Mailing List - Issue 1

Assembly Language

Norman Dunbar

Mailing List - Issue 1 Assembly Language

Author Norman Dunbar assembly@qdosmsq.dunbar-it.co.uk

Learning Assembly Language on your Sinclair QL and/or derivative is still easy, even with the sad demise of QL Today Magazine.

This, somewhat irregular, eMagazine hopes to continue on from where QL Today left off after all those years. The benefits of working for myself is that I no longer worry about meeting deadlines, if I can't get an issue out every now and then, then it is a minor niggle (because I would like to do so) but it's not the end of the world. Hopefully, my faithful readers will bear with me!

Preface

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 DocBook source format, aka XML, 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!

Welcome

Welcome to the first issue of the new QDOSMSQ Assembly Language mailing list. Hopefully, there will be many more to come. As of 20th October 2013 there are 42 of you, including myself, out there showing an interest in Assembly Language. Let's hope we can improve - tell all your friends! I've told both of mine! ;-)

Note

Since that was written, there are now 56 subscribers.

This will not be a regular "magazine" as I have just started a new contract which is very intense and busy - I'm helping to supporting over 600 Oracle databases for a large supermarket who's name begins with 'M' and ends with 'orrisons' - I've never been so busy in my life before! However, it's good work and it pays reasonably well, so I'm not complaining. Plus, I get to use new and up to date stuff which I didn't get to do too often in Government work!

In addition, I don't have to worry about publication dates and I can use better diagrams and tables than I've been using before. No more ASCII "Art" required. In fact, I no longer need to convert my XML source into plain text, I build the HTML. PDF and even, dare I say it, Epub eBook files, directly from the XML source code.

1.1. 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*[1] 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!

1.2. 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

[1] http://qdosmsq.dunbar-it.co.uk/mailinglist/

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 Docbook 4.5 Article 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.

1.3. Coming Soon

It is my intention to package up all my previous articles from QL Today, in a PDF book which I intend to send out to the list. I don't know when this will actually happen, but it should happen soon or at least, soonish! I need to ensure that it is suitable for publication, that spelling errors have been corrected and that George's corrections and/or comments have been included. With a bit of luck, it might even be available in HTML format as well, for those of you who are not fond of PDF files for everything.

It does depend on my ability to get the mailing list to allow attachments. The manual says that it does, but so far, I've been unable to actually get a test email sent out with any form of attachment! But I'm still trying, when I get the time.

LibGen

As you may well be aware, **LibGen** suffered from the demise of the printed version of QL Today magazine, in as much as it was only half completed when the magazine, ahem, folded! Fear not, I intend to finish the series and get you a working utility along with all the required source code.

Sadly, since the last instalment of **LibGen** was printed, I've been doing some work on it, and I've introduced a horrendous bug which I'm trying to track down. The symptoms are as follows:

- **EX** the utility, and it appears on screen. It looks as good as it ever did!
- Press ESC to exit immediately. It exits.
- **QPC** is now unusable! The mouse pointer no longer wishes to pass in front of the main **QPC** window but will happily pass *behind* it!

I've been through the code changes with a fine tooth comb, and nothing I've thought of fixed the problem. I'm not looking through the changes I had to make to the window definition, which does worry me a little as if this is a problem with the definition, then it *might* imply that somewhere in WM_SETUP there is a bug that allows a malformed window definition to be accepted. Mind you, having said that, I'm not sure what validation could be done to ensure that everything matches up ok before WM_SETUP does what it does!

The EX Files

I got my head in a tangle recently, but luckily Marcel and company on the QL Users mailing list sorted me out. Them, and the **TK2** manual from Dilwyn's web site!

I was trying to get my head around the contents of a job's stack when first set up. I know some channels and a command string could be on the stack, but I was seeing some extra space being allocated, and the extra space was always 12 bytes more than my dataspace required.

As you know, and as I should have known, when you create a job in QDOSMSQ you supply a code size and a data size. That's all well and good, but when **EX/EW/ET** etc create a job from a file on a device, then other things can happen. These are briefly explained in the QDOSMSQ documentation but the **TK2** manual gives a little more details. It also give a brief hint at a "special" program type with has the ability to be created and then allowed to perform some form on initialisation by itself, using a special routine, but the operating environment that it operates in while running this code is not that of an independent job, but is in fact, the context of **SuperBasic** instead!

Quite why a job would need to be able to initialise itself while in the **SuperBasic** environment was a little puzzling on the QL Users mailing list, and so far, no-one has come up with a good reason for it. I shall continue to ignore it for now!

3.1. My Problem

I have no idea how I got my head so tangled up, but I was of the opinion that in order to pass channels to a job, you had to do something like this:

```
1000 OPEN_IN #3,flp1_fred_txt
1010 OPEN_NEW #4, flp1_newfred_txt
1020 EX #3 to flp1_test_exe to #4
```

Of course, I was completely wrong! That little scenario actually opens the two files, and executes the job. The **EX** code will check that #3 and #4 are already open in **SuperBasic**, and if so, close them, before creating a PIPE channel from **SuperBasic** channel #3 to the input channel for the job and a PIPE channel from the job's output channel to **SuperBasic** channel #4. These two channel ids are placed on the job's stack when it is created.

My brain thought that the job would execute and read from the file attached to #3 while it wrote to the file attached to #4. How wrong was I? Anyway, the upshot was I learned a couple of things - or at least, remembered them. I found on Dilwyn's web site, a document about creating filters that I wrote years ago which explained exactly how the above worked. What I should have been thinking, in order to send the file #3 to the job and from there to #4 was the following:

```
1000 OPEN_IN #3,flp1_fred_txt
1010 OPEN_NEW #4, flp1_newfred_txt
1020 EX flp1_test_exe, #3, #4
```

Anyway, the fact that I got myself confused means that I've done a bit of investigation and here's the explanation for your information. The remainder of this section assumes the following setup.

The job has a standard job header in the following format:

```
Job_start    bra.s   start
             dc.l    0
             dc.w    $4afb
             dc.w    $08
```

```
            dc.b    'Job Name'

start       ; Job code actually begins here....
```

When the job is finally created, certain registers are given specific values. Unless otherwise stated, all registers are set to zero, except:

- A6.L - points to the end of the internal job header, and the beginning of the code section. This is the label "Job_start" above.
- A4.L - contains the length of the job's code section (ie, the file size on disc).
- (A6,A4.L) - points to the first byte of the job's data section.
- A5.L - contains the size of the job's data space, plus an allowance for the size of the parameter string and any opened channels on the job stack, plus 4, rounded up to an even number.
- (A6,A5.L) - points to the first byte past the end of the job's data space.
- A7 - holds the address of the job's stack, which is at the top of the data space, so is set to be A6+A5-4 for a job that was passed no channels and no command string. For jobs with parameters and/or command strings, A7 is set to A6+A5-sum(stuff on stack) - see below.

  A7 always begins by pointing at a word on the stack, containing the number of channels that are stacked for this job. Above the word are the individual channel IDs (.L) and above the last one is a word containing the length of the command string followed by the bytes of the command string, plus a single padding byte, if required, to keep things even.

Note

You should be aware that a job needs to have been written in such a way as to use the channels etc passed to it by the **EX** command. The data described below will always be placed on the job's stack, even if the job ignores it. Many assembly language programs do something like the following, on startup, to manipulate the stack and clear the various channels and such like off the stack if it doesn't require them. It can help save a few bytes here and there, and get you a few more bytes of stack/data space!

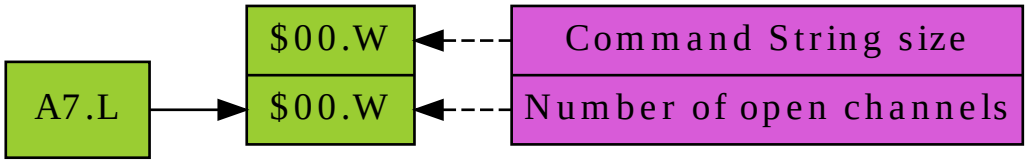
```
Job_start   bra.s   start
            ...

start       move.w  (a7)+,d0        Fetch number of open channels on stack
            lsl.w   #2,d0           Multiply by 4 to get bytes total
            adda.w  d0,a7           Skip over the channel ids on stack
            move.w  (a7)+,d0        Fetch the size of the command string
            addq.w  #1,d0           Add one in preparation to ...
            bclr    #0,d0           ... Make d0 even
            adda.w  d0,a7           Tidy command string, and padding, off stack.
            ...
```

3.2. No Channels, No Parameter String

```
1000 EX flp1_test_exe
```

This is the simplest case. A7 points to a pair of words on the stack. The first, as explained above, is the number of open channels (zero) and the second is the length of the command string (also zero).

The stack will look like this:



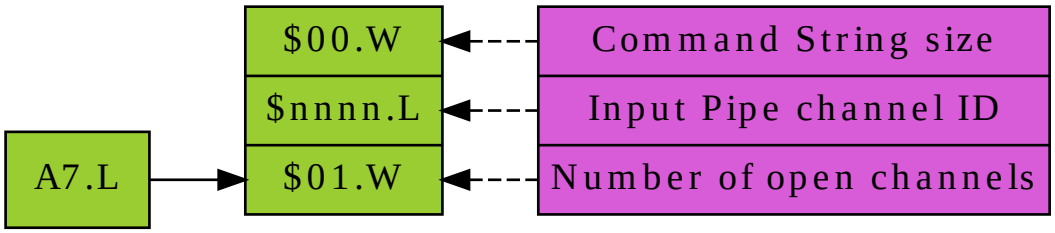
As you can see, A7 points at a long word of zero on the stack. This is broken down into two zero words, one representing the number of open channels on the stack, the other, representing the size of the passed command string. Both of which, are zero. Simple case to begin with.

3.3. One Input Pipe

```
1000 EX #3 TO flp1_test_exe
```

The current **SuperBasic** channel #3 will be closed and a new PIPE channel opened for input. The channel id will be put on the stack at 2(A7) and the word at (A7) will be $01. The word at 6(A7) will be set according to the command string passed, in this example, zero. The job should not close the PIPE channel, but simply remove itself from the system when complete.

The stack will look like this:



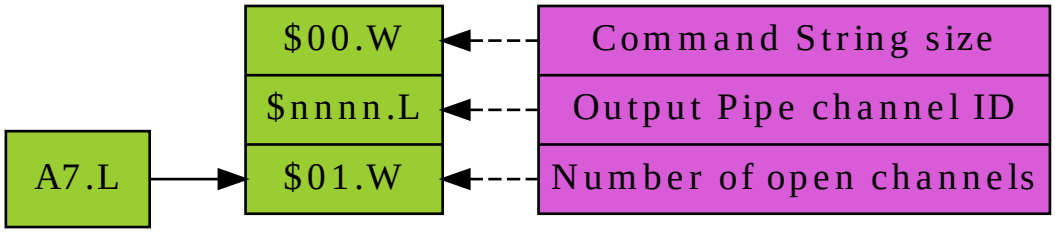
Anything printed by **SuperBasic** to channel #3 will be read by the job, provided the job actually reads from the PIPE of course!

3.4. One Output Pipe

```
1000 EX flp1_test_exe to #4
```

The current **SuperBasic** channel #4 will be closed and a new PIPE channel opened for output. The channel id will be put on the stack at 2(A7) and the word at (A7) will be $01. The word at 6(A7) will be set according to the command string passed, in this example, zero. The job should not close the PIPE channel, but simply remove itself from the system when complete this will set EOF on the channel which can be detected by **SuperBasic**, or, by another job that is reading the output from this one.

The stack will look like this:



Anything written by the job to the PIPE channel can be read from **SuperBasic** by something like:

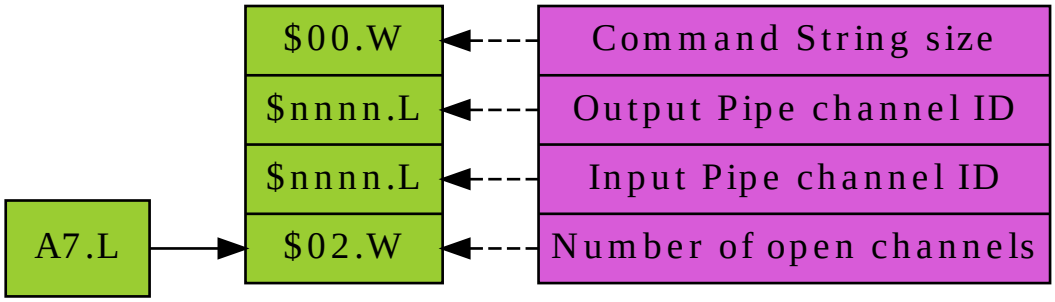
```
1000 EX flp1_test_exe to #4
1010 REPeat loop
1020   IF EOF(#4) THEN EXIT loop: END IF
1030   INPUT #4, a$
1040   PRINT "The job says ... " & a$
1050 END REPeat loop
```

3.5. One Input and One Output Pipe

```
1000 EX #3 TO flp1_test_exe TO #4
```

The current **SuperBasic** channels #3 and #4 will be closed and a new PIPE channels opened for input and output. The input channel id will be put on the stack at 2(A7), the output channel id will be stacked at 6(a7) and the word at (A7) will be $02. The word at $0a(A7) will be set according to the command string passed, in this example, zero. The job should not close the PIPE channels, but simply remove it-self from the system when complete this will set EOF on the output channel which can be detected by **SuperBasic**, or, by another job that is reading the output from this one.

The stack will look like this:



3.6. No Pipes, One File

The files can be already opened channels or a filename. The file is always opened for input - OPEN_IN - if a filename is passed.

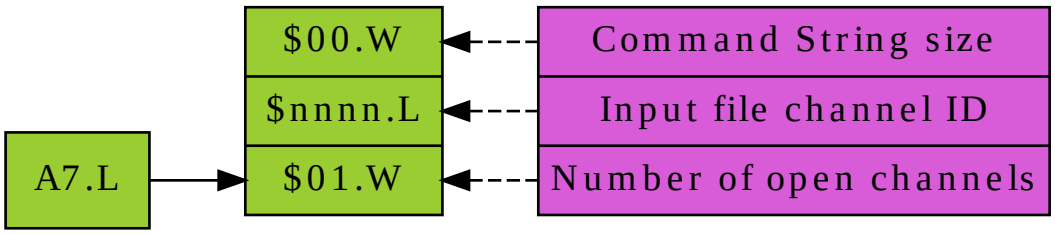
```
1000 EX flp1_test_exe, #3
```

or

```
1000 EX flp1_test_exe, flp1_fred_txt
```

Obviously, channel #3 must be open in such a way as to allow reading from the channel.

The stack will look like this:



3.7. No Pipes, Two Files

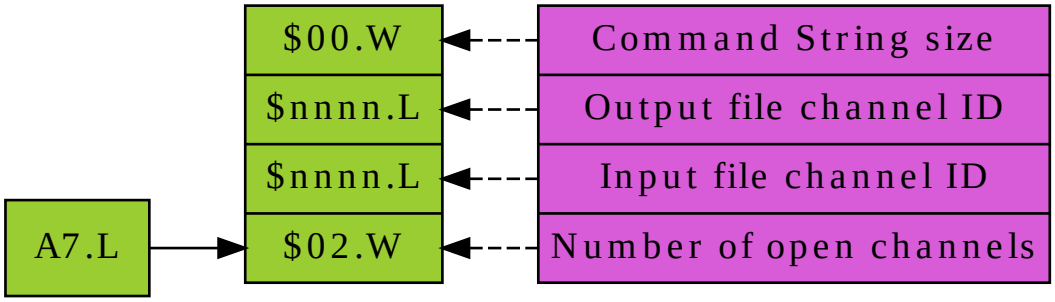
The files can be already opened channels or a filename. The first file is always opened for input - OPEN_IN - if a filename is passed. The last file is always opened in OPEN_OVER mode when passed as a filename. Obviously, if passed as a channel, it has to be writable.

```
1000 EX flp1_test_exe, #3, #4
```

or

```
1000 EX flp1_test_exe, flp1_fred_txt, flp1_frednew_txt
```

The stack will look like this:



3.8. No Pipes, N Files

```
1000 EX flp1_test_exe, #3, #4, #5 ..., #n
```

or

```
1000 EX flp1_test_exe, flp1_fred_txt, flp1_barney_txt, ..., flp1_frednew_txt
```

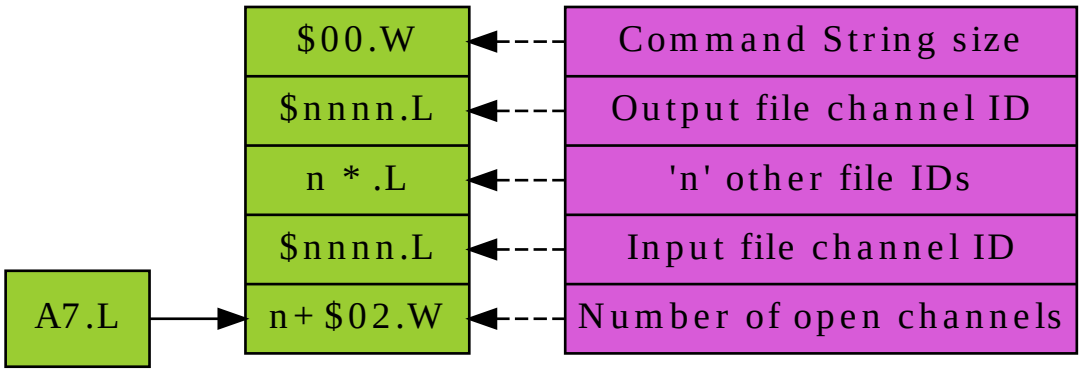
The files can be already opened channels or a filename. The first file is always opened for input - OPEN_IN - if a filename is passed. The last file is always opened in OPEN_OVER mode when passed as a filename. Obviously, if passed as a channel, it has to be writable.

Note

At this point in time, I'm not certain which mode the intermediate files and/or channels are opened. I assume, always a bad thing, that if I pass channels over, then they are open in the mode that they were originaly opened in, so if I try to pass a channel that was OPEN_IN'd, then writing to that particular channel will fail.

However, if I pass a file name, what mode does QDOSMSQ open the file in.

The stack will look like this:

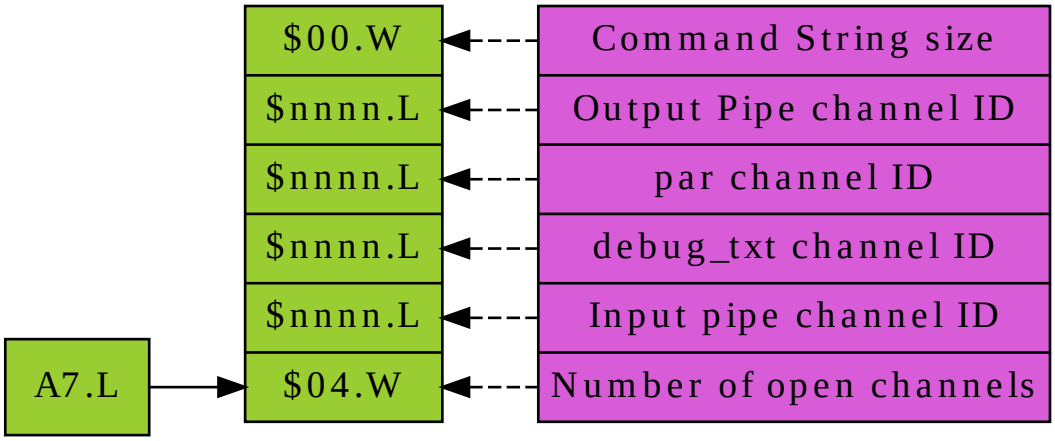


3.9. Pipes and Files

```
1000 EX #3 to flp1_test_exe, debug_txt, par to #4
```

or similar command line, where the files are passed as #channel numbers for already opened files and the pipe channel, #3 in this case, will be closed if it is already open, and the opened as an input pipe and passed to the job as the first channel on the stack. Channel #4 will also be closed, if currently open, and passed as a new output pipe as the final channel id on the job's stack.

The stack will look like this:



3.10. Pipelines

Jobs can be pipelined together so that each one does a little bit of work by reading from an input PIPE and writing to an output PIPE. The next job in line will read this job's output as its own input, and so on.

```
1000 EX flp1_test_exe, flp1_fred_txt TO flp1_paginate_exe TO flp1_linenumber_exe,
 flp1_frednew_txt
```

Each job in the above will have the channels set up on stack as described above and data will pass from the file **`flp1_fred_txt`** through the various jobs before finally being written back to the disc as **`flp1_frednew_txt`**, all paginated and line numbered.

In the above command, there will be 3 separate jobs in the pipeline:

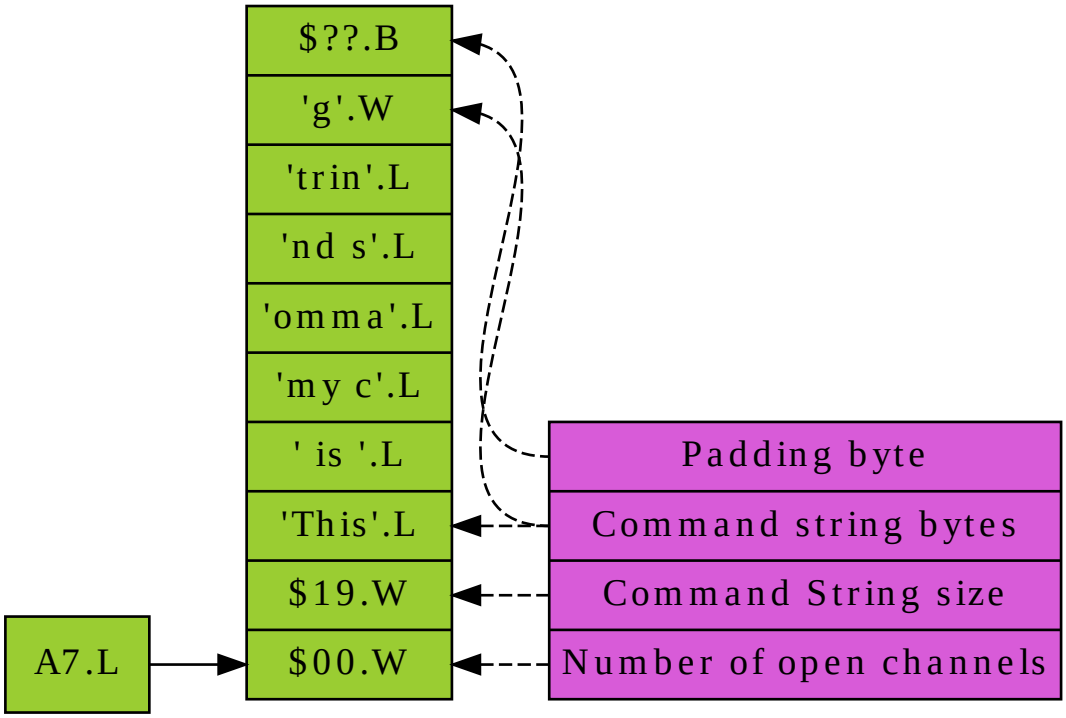
3.11. Command String

```
1000 EX flp1_test_exe ; 'This is my command string'
```

This is another simple case. A7 points to a single word on the stack. As explained above, is the number of open channels and following this word, is a set of channel IDs as described above. Following the final channel id, of zero to however many there were, is another word defining the length of the command string.

The example command above will therefore have a channel count of zero followed by a word of 25 decimal ($19) defining the size of the command string, then the bytes of the command itself, and finally, because the length is odd, a single byte of padding to even up the stack pointer. The value of the padding byte is undefined - so don't count on it being zero, for example.

The stack will look like this:



So there you have it, all sorts of ways in which the parameters and files passed to a job, are arranged on the job's stack when the job starts.

An Example - LibGen_Lite

Putting all that we have learned above into action, I offer for your delight, **LibGen_Lite**. This is a utility that does almost exactly what **LibGen** is intended to do, but without the bells and whistles of the Pointer Environment. It uses the techniques described above to filter out the bits we don't need from a SYM_LST file, and write out only the bits we do need.

You execute **LibGen_Lite** as follows:

```
ex LibGen_lite_bin, sym_lst_file, Lib_file ; 'Bin_file_name'
```

Where:

- *Sym_lst_file* is the listing file created by George's **SYM_BIN** utility. This is the same as the SYM_FILE in **LibGen** itself. This will be opened by QDOSMSQ before it gets to the utility, so there is no file opening and error handling code required.
- *Lib_file* is the output library file, the same as LIB_FILE in **LibGen** itself, and will be opened for output prior to the program executing.
- *Bin_File* is the same as BIN_FILE in the PE version of **LibGen**. This is the name of the binary file that we are building a library for. It will usually be the same name as the Sym_lst file, with a 'bin' extension rather than a 'sym_lst' extension.

So, if you have assembled, to **ram1_**, a test library called **ram1_test_lib_asm** to create **test_lib_bin**, **test_lib_lst** and **test_lib_sym**, then you have run **SYM_BIN** against **test_lib_sym** to create **test_lib_sym_lst**, and you wish to filter the results and create a library, this is what you would do:

```
ex LibGen_lite_bin, ram1_test_lib_sym_lst, ram1_test_lib_lib ; 'ram1_test_lib_bin'
```

There's nothing to stop you developing a utility library on **ram1_** and then saving the results to **win1_libs_**, in that case, the following would create the library for you, in the correct place.

```
ex LibGen_lite_bin, ram1_test_lib_sym_lst, win1_libs_test_lib_lib ; 'win1_libs_test_lib_bin'
```

Obviously, you need to make sure that you keep your source and bin files in the correct places. Now, when you wish to use the library in a program you are writing in assembler, all you need is the following:

```
          in win1_libs_test_lib_lib
```

And that's all there is to it. That file will define the various entry points into the library, and will execute the "lib win1_libs_test_lib_bin" command to pull in the actual library file itself. Easy!

Here's the code, as you can see, there's not much to it as we have no need to open files, check that they exist or close them etc. It's done for us, by **EX** or **EW** as we choose, so there's less code for us to write.

4.1. Program Listings

```
;============================================================
; Libgen_lite:
;
; A filter program using an input and output channel, passed on
; the stack for it's files, plus a command parameter which is
```

```
; the binary file we wish to include.
;
; EX libgen_lite_bin, sym_lst_file, lib_file ; 'bin_file'
;
;-----------------------------------------------------------------
; 04/12/2013 NDunbar Created for QDOSMSQ Assembly Mailing List
;-----------------------------------------------------------------
; (c) Norman Dunbar, 2013. Permission granted for unlimited use
; or abuse, without attribution being required. Just enjoy!
;=================================================================

me            equ   -1
infinite      equ   -1
err_bp        equ  -15
eof           equ  -10
linefeed      equ  $0a
star          equ  '*'
plus          equ  '+'

param         equ  $0c
param_size    equ  $0a
lib_id        equ  $06
sym_id        equ  $02


start         bra  libgen_lite
              dc.l 0
              dc.w $4afb

name          dc.w name_end-name-2
              dc.b 'LibGen_lite'
name_end      equ  *

buffer        ds.b 128+2

lib_text      dc.w lib_text_end-lib_text-2
              dc.b '          lib '
lib_text_end  equ  *


;-----------------------------------------------------------------
; Stack on entry:
;
; $0c(a7) = bytes of parameter + padding, if odd length.
; $0a(a7) = Parameter size word.
; $06(a7) = lib_file channel id (output).
; $02(a7) = sym_lst_file channel id (input).
; $00(a7) = How many channels? Should be $02.
;-----------------------------------------------------------------
```

The first part of the program, as shown above, consists of the introductory text, a few equates and the standard QDOSMSQ job header. Note that the text at label lib_text has 10 spaces at the start, and one at the end. The last few lines describe what the stack looks like when this program is executed.

```
libgen_lite       cmpi.w  #$02,(a7)          ; Two channels is a must
                  bne.s   bad_parameter      ; all ok so far
                  tst.w   param_size(a7)     ; Was there a parameter?
                  bne.s   start_loop         ; All ok so far

bad_parameter     moveq   #err_bp,d0
                  bra     error_exit         ; Bale out, another bad parameter

start_loop        moveq   #infinite,d3       ; Timeout - preserved throughout
read_loop         move.l  sym_id(a7),a0      ; Input channel id

read_loop_2       lea     buffer+2,a1        ; Where to read the data into
```

```
                moveq   #128,d2             ; Maximum size of the buffer
                moveq   #io_fline,d0
                trap    #3
                tst.l   d0
                beq.s   read_ok             ; Not EOF yet, carry on
                cmpi.l  #eof,d0             ; EOF?
                beq.s   end_loop            ; Yes, exit the main loop
                bra.s   error_exit          ; Some other error occurred
```

This is the main part of the code. We start by checking the word that A7 points to on the stack. If it isn't exactly $02, then we bale out with a bad parameter error in D0. If we do have two files on the stack, we then make sure that we have a parameter string as well. This can be pretty much anything, so the only valid check we can do here is that we have a length greater than zero. If we don't, then out we go with another bad parameter error.

If all is well, we start the main loop by setting a timeout in D3. As D3 is never corrupted by the remainder of the code, we don't need to set it each time through the loop. The main loop itself sets A0 to be the channel id of the first file on the stack, the input file - in our case, the SYM_LST. A1 is set to be the third character in the buffer area, which is 128 bytes long plus an extra 2 bytes to hold the text length in normal QDOSMSQ string format.

We trap out to IO_FLINE to fetch a line of bytes, maximum 128, terminated by a linefeed. If this works, the Z flag will be set and D1 will contain the length of the text just read, including the linefeed. We will drop into the next part of the code (below) if all is well. We will exit the main loop if we hit EOF on the input file, and terminate the program on any other error.

```
read_ok         cmpi.w  #1,d1              ; Just a linefeed?
                beq.s   read_loop_2         ; Yes, ignore those
                lea     buffer,a1
                move.w  d1,(a1)+            ; Store length of buffer
```

The code above simply converts the buffer area into a QDOSMSQ string by storing the length word at the start of the buffer, if, and only if, the length of the text just read in was greater than one byte. As the value in D1.W is the text length, including the terminating linefeed, then a value of one byte must indicate only the linefeed. We are not interested in those, so we skip back to read the next line from the input file - the channel id in A0 is unchanged, so we don't have to load it again..

```
;---------------------------------------------------------------
; At this point we have a string in the buffer, with a linefeed
; at the end. Scan it for '*+' which, if found, causes the text
; to be written out to the output file.
;
; A0 = input channel id
; A1 = Buffer pointer -> first character
; D0 = 0
; D1 = Text length, including linefeed
; D2 = 128 = buffer length
; D3 = -1 = timeout
;---------------------------------------------------------------

instr           bra.s   instr_end_loop

instr_loop      cmpi.b  #star,(a1)+         ; Find a '*' first
instr_end_loop  dbeq    d1,instr_loop

;---------------------------------------------------------------
; When we get here, if Z flag is set, we found a '*', else, we
; didn't so ignore this text, and start reading again.
;---------------------------------------------------------------

                bne.s   read_loop_2
```

```
                cmpi.b  #plus,(a1)         ; Now, look for a '+'
                bne.s   read_loop_2        ; Abort scanning
```

When we find a valid line of text in the input file, the code above, at instr_loop, starts looking for an asterisk. If we run out of characters before we find one, then we simply go back to read the next line of input. At the end of the loop, the Z flag will be set if we found an asterisk and the dbeq instruction will not have branched, it will have dropped through. Dbeq means, remember, decrement and branch *unless* equal.

If we did find an asterisk, we check the next character and if that is not a plus sign, we are not interested in this line of text, and skip back to read the next one. If we did find a plus, we drop into the following code to process this line of text.

```
;-------------------------------------------------------------
; We have found '*+', write out the text to the lib_file.
;-------------------------------------------------------------

write           move.l  lib_id(a7),a0      ; Get the lib_file id
                lea     buffer,a1          ; Reset the buffer pointer
                move.w  (a1)+,d2           ; Get the length, inc linefeed
                bsr.s   write_str          ; Write it to output file
                beq.s   read_loop          ; Continue if ok, else
                bra.s   error_exit         ; Exit on error
```

The line of text, just read in from the input file contains the text '*+' so we want to write this line out to the lib file. And that's exactly what the code above does. The file ID is obtained from the stack and the buffer written out, exactly as it was read. Assuming the write succeeds, we go back and read another line of text - from the point where we load A0 with the input channel id - otherwise, it's time to bale out.

At the end of the input file, the following code will be executed.

```
;-------------------------------------------------------------
; We get here on EOF.
;
; D0 = -10 = EOF
; D2 = 128 = buffer length
; D3 = -1 = timeout
; A0 = sym_lst_file channel id (input)
; A1 = Buffer + 2
;-------------------------------------------------------------

end_loop        move.l  lib_id(a7),a0      ; Get the lib_file id
                bsr.s   write_lf           ; Write out a linefeed
                bne.s   error_exit         ; Exit on error

                lea     lib_text,a1        ; String to write
                move.w  (a1)+,d2           ; Size word
                bsr.s   write_str          ; Write it out
                bne.s   error_exit         ; Exit on error

                move.w  param_size(a7),d2  ; Size of parameter string
                lea     param(a7),a1       ; Pointer to parameter string
                bsr.s   write_str          ; Write it out
                bne.s   error_exit         ; Exit on error

                bsr.s   write_lf           ; And a final linefeed
                beq.s   all_done           ; Exit on error
```

At the end of the input file, our output file consist of all the code entry points in our library. As **GWASL** requires these, plus a "lib" instruction immediately after the offsets, the utility adds one to the output

file, thus your code need only "in" the lib file just created to be sure of getting everything included correctly.

The output file ID is grabbed from the stack, and a linefeed written to it, followed by the "lib" command and then the parameter that was passed to the utility on the command line. This is the name of the bin file that the library of routines is held in. Once a final linefeed is written, we exit via the following code.

```
error_exit        move.l  d0,d3             ; Error code we want to return
                  bra.s   suicide           ; And die

all_done          moveq   #0,d3             ; No error code

suicide           moveq   #mt_frjob,d0
                  moveq   #me,d1            ; This job is about to die
                  trap    #1
```

In the event of any errors in the utility, we would exit through the error_exit label above. D0's error code will be copied to D3 ready to return to **SuperBasic**. In the event that all was well, we exit via the code at all_done above, where D3 is cleared to indicate all was well. The job then commits suicide which will cause all it's files to be flushed and closed prior to the job being removed.

```
;---------------------------------------------------------------
; Write_str
;
; Write a string from A1, to the channel in A0 with the length in
; D2 and D3 holds the timeout.
;
; Exit with D0 holding the error code and the Z flag set or not.
; D1 and A1 are corrupted.
;---------------------------------------------------------------

write_str         moveq   #io_sstrg,d0
write_trap        trap    #3
                  tst.l   d0
                  rts

;---------------------------------------------------------------
; Write_LF
;
; Write a linefeed to the channel in A0, with D3 holding the
; timeout.
;
; Exit with D0 holding the error code and the Z flag set or not.
; D1 and A1 are corrupted.
;---------------------------------------------------------------

write_lf          moveq   #io_sbyte,d0
                  moveq   #linefeed,d1
                  bra.s   write_trap
```

The code above defines two simple sub-routines that write a linefeed or a string of text to the channel id in A0.

4.2. LibGen_Lite - Example Usage

The following is a SYM file from an early version of **LibGen_lite**, which has been passed through George's **SYM_BIN** to create a textual version of the binary symbol file created by assembling the utility.

```
ME                              EQU   $FFFFFFFF
ERR_BP                          EQU   $FFFFFFF1
EOF                             EQU   $FFFFFFF6
LINEFEED                        EQU   $0000000A
START                           EQU   *+$00000000
LIBGEN_LITE                     EQU   *+$000000AA
NAME                            EQU   *+$0000000A
NAME_END                        EQU   *+$00000017
BUFFER                          EQU   *+$00000017
LIB_TEXT                        EQU   *+$0000009A
LIB_TEXT_END                    EQU   *+$000000AA
BAD_PARAMETER                   EQU   *+$000000B6
START_LOOP                      EQU   *+$000000BC
ERROR_EXIT                      EQU   *+$0000012A
READ_LOOP                       EQU   *+$000000BE
READ_OK                         EQU   *+$000000DA
END_LOOP                        EQU   *+$00000108
INSTR                           EQU   *+$000000E6
INSTR_END_LOOP                  EQU   *+$000000EC
INSTR_LOOP                      EQU   *+$000000E8
WRITE                           EQU   *+$000000F8
WRITE_STR                       EQU   *+$00000136
WRITE_LF                        EQU   *+$0000013E
ALL_DONE                        EQU   *+$0000012E
SUICIDE                         EQU   *+$00000130
WRITE_TRAP                      EQU   *+$00000138
```

The following command was executed to use **LibGen_Lite** to produce a lib file for us to use.

```
ex LibGen_Lite_bin, ram1_LibGen_Lite_sym_lst, ram1_LibGen_lib ; 'win1_libs_libgen_lite_bin'
```

The above command created, as the output file, the following:

```
START                           EQU   *+$00000000
LIBGEN_LITE                     EQU   *+$000000AA
NAME                            EQU   *+$0000000A
NAME_END                        EQU   *+$00000017
BUFFER                          EQU   *+$00000017
LIB_TEXT                        EQU   *+$0000009A
LIB_TEXT_END                    EQU   *+$000000AA
BAD_PARAMETER                   EQU   *+$000000B6
START_LOOP                      EQU   *+$000000BC
ERROR_EXIT                      EQU   *+$0000012A
READ_LOOP                       EQU   *+$000000BE
READ_OK                         EQU   *+$000000DA
END_LOOP                        EQU   *+$00000108
INSTR                           EQU   *+$000000E6
INSTR_END_LOOP                  EQU   *+$000000EC
INSTR_LOOP                      EQU   *+$000000E8
WRITE                           EQU   *+$000000F8
WRITE_STR                       EQU   *+$00000136
WRITE_LF                        EQU   *+$0000013E
ALL_DONE                        EQU   *+$0000012E
SUICIDE                         EQU   *+$00000130
WRITE_TRAP                      EQU   *+$00000138

          lib win1_libs_libgen_lite_bin
```

You can hopefully see that all the equates from the top file have been removed from the generated lib file. Only those symbols that relate to code labels are left behind. These are the entry points to various routines in the final library file. At the end of the generated file, you will see that there is a "lib" command to pull in the required binary library file immediately after the equates that define the included code routines.

Obviously, testing **LibGen_Lite** against itself isn't really what the utility was designed for, but it does show an example of its use.

Now there's a problem or two with this "lite" version. You have no choice in which routines are included in the library. The symbol file lists *every* code label and equate when created. **LigGen_Lite** removes only the equates - which are not required - and leaves *all* the code labels. You might have built a library with a few routines you wish to be visible, and a number of helper routines that are desired for internal use only - **LibGen_Lite** can't help you there, you will need to edit the generated lib file manually and remove the ones you don't wish to be visible.

The PE version of **LibGen** will, when I get it debugged, allow you to pick and choose what you wish to include before it writes out the generated lib file.

As mentioned way back at the beginning of this article, you don't have to pass filenames across, the following was useful in testing:

```
ex LibGen_Lite_bin, ram1_LibGen_Lite_sym_lst, #2 ; 'win1_libs_libgen_lite_bin'
```

The output being written to the screen on channel #2 instead of being written to a file, which then had to be edited to see if all was well. The following works equally as well:

```
open_in #3, ram1_LibGen_Lite_sym_lst
ex LibGen_Lite_bin, #3, #2 ; 'win1_libs_libgen_lite_bin'
```

4.3. Errors - Beware!

In the event of an error, you will notice that the code returns the error in D3, just before committing suicide. However, what happens to these error codes when the job exits? Simple - if you **EXEC** or **EW** the code, you will not see the errors. The job is running independently, and doesn't let **SuperBasic** know that it failed.

If you start the job with **EW** or **EXEC_W**, **SuperBasic** waits for the job to complete, and when it has done so, if an error occurred, **SuperBasic** will report it back to you.

For this reason, if you are having problems when using the utility, and you are wondering why, try using **EW** instead of **EX** and see what error messages you get back. It might help.

ET Phone Home

JMON2 and **QMON2** are useful tools in helping debug assembly language programs that are not behaving correctly. But how would you use either of these tools to enable you to debug a filter like the **LibGen_Lite** utility above?

The simple answer, as far as **JMON2** is concerned, is that you cannot. (Or, at least, it appears that way as far as my testing has shown.)

QMON2, on the other hand, can be used quite easily. And the infrequently used **ET** command is all that is required. Oh, and some way of finding out which jobs are running in the system of course.

To use an example from my own testing, here is the command I used. You can see that it is similar to an example shown above, but I have replaced the **EX** command with the **ET** variant.

```
jobs
et LibGen_Lite_bin, ram1_LibGen_Lite_sym_lst, #2 ; 'win1_libs_libgen_lite_bin'
jobs
```

The first **`jobs`** command lists the currently running jobs, as you would imagine. The **ET** command executes the filter task, but stops after the job has been set up. The final **`jobs`** command lists all the jobs again and allows you to spot the new job.

You will possibly notice that regardless of what the job name should be, according to the job code, it isn't anything like it! That's the job you want to trace with **QMON2**, so run the command to attache to the job number. In my test, it was job 17, so:

```
qmon2 17
```

And debugging can continue as normal, even though this is a filter that we are debugging. Neat!