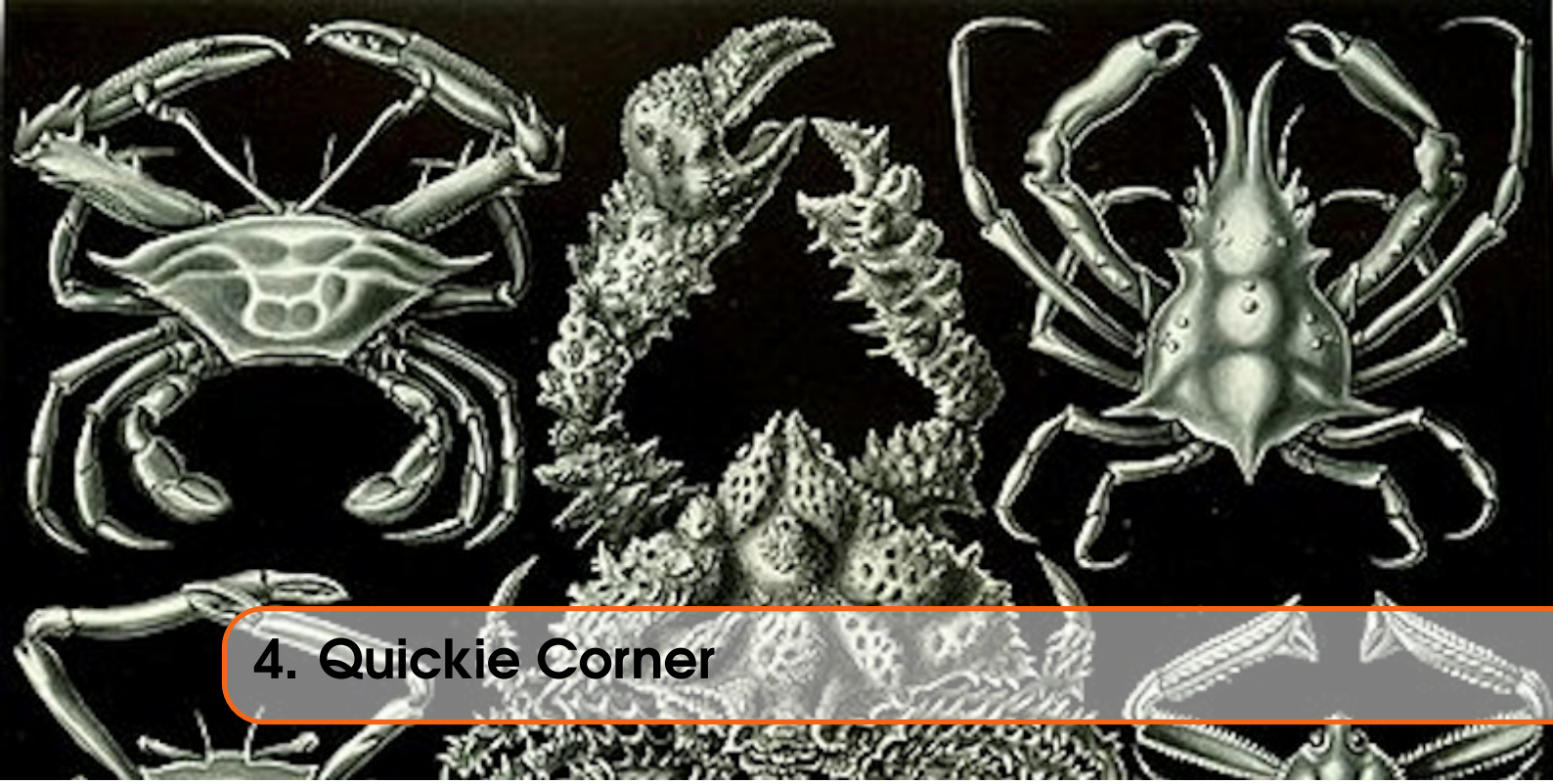
Get hold of the SMSQ/E Reference Manual from Wolfgang Lenerz's web site[3] for the official version. Alternatively, there are copies on Dilwyn's pages:

- 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. Click the link then click on the funny looking downward pointing arrow on the far right to start the download.

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] https://github.com/SinclairQL/GeorgeGwilt/blob/main/Gwass/gwassp22.zip
[8] http://www.dilwyn.me.uk/asm/gst/gstmacroquanta.zip



4. Quickie Corner

In this issue's "Quickie Corner" I shall take a brief look at program structure. What I mean by this is how to set up the various constructs much loved in structured programming. Plus some other hopefully useful stuff as well. I won't be showing any fully executable code, just the basics.

It's all very well, in SuperBASIC and S*BASIC[1], having stuff like `REPeat...END REPeat` or `IF...ELSE...END IF`, for example, but how do we do this in Assembly Language?

Some of what we will discuss is hopefully familiar from S*BASIC, but other constructs may not be. Worry not!

Read on.

4.1 Repeat Loop

The simplest of all the looping constructs. You have a loop start and a loop end. Between the two is the loop body, which is where all the work you wish to carry out repeatedly is located. The loop end simply passes program control back to the beginning of the loop.

```
repeat
    ; Do loop body code here.
    ...

endRepeat
    bra loopStart
```

Listing 4.1: REPEAT ... END REPEAT

Of course Listing 4.1 is an infinite loop; there is no "get out" clause, so once it starts it will continue until power is removed or the world ends! This sort of loop is acceptable under certain circumstances, but normally, users need an opportunity to finish things off nicely. Listings 4.2 and

[1] Henceforth referred to as simply S*BASIC.

4.3 should improve things by offering an exit clause. I'm assuming that D0 is zero if we wish to exit. Other exit conditions are available!

```
repeat
    ; Do loop body code here.
    ...

endRepeat
    ; Test for exit condition and ...
    ; continue looping until D0 = 0.
    tst.l d0
    bne repeat

exitRepeat
    ; Continue onwards from here.
    ...
```

Listing 4.2: REPEAT ... EXIT ... END REPEAT

```
repeat
    ; Do loop body code here.
    ...

endRepeat
    ; Test for exit condition and ...
    ; continue looping until D0 = 0.
    tst.l d0
    beq exitRepeat

    ; It appears we are not yet done.
    bra repeat

exitRepeat
    ; Continue onwards from here.
    ...
```

Listing 4.3: Alternative REPEAT ... EXIT ... END REPEAT

Obviously, Listing 4.2 is the better option as it is slightly shorter, however, there may be code where it is not the better option. It all depends.

The test for the exit condition can come anywhere in the loop body. You can test at the start, at the end, or part way through—it all depends of what the code is doing.

4.2 Repeat ... Until Loop

This is not a structure available on the QL, but it is available in other languages, so might be useful to know about. The body of the loop will be executed at least once as the condition test is at the bottom of the loop. Had the condition test been at the top of the loop, it would have been a While ... End While Loop as discussed in Section 4.3.

Listing 4.4 shows a minimal example where we test D0 and it it is zero, we don't wish to execute the loop body—effectively `REPEAT...UNTIL D0 = 0`.

```
repeatLoop
    ; Do loop body code here
```

```
    ...

until
    ; Do condition test last.
    tst.l d0
    brne repeatLoop

    ; Continue from here.
    ...
```

Listing 4.4: REPEAT UNTIL . . .

You will note that the test of `D0` is negated in order to keep looping around.

This construct is also known as the `Do...Until loop`.

4.3 While . . . End While Loop

This is another structure that isn't available on the QL, but is in other languages. It is similar to the Repeat . . . Until Loop discussed in Section 4.2 but with the condition test at the top of the loop rather than at the end. As the test is carried out at the top of the loop, the loop body may not be executed if the condition is true on entry to the loop.

Listing 4.5 shows a minimal example of this structure—effectively, `While D0 = 0`.

```
whileLoop
    ; Do condition test first.
    tst.l d0
    brne exitWhile

    ; Do loop body code here
    ...

endWhile
    bra whileLoop

exitWhile
    ; Continue from here.
    ...
```

Listing 4.5: WHILE . . . END WHILE

You will, hopefully, note that the condition has again been reversed, we loop back to `whileLoop` if `D0` is not equal to zero

4.4 For . . . End For Loop

`FOR` loops are available on the QL. In Assembly Language we can count up or down, as we can in S*BASIC, but the CPU in the QL, and emulators, has a looping instruction—`dbra` also known as `dbf`—Decrement and branch until false. As the terminating condition is False, the loop will only terminate when the counter register decrements from zero to -1.

Counting upwards can't use that instruction though. Listing 4.6 shows a small example of a FOR loop in Assembly. The condition to determine the end of the loop is at the top of the loop. This example is effectively `FOR d0 = 0 TO 10 STEP 1...END FOR`.

```
forLoop
    clr.b d0                ; For do = 0 ...

nextFor
    cmpi.b #10,d0           ; To 10 ...
    beq exitFor

    ; Do loop body code here.
    ...

endFor
    addq.b #1,d0            ; ... Step 1
    bra nextFor

exitFor
    ; Continue from here
    ...
```

Listing 4.6: FOR ... END FOR forwards STEP

Counting downwards, of course, can make use of `dbra` as Listing 4.7 shows. This example is effectively `FOR d0 = 10 TO 0 STEP -1...END FOR` but uses values from 9 down to -1.

```
forLoop
    moveq.b #10-1,d0        ; For d0 = 10 ...

nextFor
    tst.b,d0                ; To 0 ...
    beq exitFor

    ; Do loop body code here.
    ...

endFor
    dbra d0,nextFor         ; Step -1

exitFor
    ; Continue from here
    ...
```

Listing 4.7: FOR ... END FOR backwards STEP

In Assembly, it is just as simple to use the actual values from 10 to 1 in the register but the `dbra` instruction wouldn't then be possible as that instruction, as we know, terminates the loop when the register value is -1 as opposed to zero.

4.5 For ... Next ... End For Loop

Sometimes, in a `FOR` statement, we want to skip one or more values in the loop. In those cases we make a test and if true, we call `NEXT` in our loop body. Listing 4.8 shows an example where we execute `NEXT` if the value in `D0` is 7.

```
forLoop
    clr.b d0                ; For do = 0 ...

nextFor
```

```
    cmpi.b #10,d0           ; To 10 ...
    beq exitFor

    cmpi.b #7,d0            ; Is D0 = 7? If so ...
    bra nextFor             ; ... NEXT.

    ; Do loop body code here.
    ...

endFor
    addq.b #1,d0            ; ... Step 1
    bra nextFor

 exitFor
    ; Continue from here
    ...
```

Listing 4.8: FOR . . . NEXT . . . END FOR

4.6 If . . . End If

The `IF` statement can lead you down a rabbit hole or two! There are all sorts of potential conditions, signed and unsigned. For more details, see this issue's Beginner's Corner on page 13 for details. For the descriptions below, I'll be sticking to equals and not equals, as these are the simplest to explain and understand; plus we don't have to consider signs either!

Listing 4.9 has the code for a simple `IF D0 = 10 THEN...END IF` statement.

```
ifTest
    ; If D0.B <> 10 Then skip to endIf.
    cmpi.b #10,d0
    bne endIf

thenSection
    ; Do code for D0 = 10 here.
    ...

endIf
    ; Continue from here.
    ...
```

Listing 4.9: If . . . THEN . . . END IF

Once again, you will notice that we reverse the flow and the test. The code is actually testing that `D0` is not equal to 10 and skipping the body of the code, rather than testing if it is equal to 10 and executing the code.

4.7 If . . . Else . . . End If

Listing 4.10 shows an `IF D0 = 10 THEN...ELSE...END IF` statement. There's only a small difference here, after executing the code when `D0` was equal to 10, there's a jump over the code that gets executed when `D0` wasn't equal to 10.

```
ifTest
```

```
    ; If D0.B <> 10 Then skip to elseSection.
    cmpi.b #10,d0
    bne elseSection

thenSection
    ; Do code for D0 = 10 here.
    ...

    ; And when done, skip the ELSE clause.
    bra endIf

elseSection
    ; Do code for D0 <> 10 here.
    ...

endIf
    ; Continue from here.
    ...
```

Listing 4.10: If . . . THEN . . . ELSE . . . END IF

4.8 Exit and Next

You've seen `exit` already. Based on some condition or other, we `exit` the construct by jumping to just past the end of it; or we jump back to the beginning for `next`. Listing 4.11 shows simple examples of each.

```
loopStart
    ; Do some loop body stuff.
    ...

    ; Do a 'next loop' if D0 = 0 at this point.
    tst.l d0                ; Is D0 = 0?
    beq loopStart           ; If so, next loop

    ; Do more loop body stuff.
    ...

    ; Do an 'exit loop' if D7 = 0 at this point.
    tst.l d7                ; Is D7 = 0?
    beq loopExit            ; If so, exit loop

loopEnd
    bra loopStart           ; Let's go round again!

loopExit
    ; Rest of code ....
    ...
```

Listing 4.11: EXIT and NEXT

4.9 Summary

Well, maybe this wasn't such a "quickie" as I first thought. However, we can now write code in a sort of structured manner. I suspect most of us "seasoned" Assembly programmers have been doing this sort of thing automatically for years anyway! But maybe someone will have learned something. I did!

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.

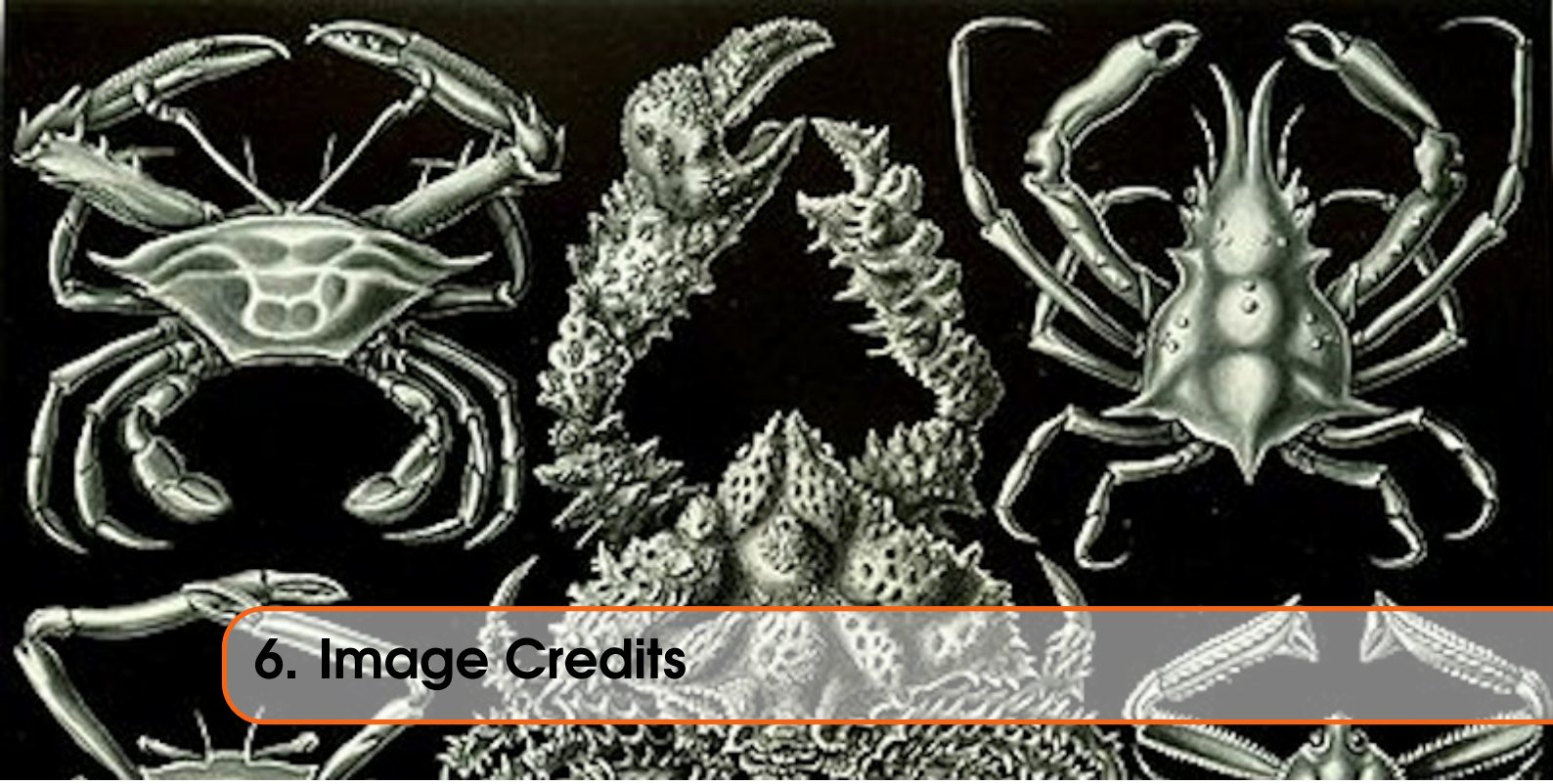
- When `A1` is zero, the SMSQ/E result is the same as the QDOS: JS result, no space yet used on the maths stack.
- When `A1` is negative, the SMSQ/E result appears to show an extra 2 bytes over the size of the space used on the maths stack so far. So for a word sized integer, 2 bytes, the value in `A1` 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. 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.