

costs £18 and is available to Z88 USER subscribers for only £12 for a limited time.

The AGM itself started at 4pm, the main proposals passed were ones to reduce slightly the size of the committee in view of the somewhat smaller membership, and thanking two committee members who had resigned recently for their service. The main officials were re-elected to their posts unopposed. During discussion at

the AGM itself, we discovered from Cyril Phillips that a couple of new faces, actually a couple of young lads, had attended the workshop and gone home the proud owners of their first QLs, having bought one each. A happy note to end a great weekend on. Just goes to show how much life there still is in the QL, and with developments like the Q40, QPC2 and (hopefully) internet access, long may it all last.

possible when opening a file - it either works or fails immediately) - useful for Things which may have only one user at a time, so you can try using it for a predefined time and only then give up.

The address of the Thing is what you get, and that's all you need, because this is the main idea: a kind of "filing system" for parts of your computer's system, together with a user registration. Sounds simple, doesn't it? And it is simple.

A job is a user of a Thing until it frees (similar to "close") it. As long as a job is a user of a Thing, its "life" depends on the presence of this Thing. As SMSQ is, just like QDOS, a self-cleaning system (which means that if you remove something from the system, all items related to this "something" are also released/closed etc.). If you remove a job in SMSQ/QDOS, then all channels which are owned by this job are automatically closed by this system, all memory owned by this job is returned to the free memory list, all jobs owned by this job are removed as well and this works recursively until all resources owned or used by this job are freed.

Things go a step further. When a job becomes the user of a Thing, it seems logical that it wants to do something with it or relies on its presence. If the Thing is removed, then naturally all jobs which are registered as users of this Thing will be removed by the system as well - how could a job continue using the Thing if it is gone? This may lead to unpredictable results and we prefer having a stable system. It may sound a bit tough to remove jobs but think about it - what else can be done?

So what are these Things? - Part 1

Jochen Merz

I know I should have written this article some time ago, but I was not sure what to write and how to start.

I decided I will start with some general explanation about Things, tell you what you could do with them (most likely, you are using them for quite a while without knowing) and carry on explaining how to use them and even how to create them. Of course, that's not going to fit into a single issue and as we don't want to fill a complete issue with assembler listings only, I'll split it into parts. Take it as an additional tutorial part to Norman's assembler series.

To make sure you don't get too confused, I will put the term "Thing" in upper case whenever I mean the SMSQ meaning of Thing.

The name Thing was used because a Thing can be virtually everything. It can be a device driver, data area, it can be a menu, it can be a system extension, it can be a program ...

The first advantage of a Thing is that it has a unique

name. You can identify a Thing by its name, and find it in the system's memory. A Thing can be positioned anywhere in the memory of your computer, so the name is the only way to find it (similar to how you refer to a file by specifying its filename).

If a job wants to use a Thing (there can be different ways of using a Thing depending on what it actually is, but more on this later). First of all, the job has to try to find out whether a particular Thing exists in your system - otherwise it can't use it. Similar to a file - you have to try to open it in order to find out if it exists.

If the Thing exists and the job is allowed to use it (there may be limitations which depend on the Thing, some Things may be used by one user at a time, others may have more than one user ...) then the job is marked as a user of this particular Thing and it now "knows" that the Thing exists and gets the address of the Thing. It is even possible to specify a timeout when trying to use a Thing (something which is not

Different Things...

As I have already mentioned ... it is possible to have different kinds of Things. Theoretically it would be possible that every piece of used memory could be a Thing. This would probably be "overkill", but it would be possible. Not every memory allocation needs to be recognisable by name, but if you think about it, you will find that it could be useful if larger parts of your memory, which contain different parts of the operating system and device drivers were recognisable by name. You could customise your system while it is running, add drivers, remove drivers, update drivers.

In the past, adding another RAM-Disk driver while another RAM-Disk driver is used will not lead to a clean system. The old RAM-Disk driver remains in the system, you will find two "RAM" entries in the QPAC2 list of devices, but the first one is not removed. You can still access it - it may be a bit tricky, but it is possible. And what about files being open to the first RAM-Disk driver? They remain open until you close them. If you open the same filename again it will be opened to a different RAM-Disk ... ooops, ambiguous, isn't it? If the RAM-Disk-driver would be a Thing, then, when the new RAM-Disk-Driver is loaded, the old RAM-Disk-Driver will be removed from the system, the whole memory used by it will be released and the jobs with open channels to this RAM-Disk will be removed. What else can be done? There is no half-way tidy-up.

Things can also be system utilities or system extension.

The Menu extension is a very good example. If you replace the Menu extension by a more recent version, then the old Menu extension goes away and is fully replaced by the new one. Try it: start a QD, try loading a file but when the file-select window appears (which is not part of QD, it is part of the Menu extension and QD is a user of the Menu Extension at this moment), then switch back to BASIC and LRESPR MENU_rext once again. First, the old Menu extension is removed (and therefore QD will be removed as well - remember, all users are zapped when a Thing is zapped) and then a new Menu extension is installed.

QD (and other programs) only use utility Things at the time they really use them - this means you can load a new Menu extension without jobs being removed as long as no pulldown windows contained in the Menu extension are open in other jobs. It would be possible (and maybe a little bit faster) for a job to register itself as a user of the Menu extension (you know, it then gets the address of the extension) and then remain being a user and call the extension every time it needs it. This will save a few use and free calls, and it is guaranteed to work too (the address CANNOT change: replacing the Menu extension, which would lead to a change of this address would not do any harm because at this time QD would not exist anymore - remember, users of the old Thing are gone...).

I hope you are not too confused by now - once again think of the Thing system as

an identification and registration system.

There is no limit on what Things can be. It seems logical that if we can have extension Things, we can have executable Things as well (just like files, you see the analogy all the time?). It is possible to have many input-channels open to the same file, so why not have many users who use the same program code which exists only once in memory? This is the concept of executable Things. The program code, which is the same all the time no matter if zero, one, or any number of the same jobs use it, needs to be in memory only one time. All QPAC2 programs are executable Things, and you can execute as many as you like until you run out of memory. Now you will definitely agree that users have to go away if the Thing they are using is removed - how can a program execute when its program code is gone? You see, the concept does not sound too drastic, does it?

The third general Thing type is the "data" Thing. Once again it is entirely up to the creator of the Thing what kind of data is stored in here, how it is organised and used. I can't repeat it often enough: the Thing system is the tool to help a job finding the Thing and registering for it.

I think that's enough for this time. You may like to read this article a second time. If this was too difficult, write and tell me what you have not understood. In the next part, we will have a look at the Things in your system (provided you have loaded QPAC2) and one of the various ways to create Things yourself.



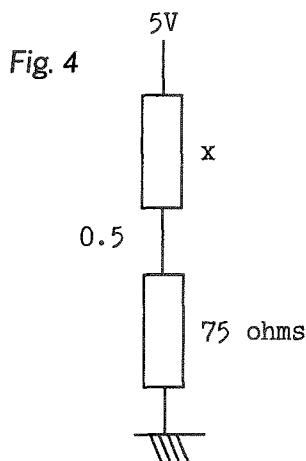
Practical Result

The picture still appeared slightly too bright in comparison to normal television.
Video output from QL RGB signals = 5 volts peak-to-peak. Now see Fig. 3



Monitor wants 0.7Vpp, that's $(1 * 0.7)/(1+0.4) = 0.5$ volts excluding sync.

Now see Figure 4.



Find the value of x (the series resistor).

$$V=IR \text{ and } V/R = I$$

$$0.5/75 = (5-0.5)/x$$

$$\text{so } x = 675 \text{ ohms.}$$

Nearest preferred resistor value is 680 ohms.

Practical Result

Brightness and contrast controls on the monitor did not require adjusting.

Tips

Place 680 ohm resistors to SCART pins 7,11,15. Place heat shrink sleeving over each. Solder RGB wires from the QL to the resistors. Use 5 core cable from QL to SCART. Use Military type high temperature, not Burglar Alarm cable. Heat the heat shrink to insulate.

The military specify high temperature cable because they are no good at soldering like the rest of us. It's nothing to do with them wanting electronic equipment to work after it's been blown up!

Next Issue: QL composite sync to SCART.



EPSON Printer Control Codes

Ian Pizer

I have experimented with using EPSON Printer ESC/P2 codes for printing from DataDesign. Most of the ESC/P2 codes are straightforward except for ESC X (m) (nL nH) which allows choice of the point size from up to 32 but the means of using the code is obscure. Here is what I have found thanks to various articles in QUANTA and QL Today:

open#3,par

bput#3,your desired codes (see below).

27,88,1,16 will give 8 point characters

27,88,1,21 for 10 point

27,88,1,24 for 12 point

By increasing the last number by steps of 4 you can get up to 32 point characters.

Example:

open#3,par

bput#3,27,88,1,48

print#3,"Some text to be printed"

will print at your printer in 24 point characters.



More THINGS... - Part 2

Jochen Merz

In the first part of this series, I (hope that I) gave you an idea what Things are. How can they be accessed, created, how does the system control them? QDOS and Minerva have no operating system calls for adding and removing Things, so a different way had to be found.

Let me start a bit technical and afterwards relax during this article I promise I will keep the technical section as short as possible. Operating system calls like memory allocation, linking in drivers etc. are all grouped under "Trap #1". This is a method of doing an operating system call in machine code. A register (D0) contains the "function code". Thing calls belong into the same group, but as the QL operating system is in ROM, it cannot be modified. Furthermore, the list in ROM cannot be extended, there was no provision for "hooks" etc. When a programmer tries to call a Trap #1 with a function code for Thing calls, the operating system returns "not implemented". (Just for the interested ones: some implementations of

SMS, mainly SMS2, contains the Thing calls as "real" Trap #1 calls). When the programmer tries to perform a Thing call and he gets a "not implemented", then he has to use the "alternative way" of doing it - which should always work. The HOTKEY System II extension does not only add improved HOTKEY facilities to QDOS and Minerva systems, it also adds the Thing calls in a slightly different way. I am not going into details now, but a programmer can easily find the Thing calls and call them with the same parameters and a function code in D0 - and they work in exactly the same way. Now the really technical bit ends.

So, in order to get Things, you need to have the HOTKEY System II loaded. It is built into SMSQ/E but needs to be loaded separately on Minerva and QDOS. If you would like to see lists of Things on your display then you should also load QPAC2. QPAC2 consists of Things and it allows you to list them, so its useful in two ways.

Type the following instruction into BASIC

```
EXEP "Things"
```

A window will appear showing you a list of Things in your system. And ... you already used a Thing called "Things". It is an executable Thing installed by QPAC2 and provides you with a job which lists the Things. So you can start programs without actually accessing any file. If you are a bit confused here, please read part 1 again - it dates back 4 months so you may have forgotten most of the theory described therein.

If we look at the Things available, you will find Things like "Button", "Exec", "Jobs", "HOTKEY", and, fairly down at the bottom of the alphabetically ordered list, "Things". This is it!! That's what you executed with the EXEP command.

Try something else, for example

```
EXEP "Pick"
```

and another window appears - giving you a listing of all jobs with open CONsoles which can be picked to the top of the pile of windows.

Next, try

```
EXEP "Button Frame"
```

... nothings happens. You may have noticed an access to your floppy and/or harddisk. "Button Frame" is not an executable Thing, therefore the EXEP command gives up on Things and tries to open a file called "Button Frame" on the currently defined program default device (can be set with PROG_USE).

Okay, let's go back into the "Things" menu and click on the word "Things" in the list. The window will change and you can see that there is (at

least) one job registered as the user of this Thing, which is, hardly surprising, "Things". Confusing? Press ESC to get back to the list and select Pick. This should be less confusing.

If you have buttons in the Button Frame of QPAC2 then you can look which job registered itself as a user - just go back to the Things list and select "Button Frame".

For the curious ones: The "Button" Thing is a Thing with Extensions you can see them in the detailed Thing window if you select "Button".

QPAC2 comes with many Things but there is one problem: most Thing names are the same in different languages, but some are not. We, the translators, (Wolfgang Lernerz for the French version and me for the German) decided independently of each other that "Button_Sleep" does not mean much to a German or French user and translated it. Big problem! From then on, it is not possible to replace an English QPAC2 by a German one and vice versa. But I guess it is too late now to change things (or Things?) and, for example, "Dateien" back into "Files". The solution is probably to have both the English and German (or French) Thing name for the same job ... but this is a different matter.

Let's create a new Thing. You want to turn XCHANGE into a Thing like the "Pick" menu? Easy, the HOTKEY System helps you doing it:

```
ERT HOT_CHP ("X", "XCHANGE")
```

You should have XCHANGE available on your current program default device. It is copied into memory and turned into an executable Thing, named XCHANGE. Check the "Things" menu, it is there!

And, even better, if you now type

```
EXEP "XCHANGE"
```

the program will be started without any disk or harddisk access! Of course, it can be started as well using the HOTKEY System II (by pressing ALT X but remember to get the HOTKEY Job going by typing HOT_GO first!).

When you remove the HOTKEY definition, the Thing will be removed as well.

The HOTKEY System II provides other functions to allow HOTKEYs to be assigned to existing Things which are not removed when the HOTKEY is removed, but this will be explained in one of the next part of this series. Also, the Things built into QPAC2 are more advanced than the Things you can create manually, but this is material for another article too.

■

Deskjet, there is a pound symbol at CHR\$(175), so all I have to do is translate code 96 to code 175.

If the character set you choose to use does not have a pound symbol, you may have to add codes here to temporarily switch to a character set that contains a pound symbol, send the code for pound in that set, then send the codes to switch back to the original character set - a useful hint sometimes.

As I occasionally correspond with readers in French or German speaking countries, I need to be able to print some accented characters, vowels especially. This also comes in useful for Welsh correspondence.

The first such accented character to be handled is a sedilla c (ç) - CTRL SHIFT 9 on the laptop on which I'm writing this in QPC. This is CHR\$(136) on the QL but CHR\$(181) on the printer.

Similarly for upper case C (Ç) CHR\$(168) on the QL. It is sent as 180 to the printer.

Translates 6 and 7 are a bodge. Since Quill does not allow double width printing, I use the key-strokes CTRL 1 and CTRL 2 to switch the printer into 5 characters per inch mode. These appear as ê (double width 1) and î (double width off), which restricts my use to these characters. Obviously, you can use any pair of characters you don't normally use - I chose CTRL 1 because it was easy to remember and because the ê character implies 'e for enlarged'

Translate 8 is another bodge to try to handle the copyright symbol © which is another QL character which can be difficult to print. Some users give up and translate it to a fairly similar looking @symbol on some printers. My approach is to translate CHR\$(127) into a c followed by a backspace to move the printhead back and overprint an upper case O, the closest I can get on my printer. You could if you really wanted to

translate © into the 3 separate characters (C and), although this can have funny effects on line lengths, because 80 character lines on screen could be 82 characters wide by the time they get printed, causing line overspill to the next line. Though as the symbol is often used in short lines such as

Copyright ©1999 Dilwyn Jones

this may not be an issue in some cases, though it does serve to illustrate how you sometimes have to think laterally to find ways of getting characters to translate satisfactorily.

Translate 9 converts CTRL SHIFT C - the É character - into an ESC character for the printer. This is useful in two ways - if you need to find a way of sneaking certain characters or functions into a document, a rare character or switching print pitch or whatever, it can be useful to directly insert characters, and have a character to correspond to ESC on screen, and É to me looks like a symbol which implies ESC! So if I wanted to double underline something, I could insert the following lines within the text, either side of the text to be underlined (again beware of wrong justification or line splits because of extra unprinted characters)

É&d2DDOUBLE UNDERLIND HEADINGÉ&d@

This can be pretty tricky stuff to work out - ignore it if you are uncertain of how it works.

Finally, in order to generate accented characters I sometimes find it useful to be able to overprint a symbol over a letter, which means I need a symbol to represent backspacing. I chose the left arrow symbol CHR\$(188) to do this - a left pointing arrow is shown in Quill to imply backspacing, so I can print a vowel followed by a backspace followed by a circumflex o←^ for example.

Things - Part 3

Jochen Merz

This time we will have a closer look at the things in your system, how they are organised in memory and how they are linked.

You will find a fairly short SBASIC program on the next

page (Listing 1) which you can type in and which will list a few more details about the Things in your system. When you run it, it will display a list of all Things which are currently available. You can see in table 2 which Things can be found in my system.

I should mention first, that hacking through a list like the

Thing list (or other lists, like the Device Driver list etc.) should be done in Assembler while the program is running in Supervisor mode. This ensures, that other jobs cannot link or unlink Things while we scan the list. We cannot do this in SBASIC, but we as long as you don't add and remove Things while the program runs

nothing will happen. And I doubt you are permanently adding and removing Things while looking at them anyway.

Things are stored in a linked list. New Things are added at the top of the list. A system variable

```
sys_thgl equ $b8
```

points to the first entry in the list. The strange PEEK_L with the two exclamation marks reads a longword from the system variables at offset hexadecimal B8 from the start of the system variables.

In "ordinary" QDOS, you could write

```
PEEK_L(HEX("280B8"))
```

instead.

The address read from this variable is a pointer to the first Thing in the Thing-list. Every Thing points to the next one, so it is very easy to scan the

; Thing linkage block

Table 1

th_nxtth equ \$00 ; long	link to NeXT Thing
th_usage equ \$04 ; long	thing's USAGE list
th_frfre equ \$08 ; long	address of "close" routine for FoRced FREe
th_frzap equ \$0c ; long	address of "close" routine for FoRced ZAP
th_thing equ \$10 ; long	pointer to THING itself
th_use equ \$14 ; long	code to USE a thing
th_free equ \$18 ; long	code to FREE a thing
th_ffree equ \$1c ; long	code to Force FREE a thing
th_remov equ \$20 ; long	code to tidy before REMOVing thing
th_nshar equ \$24 ; byte	Non-SHAReable Thing if top bit set
th_check equ \$25 ; byte	CHECK byte --- set by LTHG
th_verid equ \$26 ; long	version ID
th_name equ \$2a ; string	name of thing
th.len equ \$2c ;	basic length of thing linkage

list top to bottom. The list ends when the next pointer does not point to anything anymore but contains the value 0 instead. If you have a look at table 1, then you will see an explanation of the structure to which this pointer points.

The next three addresses will be filled in by the routine which links the Thing into the list, so you do not need to worry

about them when you create a Thing - the Usage List is a linked list of all jobs using this Thing at a given time, and the other two routines should not be touched at all. The pointer to the Thing itself needs to be filled in - nobody but you know where the Thing is actually located.

The next four routines should be provided by you so that

```
100 REMark List Things
110 :
120 listptr=PEEK_L(!!$B8)
130 chan=3:OPEN#chan,con
140 PRINT #chan;"LINK      VERS S ADDRESS  TYPE NAME"
150 REPEAT loop
160   IF listptr=0 THEN EXIT loop
170   cr_thgaddr=PEEK_L(listptr+$10)
180   PRINT #chan;HEX$(listptr,32)!
190   IF PEEK_L(listptr+$26)≠0
200     PRINT #chan;!PEEK$(listptr+$26,4)!
210   ELSE
215     PRINT #chan;"      ";
220   END IF
230   PRINT #chan;"+"((PEEK(listptr+$24),$7F)+1)!
240   PRINT #chan;!HEX$(cr_thgaddr,32);' ';
250   cr_thgtyp=PEEK_L(cr_thgaddr+$4)
260   th_nam$=PEEK$(listptr+$2C,PEEK_W(listptr+$2A)):namlen=LEN(th_nam$)
280   IF namlen<20
290     th_nam$=th_nam$&FILL$(' ',20-namlen)
300   ELSE
310     IF namlen>20 THEN th_nam$=th_nam$( TO 20)
320   END IF
330   SElect ON cr_thgtyp
340     =0:PRINT #chan;'UTIL'!th_nam$!
350     =1:PRINT #chan;'EXEC'!th_nam$!
360     =2:PRINT #chan;'DATA'!th_nam$!
370     =$1000003:PRINT #chan;'EXTN'!th_nam$!
380     =$1000004:PRINT #chan;'EXTS'!th_nam$!
390     =-1:PRINT #chan;'VECT'!th_nam$\' TH_ENTRY'!HEX$(PEEK_L(cr_thgaddr+$8),32)\\' TH_EXEC
        '!HEX$(PEEK_L(cr_thgaddr+$C),32)!
400     =REMAINDER :PRINT #chan;'      ';th_nam$!'UNKNOWN TYPE ('!HEX$(cr_thgtyp,32)!)' '
410   END SElect
420   PRINT #chan
430   listptr=PEEK_L(listptr)
440 END REPEAT loop
450 CLOSE#chan
```

Listing 1

LINK	VERS	S	ADDRESS	TYPE	NAME
000C3810		+	000C386E	EXEC	Calculator
00091E30	2.25	+	00091E62	EXEC	Sernet
000917C0	1.30	+	00100BB4	EXEC	Pic Viewer
00091760	3.31	+	00100B80	EXEC	FileInfo II thread
00091700	3.31	+	0010091C	EXTN	FileInfo II extensio
000916A0	3.31	+	001008FE	DATA	FileInfo II history
00091640	Fiv3	+	001008E0	DATA	FileInfo II database
000915E0	###0	+	001008BA	UTIL	FileInfo
00090FE0	1.00	-	000FC2D2	DATA	DATAdesign mutex
00091890	3.14	+	000918CE	EXTN	DATAdesign.engine
0008B970	A.05	+	000E6D22	EXEC	QD
0008B920	1.13	+	000DB5B6	EXTN	Scrap Extensions
0008B8D0	7.57	+	000DB81A	EXTN	Menus
00091320	1.03	+	00091360	UTIL	Button Frame
000911F0	1.03	+	000CC6C2	EXTN	Button Extensions
000911A0	1.04	+	000CF1BE	EXEC	Button_Sleep
00091150	1.02	+	000CF168	EXEC	Button_Pick
00091100	1.01	+	000CF0B0	EXEC	Hotjobs
000910B0	1.01	+	000CF098	EXEC	Hotkeys
00091060	1.04	+	000CEE78	EXEC	Channels
0008E7D0	1.02	+	000CEA44	EXEC	Jobs
0008E780	1.25	+	000CCDB2	EXEC	Files
0008E730	1.05	+	000CCAFC	EXEC	Sysdef
0008E6E0	1.02	+	000CCA26	EXEC	Rjob
0008BBD0	1.02	+	000CC928	EXEC	Pick
0008BB80	1.02	+	000CC7E6	EXEC	Wake
0008BAA0	1.02	+	000CC7CE	EXEC	Exec
0008BA50	1.02	+	000CC2C6	EXEC	Button
0008BA00	1.01	+	000CC068	EXEC	Things
000580C0	2.15	+	000B8AD2	EXTN	Jmon
0008A910	2.29	+	0008A948	UTIL	HOTKEY
0008B4B0	2.05	+	007EAEAE	EXTN	DEV
0008A1A0	3.00	+	007EAC14	EXTN	WIN Control
0008A050	3.08	+	0008A050	UTIL	DV3
00089920	2.00	+	007F64E8	EXTN	QVME
000898D0	2.09	+	0000AA6E	EXTN	Ser_Par_Prt
000581F0	2.11	+	0000A91E	EXTN	KBD
00059250	HBA	+	007F9E9E	UTIL	SBAS/QD
000591B0	HBA	+	007F9D5C	EXEC	SBASIC
00056C10	0.05	+	00056C42	VECT	THING
TH_ENTRY 00002FDA					
TH_EXEC 00002F40					

Table 2

can do whatever you like when a job tries to use, free or zap (force-free) this specific Thing. Also, a tidy-up routine can be provided (for example to unlink drivers or remove routines from an interrupt list etc.)

If the Thing can only be used from only one job at a time, then you should set the topmost bit of the non-shareable flag (marked with "-" in the listing output of the BASIC program).

Ignore the check byte, it is used for faster name comparison.

The version ID (or feature list) should be filled in by you, however.

The BASIC program reads the interesting bits from every Thing entry. In my example (table 2 again), you can see that the last Thing which has been added to the System is the Calculator (my HOTKEY definitions come last in my BOOT program!). Interesting to see, Calculator has no version number! QD, which is further down, has been LRESPRed earlier, but after QPAC2 (which links in a lot of Things ... everything from "Things" up to "Button Frame" if you read the table bottom-up.

Everything from the bottom up to "HOTKEY" is linked by the operating system SMSQ/E - this explains the order.

At the very bottom you find the special "THING" thing which defines the entry to the Thing system, but I explained this already. You will see how to use it in the future.

You can figure it out from the BASIC listing that, in my case, the Thing system variable points to \$C3810, which points to \$91E30, which points to \$917C0 ... and so on until \$56C10, which points to "0".

We also list the version number of each Thing. It is up to you what you fill in here - most people fill in four ASCII characters which define the Version. SBASIC, for example, has the version "number" HBA.

It is also possible to look at the 4 characters as 32 bits which specify "features" supported by the Thing. This means, you can't really look at "characters" but get funny patterns or strange characters, like you get when you look at FileInfo's Utility Thing.

The Thing name is reduced to 20 characters if it is longer - you need to specify the correct Thing name if you want to use or remove it, of course, so you may remove this restriction from the program.

You may have noticed that the Thing linkage contains less information than we list - the type of the Thing is not stored in the Thing linkage but in the Thing itself (th_thing points to the Thing). So we cheated a bit for now and read it from there.

We will have a look at the various structures for the various possible kinds of Things in the next part of the Things series.



foreseeable future. I have already tried to source these Eproms here in France, but have been unable to find any suitable Eproms (!).

Of course we could probably take some languages out and make them into independent LRESPR files, thus making SMSQ/E smaller for the time being, but that is just a stopgap measure - soon we will face this limit again.

I personally am not really bothered by this since I generally load SMSQ/E into RAM even on a Q60. So I would like the resellers, notably, but also the Qx0 users, to tell me whether there are really users out there that will ask for upgraded ROMs for their Qx0.

If yes, how many? If it turns out that larger Eproms cannot be built into the Qx0, where do they suggest that we go from here?

Would they be content with the upgrades to be put into RAM?

The general consensus among Q60 users on the email users group was that loading into RAM was acceptable although a minority preferred retention of the ROM chip for security and occasional backwards compatibility. Peter Graf added that suitable larger EPROMS are available:

"It is possible to use larger EPROMs, but they are 4Mbit chips (256Kbit x16) which results in 1024 KB of ROM space.

There are several manufacturers and different naming conventions. The one which is at the moment easy to get for me is the 27C4002-100 from ST.

Derek Stewart of D & D Systems added that he had had only one request for SMSQ/E on EPROM, but it would make sense to use larger chips in order to give flexibility to the operating system authors.

The "Home Thing"

Two of our news items contain references to the "Home Thing". For those of us not familiar with the term, Marcel Kilgus provided an explanation:

"Let's say you have a directory win1_myprogs_ and a basic program win1_myprogs_test_bas.

If you now execute it using

EX win1_myprogs_test_bas

you can use, within the program, the function HOME_FILE\$

to get the string

"win1_myprogs_test_bas"

i.e. "from what file was I started", using

HOME_DIR\$

you get

"win1_myprogs_"

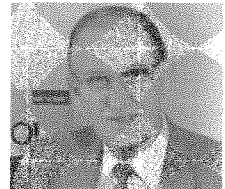
i.e. "in what directory was I started" and some more features.

This can for example be used to store configuration files along with the program, like in OPEN_IN #3, HOME_DIR\$&"test_cfg" will load the file "test_cfg" that is in the same directory as your basic program, no matter where your program is located! No manual path configuration needed."

Bill's Last Stand

Veteran trader **Bill Richardson** has announced that QL is 21 will probably be the last show he attends. In his own words:

"Bill Richardson wants to clear a lot of his stock of QL and Z88 at very low prices and will probably be at his last show unless something else turns up. The QL may be 21,



but I am 87, and its time I should pack up, having sold over 4000 QL, and 10000 Spectrums and associated bits and pieces. I have had a great time, made many friends and I am now spending more time on the fringes of politics and writing, so very sincere thanks to all the friends I have been privileged to meet over 30 years or so."

An Old Friend

Dilwyn Jones received an unexpected message from an old QL programmer, Peter Jeffries:

"Hi, I was dusting of my resume, and a random google found my name (Peter Jefferies) mentioned by you in a QL thread, in April this year.

I'm very glad the QL is still going, and sad I missed the 21st year reunion. I guess I'm a traitor moving on to video games.

P.S. latest: www.midway.com/rxpage/

Game_MortalKombat:ShaolinMonks.html

Still I've been California for 10 years now writing violent video games and enjoying the weather and missing the pubs.

If you remember, drop me an email when there is another 30th? reunion."

Dilwyn reports that Peter Jefferies wrote a number of programs for Sector Software (Taskmaster and Flashback for example) and also had connections to an early disk interface manufacturer.

Quantum Leap Software

Phoebus Dokos writes:

Please add to the news for next issue that my mailing address has changed to:

Quantum Leap Software - Phoebus Dokos

1037 Hartman Drive

Indiana, PA 15701 (USA)


```

150 REPeat change
160 e$=INKEY$(-1)
170 e=CODE(e$)
180 SELEct ON e
190   = 114 : REMark r key - increase
           red
200     red=red+1
210   = 82 : REMark Shift+r key -
           decrease red
220     red=red-1
230   = 103 : REMark g key - increase
           green
240     green=green+1
250   = 71 : REMark Shift+g key -
           decrease green
260     green=green-1
270   = 98 : REMark b key - increase
           blue
280     blue=blue+1
290   = 66 : REMark shift+b key -
           decrease blue
300     blue=blue-1
310   = 208 : REMark Up cursor key -
           increase brightness
320     red=red+1 : green=green+1 :
           blue=blue+1
330   = 216 : REMark Down cursor key :
           decrease brightness
340     red=red-1 : green=green-1 :
           blue=blue-1

```

```

350 END SELEct
360 IF red>30 : red=31
370 IF green>30 : green=31
380 IF blue>30 : blue=31
390 IF red<1 : red=0
400 IF green<1 : green=0
410 IF blue<1 : blue=0
420 print_colours
430 END REPeat change
440 :
450 DEFine PROCedure print_colours
460 AT 3,23 : PRINT "Red: ";red;" "
470 AT 5,23 : PRINT "Green: ";green;" "
480 AT 7,23 : PRINT "Blue: ";blue;" "
490 Colour=32768 + 1024*red + 32*green
   + blue
500 AT 9,23 : PRINT "Colour: ";Colour
510 WM_BLOCK 100,100,15,15,Colour
520 END DEFine

```



The HOME thing

by Wolfgang Lenerz

The latest version of SMSQE has an inbuilt support for a "Home directory thing". This may be of interest to some users, so here is a small description of it.

I – What does it do?

A - Home directory

The HOME thing implements "home directories". A home directory in this context is defined as meaning the directory from which an executable file was executed. Thus, if you have a file called `fred.exe` in a directory `win1_progs_exec_` the home directory for that file will be `win1_progs_exec_`.

The usefulness of this will vary enormously, depending on whether YOU make use of it in your programs.

B - Home Filename

The home thing also supplies what, for want of a better name, I'll call the "home filename" which is

the combination of the filename and the home directory, thus making up the complete SMSQ/E filename – in the example above this would be `win1_progs_exec_fred.exe`.

Both home directory and home filename are set up once and for all when the program starts, and are deleted when the program is removed. With one exception, they are immutable: once set, they may not be changed. They are just removed upon removal of the program itself.

C - Current Directory

The Home Thing also implements a "current directory". This is inherited from the job that is setting up the home directory (in most cases the parent job). If the calling job does not have a current directory, a copy of the home directory is used instead.

The current directory can only point to a valid directory. Within that limit it may be set/reset or otherwise manipulated by the job itself.

D - Default Directory for named jobs

Finally, there is also a default home directory for jobs that are executed through other means, perhaps through file managers that don't use the HOME thing, or, especially, through hotkeys. Due to the enormous variations that can exist in situations where jobs are executed from hotkeys, and while there is no problem when jobs are loaded from a file through a hotkey, sometimes the job code is already in memory, but no job with the name exists until the hotkey is actually pressed ("executable things"), sometimes the job executes immediately etc. In those circumstances, it will not always be possible to associate a job with a filename and directory.

It is, however, possible to set up a default home directory for jobs with a given name. When a job, for which a default directory was set up, executes, for example from a hotkey, and tries to get at its home directory, a home dir will be set up for it automatically.

Thus, whenever a job tries to find its home/current directory/file and they can't be found directly because they haven't been defined for that job, a check is also made in the default list. If the job's name is in the default list, then an entry for that job, with that default filename, is made in the home directory list.

II – Can it work for You? – SMSQ/E and QDOS

For the home directory scheme to work, the cooperation of the operating system or file manager(s) is needed: Indeed, whenever a job is executed, whoever is doing this executing must explicitly set up the home directory for the jobs that is being executed. Here, there is a difference between SMSQ/E and QDOS.

SMSQ/E now has the HOME Thing built in, and also support for it (see below for QDOS). Typically, on an SMSQ/E system, jobs will be started up through the EX(ec) command variants, through filemanagers such as QPAC II or through FileInfo. Finally, SBasic programs may also be loaded:

A - The EX(ec) etc command

Support for the Home Thing is built into the OS as of SMSQ/E version 3.11. Whenever you use the EX commands, the home directory for the job to be EXecuted will be set up automatically. This support does not exist on QDOS machines, since, obviously, the EXEC command itself had to be modified, which is not possible under QDOS.

B - QPAC II and other file managers

QPAC II has already been altered to take the new home thing into account. All file managers will need to be changed to support the home directory. If you are a programmer and have programmed a file manager, further information is given below, showing you the code that needs to be implemented for this. If a filemanager is in compiled basic (e.g. DiskMate) no further action will be necessary under SMSQ/E since the EX commands in SMSQ/E will do whatever is necessary.

C - FileInfo

Thierry Godefroy has modified FileInfo II to use the Home Thing. For the record, I have modified my own FileInfo (the initial FileInfo) to use the home thing, if present.

D - Basic

Under SMSQ/E, whenever you (q)load/(q)merge a basic program, the home directory for that basic program is set to the file just loaded. Thus basic is again an exception – it is the only job for which the Home directory may change.

E - QDOS and the HOME Thing

There is a stand alone version of the HOME thing for QDOS users (which can be downloaded at <http://www.scp-paulet-lenerz.com/14mljkl24/wolf/download/>).

Thus, QDOS systems can also profit from the home directories set up from QPAC II and FileInfo, but support for the HOME thing through the EX and LOAD commands will be non-existent, since that requires a change in these commands. The same is true for filemanagers that are compiled basic.

III – How to load the HOME Thing

The HOME thing is already present in SMSQ/E v. 3.11 onwards and doesn't need to be loaded.

On Qdos, use:

```
a=RESPR (file_length)
LBYTES <device>_home_bin,a
CALL a
```

or the LRESPR variants if your system has them.

IV – Using the HOME Thing

A - From SBasic

There are several new SBASIC keywords for this.

1 - Get the home directory

```
result$ = HOME_DIR$(job_id)
```

This function returns the home directory for the job given as `job_id`. To avoid programs stopping with an error if for some unimaginable reason the home directory cannot be found this function returns an empty string if that error happens.

The job ID is optional, in that case -1, meaning the current job, will be assumed.

E.g.:

```
...
100 define procedure init
110   mydir$ = HOME_DIR$
....
```

2 - Get the home filename

```
result$ = HOME_FILE$(job_id)
```

Same as for the home directory, but for the home filename.

3 - Get the current directory

```
result$ = HOME_CURR$(job_id)
```

Same as for the home directory, but for the current directory.

4 - Default names

```
HOME_DEF job_name$, file_name$
```

This sets a default filename for a job with the name given as first parameter. This is useful for "executable things", where no filename is readily available, or for file managers that haven't integrated calls to the home thing. Please refer to the section I-D above for more information on this.

With this keyword, you set up the default job name and filename that is to be used for the home/current file/dir.

Please note that the `file_name$` parameter must indeed be a `FILENAME`, not a directory name.

Example:

```
HOME_DEF "Sbasic", "dev1_sbasic_test_bas"
```

5 - Get the version of the HOME thing

```
result$ = HOME_VER$
```

B - From machine code

The thing can of course also be called from machine code. It implements various extensions – it is an extension thing.

There is some ready to use wrapper code in the SMSQ/E source tree, namely in the file "util_gut_home_asm". If you still want to go the manual route, here is the documentation for the different extensions:

To use them, in short, first you `USE` the thing (A0 = thing name, D2 = extension). You might want to make sure you use a call that returns the pointer to the thing linkage base in A2 and a pointer to the thing in A1 as these will be expected when calling the call routine. (If you have access to the SMSQE sources, a good vector for this is `gu_thjmp`).

The name of the thing is, imaginatively, "HOME", and the names of the individual extensions will be given below.

After you have used the thing, you then call the `thh_code` routine of the thing with A2 pointing to the linkage and A1 to a parameter list.

Each extension thus has its own parameter list. They are explained below for each extension. An example for using the thing from machine code could be as follows, to `SET` the default name/dir (this uses the `gu_thjmp` routine from the SMSQE sources, which returns the correct values in A1 and A2).

It is presumed for this example that, on entry, D1 contains the job ID of the job for which the directory/file is to be set, that the job name, for which a default is to be set up can be found at label called 'jobname' and that the filename for this job can be found at a label called 'filename'.

```
homereg    reg    a0-a4/d0-d3
d1stak     equ    4
           ; where D1 is on stack
```

```

setdflt
    movem.l homereg,-(sp)           ; keep my regs
    lea     home_name,a0           ; point to name of thing
    moveq   #-1,d3                 ; wait forever
    moveq   #-1,d1                 ; I will use the thing
    move.l  #'SETD',d2             ; extension in thing to use
    moveq   #sms.uthg,d0           ; use thing
    jsr     gu_thjmp               ; on return A2= ptr thg header, a1 to thg
    tst.l   d0                     ; ok?
    bne.s   no_thg                ; no, ignore
    move.l  a1,a0                  ; pointer to thing (!!!)
    sub.l   #16,sp                 ; get some space
    move.l  sp,a1                  ; and point to it
    move.l  #$c1000000,(a1)        ; thp.call+thp.str
    lea     jobname,a4             ; point to jobname to set
    move.l  a4,4(a1)               ; set pointer to this string
    move.l  #$c1000000,8(a1)       ; thp.call+thp.str
    lea     filename,a4           ; point to file/dirname to set
    move.l  a4,12(a1)              ; set pointer to this string
    jsr     thh_code(a0)           ; call extn thing
    add.l   #16,sp                 ; reset stack
    lea     home_name,a0          ; now free thing, ignore error on call
    moveq   #sms.fthg,d0
    moveq   #-1,d1
    jsr     gu_thjmp               ; free thing
no_thg
    movem.l (sp)+,homereg          ; ignore error
(...)

home_name
    dc.w    4, 'HOME'

```

The extensions are as follows:

GETH
GETF
GETC

Respectively get the home directory, home filename and current directory.

entry	exit
D0	0 or error
D2	buffer size needed (errorng)
A1 pointer to parameter list	preserved
A2 pointer to thing linkage	preserved

The parameter list is as follows:

0(a1)	long word	job id of job for which info is to be gotten
4(a1)	word	\$A100 (corresponding to thp.str+thp.ret)
6(a1)	word	length of buffer for return string
8(a1)	long word	pointer to buffer for return string

The routine will return 0 if no error occurred, erritnf if the job with the given ID doesn't have a home directory and errorng if the buffer is too small for the entire directory/filename. In this latter case, the routine will not touch the given buffer but just return the needed size in D2.

SETD

set the default directory for a given name.

entry	exit
D0	0 or error
A1 pointer to parameter list	preserved
A2 pointer to thing linkage	preserved

The parameter list is as follows:

0(a1) long word	\$C1000000 (corresponding to thp.str+thp.call)
4(a1) long word	pointer to string for jobname
8(a1) long word	\$C1000000 (corresponding to thp.str+thp.call)
12(a1) long word	pointer to string for filename

V – Setting up a HOME Directory

Normally, jobs should not try to set up a home directory for themselves. This should be left to the system/filemanager. When a job is started with the SMSQ/E EX, EW or any of the similar commands, this is done automatically. However, filemanager writers may be interested in this info.

Since there can only be one home directory for a job and since that can only be defined once, the keyword will give an error if the home directory is already set for this job. Otherwise, this keyword will set the home directory, the home file and the current directory.

This keyword exists mainly for testing purposes.

A - From Sbasic

HOME_SET job_id, device_and_file_name\$

Set the home directory, home filename and current directory. You pass the thing the job ID of the job for which this is to be set up and the entire filename, including the device and directory. The thing extracts the home directory from the filename. If you want to set up the home directory for the current job, you may pass -1 as parameter.

B - From Machine Code

Please read the general rules on how to use the thing from machine code (section IV, above). The extension to use here is SETH. to SET the home name/dir (this uses the gu_thimp routine from the SMSQE sources, which returns the correct values in A1 and A2).

It is presumed here that, on entry D1 contains the job ID of the job for which the directory/file is to be set and that the filename for this job can be found at a label 'filename'.

SETH

This sets the home directory.

entry	exit
D0	0 or error
A1 pointer to parameter list	preserved
A2 pointer to thing linkage	preserved

The parameter list is as follows:

0(a1) long word	job id of the job for which this is set
4(a1) long word	\$C1000000 (corresponding to thp.str+thp.call)
8(a1) long word	pointer to string for entire filename

The routine will return 0 if everything went OK else any error from the memory allocation routine or errfdiu if the job with this job ID already has a home directory set up.

The following is an example of a routine that can use this to SET the home name/dir (this uses the gu_thjmp routine from the SMSQE sources, which returns the correct values in A1 and A2).

It is presumed here that, on entry D1 contains the job ID of the job for which the directory/file is to be set and that the filename for this job can be found at a label 'filename'.

```
homereg
    reg      a0-a4/d0-d3
    distak   equ 4                      ; where D1 is on stack

sethome
    movem.l  homereg,-(sp)              ; keep my regs
    lea      home_name,a0              ; point to name of thing
    moveq    #-1,d3                    ; wait forever
    moveq    #-1,d1                    ; I will use the thing
    move.l   #'SETH',d2                ; extension in thing to use
    moveq    #sms.uthg,d0              ; use thing
    jsr      gu_thjmp                  ; on return A2= ptr thg header, a1 to thg
    tst.l    d0                        ; ok?
    bne.s    no_thg                   ; no, ignore
    move.l   a1,a0                     ; pointer to thing (!!!)
    move.l   distak(sp),d1             ; get job ID back
    sub.l    #12,sp                    ; get some space
    move.l   sp,a1                     ; and point to it
    move.l   d1,(a1)                   ; insert ID of job
    move.l   #$c1000000,4(a1)          ; thp.call+thp.str
    lea      filename,a4              ; point to file/dirname to set
    move.l   a4,8(a1)                  ; set pointer to this string
    jsr      thh_code(a0)              ; call extn thing
    add.l    #12,sp                    ; reset stack
    lea      home_name,a0              ; now free thing, ignore error on call
    moveq    #sms.fthg,d0
    moveq    #-1,d1
    jsr      gu_thjmp                  ; free thing
no_thg
    movem.l  (sp)+,homereg              ; ignore error
(...)

home_name
    dc.w     4,'HOME'
```

Finally, I'd like to point out that, whilst I have divised the home thing initially, Marcel Kilgus later recoded much of the code and also implemented some better ideas of his.

Special QUILL printer drivers

by Dilwyn Jones

Much maligned, Quill still has its uses. Fancy using it to transfer text to a PC? How about generating some HTML code? Or exporting Quill DOCs as simple plain text?

This is a set of five printer_dat files for Quill or Xchange intended to allow Quill to perform 5 special functions:

1. Print to HP Deskjet printers (also works on some Laserjets) (**HPDJ_printer_dat**)
2. Generate simple plain text files with QL-style linefeed end of lines (**QLplaintext_printer_dat**)

3. Print to a plain text file with carriage returns added to end of lines for transfer to PCs, and translation of character codes below 128. (**QLtoPCtext_printer_dat**)
4. Print the text file as a simple HTML file (**HTML_printer_dat**) although this driver only works on the Xchange version of Quill (code sequences too long for standard QL Quill and cause an error due to failure to read the printer_dat) and may need to be renamed to xchange_dat.

Precisely each second every satellite starts transmitting its sequence of signals. Among them is its own unique identifying signal; this is the PRN or Pseudo Random Number, sent as a long binary number, that I also did not understand then. A good receiver will have a number of processing channels, and, after the initial start up period to find out the time, and what satellites are in view etc., each channel slides (in time) a replica of one of the codes until a correlation is found. So, by this the receiver identifies the satellite, and the slide time gives the delay in receiving its signal, that is the time taken for the signal to travel; divide by the speed of propagation, and you have the distance from the satellite.

The signal, whizzing along at around 300,000 kilometres per second or 30 cm every nanosecond, takes about 68 milliseconds to arrive from a satellite overhead, and from one on the horizon, about 86 milliseconds. Measuring this travel period accurately enough obviously needs a very precise knowledge of the time the radio signal set off, and each satellite has a number of atomic clocks on board, again monitored and controlled from the ground so that they are all synchronised within the satellite and throughout the swarm. However it is difficult to make a correspondingly accurate clock reasonably cheaply so the receiver has to synchronise its own quartz crystal controlled clock to the GPS time, as I will describe in a moment. A pay-off from this is that you can get a very accurate time check – potentially many times more accurate than the MSF transmissions used in radio-controlled clocks, which, with a 60kHz carrier, can give a signal correct only to about one millisecond.

To get a fix, four satellites are needed: the distance from the first defines a spherical 'surface' on which the receiver (strictly its antenna) must be, somewhere. The same applies to a second satellite, so the antenna is now known to be located on the circle where this 'sphere' cuts the first, since it must be on both. A third sphere cuts this circle in two places, and, assuming that the times of travel of the signals, and hence the distances, are accurate, a fourth sphere should pass through just one of these points, giving a unique position relative to the satellites' frame of reference.

With the receiver's clock unadjusted, this fourth sphere will miss the point established by the first three; but they can then be made to coincide by calculating a small correction, and adding it to (or subtracting it from) each measurement: this correction is how much the receiver clock is fast or slow on satellite time. By applying it, the ground receiver's clock is synchronised with satellite time. A succession of these corrections over a few seconds will also give the rate of gain or loss in the receiver's clock.

However the measurements and calculations are not precise, so, in subsequent fixes, even with the corrected clock, the 'spheres' will not meet exactly at the same point. More satellites' distances are measured giving a spread of intersections over a small space called a "resection". A calculation from the points defining the resection gives the most probable position (MPP in the jargon) and a better, average, correction for the clock; also the area of the resection is a measure of the additional inaccuracy of the fix, called "Dilution Of Position" or DOP, above that expected from the inherent tolerances of the system itself. It is a remarkably clever and ingenious system.

Executable Things - Part 1

by George Gwilt

I describe here how and why I came to use executable Things.

Why

Although setting PROGDS to the directory containing executable programs can make it easier to load them by omitting the directory when typing the EX command it sometimes fails. In my case this is because I have set PROGDS to WIN1_C68_ instead of the usual WIN1_SYS_ and then tried to access NET_PEEK from WIN1_SYS_ by typing EX NET_PEEK only to see the message "not found". When this happened several times I recalled that I accessed QD by typing EXEP and that this worked whatever the value of PROGDS. QD is a Thing and is accessed by the keyword EXEP.

I then determined to do the same with NET_PEEK. However, my program would have to be available to those who did not have Things. This meant that I had to arrange for NET_PEEK to be capable of being either loaded by EX or being set up as an executable Thing by LRESPR.

How

First of all we have to know enough about Things for our purpose.

Things are kept in a linked list. Each Thing has an associated linkage block as follows:

Item	Position	Meaning
TH_NXTH	\$00	→ next linkage block
TH_USAGE	\$04	USAGE list
TH_FRFRE	\$08	code called when force remove frees a thing
TH_FRZAP	\$0C	code called when thing owner is removed
TH_THING	\$10	→ Thing itself
TH_USE	\$14	code to USE the Thing, or 0
TH_FREE	\$18	code to FREE the Thing, or 0
TH_REMOV	\$1C	code to force FREE the Thing, or 0
TH_NSHAR	\$20	byte set if Thing not shareable
TH_VERID	\$26	version ID
TH_NAME	\$2A	name of Thing

All these are long words except TH_NSHAR which is a byte and TH_NAME which is a string. For all the executable programs I have made into Things I have set the whole linkage block to zero apart from TH_THING, TH_VERID and TH_NAME. The item TH_NXTH must not be zero of course, but it is set by the appropriate linking code as we will see later.

We must now have a look at the Thing itself. I mean by that the Thing to which TH_THING points.

All Things have a header which starts with:

THH_FLAG	\$00	"THG%"
THH_TYPE	\$04	type of Thing

The type can be -1 to 4 with various meanings. Ours is 1, which means "executable code"

Our Thing continues:

THH_HDRS	\$08	offset to code
THH_HDRL	\$0C	size of code
THH_DATA	\$10	dataspace
THH_START	\$14	offset to start of program or 0

The offsets here are all measured from the address of THH_FLAG.

The information given here in the Thing is enough to set up our program as a Job.

The QDOS software for creating a job, MT_CJOB with Trap #1, asks for the length of code and the length of dataspace and also allows an explicit start address to be given. This start address can be to a single version of the code, thus enabling several versions of the program to be running simultaneously with only one copy of the code.

MT_CJOB uses the information it has been given to set up an area in RAM. This starts with a \$68 byte header which is followed by an area equal to the sum of the lengths of code and amount of dataspace requested. The header contains a pointer to the start of the program. This points either to the area immediately following the header, or to the start address given to MT_CJOB if this is not zero.

One item relating to the program is not held within the \$68 byte header. This item is the program's name. I imagine the reason for this is that names are of indeterminate length so that the header would have had to contain, for the name, a fixed space which will be either too large, which is wasteful or too small which is restrictive. The compromise solution was to set the name 8 bytes after the end of the header which, of course, puts it inside the code, which immediately follows the header. The name is preceded by a word containing the marker \$4AFB.

A normal program will thus start with, say, a short branch followed by a long word which is followed by

\$4AFB. Then comes a string which is taken as the name of the program which is what will be shown, for example, if you type JOBS.

The program NET_PEEK was written to be re-entrant. That is, it does not alter its own code. This means that multiple copies of the code are not needed for multiple versions of the program.

We can now determine what the Thing linkage block and the Thing itself should contain. Before detailing these contents I should state how the linkage block and the Thing will be set up. This will be done by adding a piece of code to the start of the program. We will CALL this code. In other words we will LRESPR the program to create the Thing. As a consequence the whole of the program's code will have been loaded into a space allocated from the heap.

There is just one more element to be introduced before commenting on the code itself. It is this. Although I determined that NET_PEEK should become a Thing, as I indicated above, I realised also that some machines may not have the Thing code and that therefore NET_PEEK should remain capable of being started by EX. This means that the initial code for NET_PEEK has to decide whether it has been invoked by EX or by LRESPR. Furthermore, it is necessary for safety that a Thing should be set up from master BASIC and not a daughter basic in SMSQ/E. This is because a daughter basic can be removed but the master BASIC cannot.

Comments on the Code below

Initial Code

Whether the program is started by LRESPR or EX the first instruction obeyed is that at HEADZ. This is a normal start of a program with a branch round the name. In this case we go to START where we have to decide how we were born.

We use MT_INF to put our Job ID in D1.L. This will be zero if we were LRESPRd from master BASIC. If so we jump to SET_THING. Otherwise a check is made to ensure that we were not LRESPRd from a daughter BASIC. Each BASIC in SMSQ/E has "SBAS" in -4(A6). Any other non zero ID is taken as arising from EX, in which case we branch to the real start of the program at STARTA.

SET_THING

We set A1 pointing to the linkage block and A0 to the Thing itself.

The software linking in the Thing requires the items from TH_THING to TH_NAME to be filled in, so that is what we do. Most of them are zero for we don't need code to use, free, force free or remove the item.

The Thing itself is filled in with, effectively, the instructions on how the executable program is to be set up when the Thing is called. Thus the code to be set is determined by THH_HDRS, which is set to HEADZ. THH_HDRL is set to PRS - HEADZ, which is the length of code from the start of the program to the end of its name. This is all the "code" we need. The dataspace, \$3300 bytes, is set in THH_DATA. Finally, the real start address, STARTA, is set in THH_START.

If the program were not re-entrant the "code" would have to be the entire program, so the length would have to be set accordingly in THH_HDRL. Also the value of THH_START would have to be zero. If this had been done with NET_PEEK, the start would have been at HEADZ and not STARTA. Would this have worked? Yes it would, since the initial code would have decided that the program had not been LRESPRd and a branch would have been made to STARTA as needed.

Linking the Thing

In the operating system SMS2 there are various Trap #1 routines, D0 = \$26 to \$2C inclusive, which all relate to the Thing system. Alas none of them are available elsewhere, including SMSQ/E. To enable these to be used when these Trap #1s are not available there is, at the end of the list of Things a Thing called THING.

THING contains entries to two vector routines, THH_ENTRY and THH_EXEC. We are interested in the first, since, by jumping to it as a subroutine with the registers set up as for the missing Trap #1s these are implemented.

Code to access the vector is given at GU_THVEC. We use this with D0 = 8. To link in our Thing we need its address in A1, which it is, and D0 set to \$26 which we do.

When this is done we return to BASIC.

Code

```
HEADZ      BRA.S      START
           DC.L        0
           DC.W        $4AFB
           DC.W        N_END-NAME
NAME        DC.B        "NET_PEEK V",VERSION," July 2007"
N_END      DS.B        0
PRS

START      MOVEQ       #MT_INF,DO                ; Set the . .
           TRAP        #1                        ; . . JOB ID . .
           TST.L       D1                        ; . . in D1.L
           BEQ         SET_THING                 ; Master BASIC
           CMPI.L      #"SBAS",-4(A6)           ; Daughter BASIC? . .
           BNE         STARTA                    ; . . No so it was EX
           MOVEQ       #-19,DO                   ; We cannot set the . .
           RTS         ; . . Thing from here

SET_THING
           LEA         TLINK,A1                  ; address of linkage block
           LEA         N_THING,A0                ; address of Thing itself
           MOVE.L      A0,$10(A1)                ; set address of Thing

; We must link in this Thing

           MOVEQ       #8,DO                     ; THH_ENTR
           BSR         GU_THVEC                   ; Get Thing vector to A4
           BNE         OOPS_1      ————
           MOVEQ       #$26,DO                   ; Link in . .
           JSR         (A4)                       ; . . the Thing
           TST.L       DO
OOPS_1     RTS         ; Back to BASIC

*

; This is the NET_PEEK Thing

N_THING    DC.L        "THG%",1                  ; Marker and type 1
           DC.L        HEADZ-N_THING             ; Offset to "code"
           DC.L        PRS-HEADZ                 ; Size of "code"
           DC.L        $3300                     ; Dataspace
           DC.L        STARTA-N_THING            ; Offset to start of program

; This is the linkage block

TLINK      DCB.W       19,0                      ; Zeroes up to Version
           DC.L        "1.00"                   ; Version
           HED1        <"NET_PEEK">,TLINK1       ; Thing name

; NB HED1 is a macro setting the string < . . > with label TLINK1

; Here is the actual start of the program

STARTA     LEA         (A6,A5.L),A7              set STACK
           LEA         (A6,A4.L),A6              -> DATA SPACE
```

. . . Program continues . . .

```
; Routine to get Thing Vector to A4
; At entry DO.W = $08 for TH_ENTRY routine
;               = $0C for TH_EXEC routine

VECR    REG      D1-3/D7/A0                ; Registers to keep . .
GU_THVEC MOVEM.L  VECR,-(SP)                ; . . Keep them
        MOVE.W   DO,D3
        MOVEQ    #MT_INF,DO                ; Get system information
        TRAP     #1
        MOVE.W   SR,D7                    ; Preserve status register
        TRAP     #0                      ; Supervisor mode
        MOVE.L   SV_THINGL(A0),D1          ; address of thing linkage
        BEQ      NOT_THERE ----->
        MOVEA.L  D1,A0
THVEC_LP MOVE.L   (A0),D1                  ; Next block . .
        BEQ      FOUND                    ; Last block - found
        MOVEA.L  D1,A0                    ; Reset A0 . .
        BRA      THVEC_LP                  ; . . and try again

NOT_THERE MOVEQ   #-7,DO
        BRA      TH_RT

FOUND     MOVEA.L  $10(A0),A0                ; Pointer to THING Thing
        CMPI.L   #-1,4(A0)                ; Is it type -1? . .
        BNE      NOT_THERE ----->        ; . . 'fraid not
        MOVEA.L  (A0,D3.W),A4              ; set the vector to A4
TH_RT     MOVE     D7,SR                    ; return to user mode
        MOVEM.L  (SP)+,VECR                ; replace registers
        TST.L    DO                        ; set condition codes . .
        RTS                                     ; . . and return
```

Directories

by David Denham

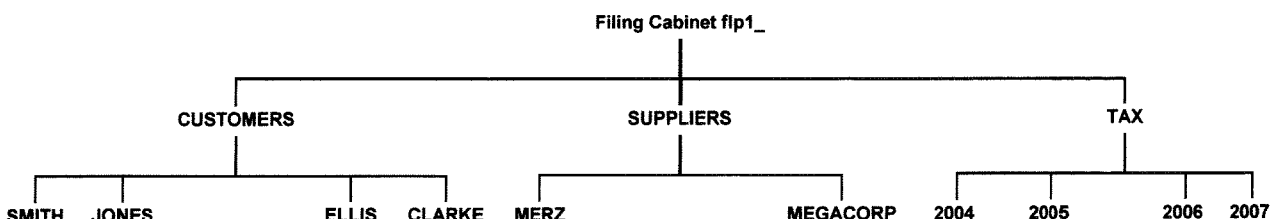
The Toolkit 2 manual says that "a 'directory' is where the system expects to find a file" and goes on to explain that this can be as simple as a drive name like MDV1_ or FLP2_

To help us understand exactly what a directory is, we need to draw a comparison with paper based files.

Suppose our office has a few filing cabinets. All our paper files can be placed in any of these filing cabinets. One is called MDV1_, another is called FLP1_ and another is called WIN1_. These correspond to our microdrives, floppy disk drives and a hard disk respectively. These filing cabinets all have drawers and each drawer contains a

number of folders into which the loose bits of paper are grouped. It's obviously convenient to have related files together. For example, one folder per company we deal with.

Grouping papers together like this can make it easier to find something. We have reserved a drawer labelled CUSTOMERS for all our customer correspondence, and another drawer labelled SUPPLIERS for all our suppliers. A third drawer is labelled TAX and contains our tax correspondence for each year in its own folder. We could use a diagram like this to represent the filing cabinet:



If you wish to remain in mode 16, but wish to switch to the QL colours temporarily, you can opt to use the QL colour definitions with the command COLOUR_QL, which will use QL colours 0 to 7 and the stipples thereof.

You will find that QemuLator always goes into full screen mode when using SMSQ_GOLD. You can use ALT TAB in Windows to switch in and out of Windows to other applications as required. If you need to return to QemuLator's menus (e.g. to change one of the 8 drive assignments) you can press F12 to toggle between full screen and QemuLator menus, but it does tend to kill SMSQ/E, so you have to go into the "QL" menu, and use the "STOP" command to stop the emulation, then restart it in the usual way. I'm not sure if

this is a bug or just a limitation in how QemuLator can work.

QemuLator can be downloaded as a shareware program from Daniele Terdina's website at:

www.terdina.net/ql/q-emulator.html

The free trial version limits itself to running at QL speed and with only 128K or 384K RAM. This version does not run SMSQ/E at all. Register with the author (registration costs 40 U.S. dollars) and you get a code to convert it to a much upgraded version with many extra facilities such as level 2 subdirectories, access to QXLWIN files, TCP/IP driver, faster loading of BASIC programs and much faster operation and support for much larger RAM.

Executable Things - Part 2

by George Gwilt

Previously I described the Thing THING. This is the last Thing in the Thing's linked list. Its purpose is twofold. First it allows those who are not using the operating system SMS2 (and I suspect that is most of us) to use SMS2's extra Trap #1 routines. These routines all relate to Things. I have already mentioned the one used to link a new Thing into the list.

The second purpose of THING is to enable the execution of an executable Thing. I found the need to investigate this routine for the following reason. An executable Thing will normally be started by the keyword EXEP. This is the equivalent of starting an executable program by EX. Thus I can type:

```
EX NET_PEEK
```

or

```
EXEP NET_PEEK
```

to achieve what, to all intents and purposes, is the same result.

However, EX allows a user to put information on the program's stack when it is started. So:

```
EX NET_PEEK;"Hullo NET_PEEK"
```

would set the string "Hullo NET_PEEK" onto NET_PEEK's stack.

So would:

```
EXEP NET_PEEK;"Hullo NET_PEEK"
```

So far so good.

EX goes further though. It allows channel ID's to be put on the stack too.

At this point I should describe what this looks like to a programmer.

From the earliest days of the QL every job created by MT_CJOB had its stack pointer reduced by 4 from its highest position. Provided that the dataspace requested was at least 4 this sets two zero words on the stack. It was envisaged that when each program started it would have on its stack a word giving the number of channels followed by that number of IDs. After that would be a word giving the length of a following string. The two zero words of course signify no channels and no string. When TK2 appeared it contained EX which replaced the original EXEC and allowed both channels and a parameter string to be set.

Thus:

```
EX NET_PEEK,ram1_data,#3,#4;"Hullo NET_PEEK"
```

would put on NET_PEEK's stack 3 followed by three IDs then 14 followed by the string. The first ID would be for the channel opened for ram1_data with NET_PEEK as the owner. The owner of the other channels, #3 and #4, would be the BASIC from which the EX command had been given. When I tried this with EXEP I discovered that it allowed no channels of any sort. At this stage I decided to investigate what was probably at the back of EXEP That is the second vector routine of THING, TH_EXEC.

Here is what TH_EXEC says it expects in its registers:

- D1.L The ID of the owner
- D2.L priority*2^16 + timeout (0 for 'EX' or -1 for 'EW')
- A0 pointer to the Thing's name
- A1 pointer to the parameter string

At first sight this is not encouraging for one hoping to set channels IDs on the Thing's stack. However, investigation showed that the "parameter string" was actually the entire contents of the stack. That is the string contained the number of channels followed by the IDs then the parameter's length followed by the parameter. So TH_EXEC can put channel IDs to an executable Thing's stack after all.

By now I had determined to produce a keyword EXEG which I would use in place of EXEP when I wanted to set channel IDs on a Thing's stack. I would type:

```
EXEG NET_PEEK,ram1_data,#3,#4;"Hullo NET_PEEK"
```

to put the IDs and string on the stack.

I would have to write code to examine the parameters of EXEG and from this make up the contents of the Thing's stack to be presented to TH_EXEC as the "parameter string" to which A1 points.

There is a snag, however. When EX produces the ID for a file, such as ram1_data in my example, it sets the owner of the ID as the program it is executing. Since the IDs have to be produced before TH_EXEC is called this would not be possible with EXEG.

There is at least one way round this difficulty. The real reason why a channel ID should have the executable Thing as owner is that the channel will automatically be closed when the job stops. It would be possible to have an intermediate job which would both be the owner of the channel and would also call TH_EXEC. If this intermediate job removed itself when the executable Thing did, then the effect would be as if the Thing were the channel's owner.

Well, yes, this works but is rather messy. In the end I decided to forget all about TH_EXEC and tackle the problem directly.

The general idea was to create the Thing's job, then examine the parameters of EXEG and load the result onto the Thing's stack. But before we create the job we need to know the number of bytes which the channel IDs and the parameter string will need, because this has to be added to the dataspace defined in the executable Thing.

There are nine steps. The code for these will be given later but it is worth a preliminary comment on each of them here.

1. Find the number of channels and length of parameter string

We have to find this information from the parameters presented to EXEG. Information about all parameters is set in 8-byte blocks. The first of these is found at (A3,A6.L) and the end of the last at (A5,A6.L). These addresses point to the last section of the Name Table in which the parameter information is stored.

The number of parameters can be calculated as

$$(A5 - A4)/8$$

We need to know the format of the 8-byte entry for a parameter. It is:

Word	Word	Long
Type	-> Name Table	-> Value

Type

Type is a mixture of a code giving the type of parameter, a code giving the type of separator which follows the parameter and a third code indicating whether or not the parameter is preceded by hash (#).

The second of the three codes is a number from 0 to 7 held in bits 4 to 6 of the Type word. All we need to know is that number 1 is a comma (,) and number 2 is a semicolon (;).

The third code is in bit 7. Zero means no hash and 1 means there is a hash.

The remaining bits in the Type word which indicate the type of parameter include:

\$0002 for undefined floating point and

\$0201 for string variable

-> Name Table

The second word is a number giving the entry in the Name Table of the parameter's name or -1 if there isn't an entry. The nth entry will be indicated by the number n-1.

It is important to realise that this applies only to parameters. A normal Name Table entry also has a pointer in this position, but its value is the offset to the name in the Name List.

Jan Jones says in QL SuperBASIC - The Definitive Handbook,

"... an entry is made at the top of the nametable for all the actual parameters. Such entries are not permanent, they only exist for the duration of the procedure. If the parameter is a simple variable, the new entry is a copy of the entry for that name with the pointer to the namelist replaced by a pointer to the original nametable entry."

-> Value

If there is a value for the parameter it can be found from the long word pointer which is the offset from the start of the variables area.

The parameters presented to EXEG consist of the Thing name, which must be present, followed by the channels and then the parameter string for the Thing itself. If t is the number of parameters following the Thing name, then the number of channels is t if there is no parameter string and $t - 1$ otherwise. We can tell if there is a parameter string by examining the second last separator, if $t > 0$. If $t = 0$, then there are no channels and no parameter string. If the second last separator is a comma there is no parameter string. If the separator is a semicolon there is a parameter string.

If there is a parameter string, its length is found by examining its value in the variables area.

2. Find the Thing's Name

The first parameter to EXEG is the Thing's name. This might be presented with or without quotes. If it is without quotes it will be taken by the system as an undetermined floating point variable. In the first case its type code will be \$0201 and in the second \$0002.

In the first case the name can be put onto the maths stack by the vector CA_GTSTR. In the second case the name has to be extracted from the Name List and put on the maths stack. This is done by the subroutine n_to_stack. This finds the position in the Name Table for the name we want. In this case the second word in the 8-byte entry gives the offset in the Name List of the name we want.

After a check that there is enough room on the maths stack, by using BV_CHRIX, the routine copies the name to the stack from the Name List.

3. Find the Thing's Linkage Block

The Thing's linkage block is found by scanning the chain of linkage blocks until one is found with the Thing's name. There is a special subroutine, cp, which compares the Thing's name, which is on the maths stack, with the name in the linkage block. We could have used the vector UT_CSTR but this would have entailed subtracting A6 from A2 since both A1 and A2 have to be relative to A6 and only A1 is. This is unsafe since A6 can change at any time. So I wrote the subroutine cp instead.

4. Find the Thing

The address of the Thing is at long word \$10 from the start of the linkage block.

5. Create the Thing's Job

We need to set:

D1.L = the owner job : We set that to 0 for "independent"
D2.L = length of code : From the Thing at \$C(A0)
D3.L = dataspace : From the Thing at \$10(A0) plus space for the channel
; IDs and parameter string
A1.L = start address : From the Thing at \$14(A0)

6. Adjust the Thing's stack and add the code

When the job is successfully created we first adjust the job's stack to make room for the IDs and string. The address in A0 set by MT_CJOB is the address immediately following the \$68 byte header set up for the job. The position for A7 in this header is thus at -12(A0)

We then fill in the code from the address given in the Thing to the address in A0 set by MT_CJOB.

7. Put the channel IDs on the stack

The number of channel IDs to be put on the stack is in the top word of D5 (nc). This is set on the stack. We then examine nc parameters and set the appropriate IDs on the stack. For each parameter there are two possibilities. Either the channel is explicit, heralded by hash (#) or it is the name of a file which must be opened. We determine whether it is hash or not by testing bit 7 of the second byte of the parameter block. If this bit is set we read in the integer parameter and use it to access the BASIC channel block which contains the ID we want.

Each BASIC channel block is \$28 bytes long and the first one is at the address BV_CHBAS(A6) relative to A6. The ID is the first long word in the block.

If we find that there is no hash we examine the type of parameter to see whether it is a proper string by testing the first byte of the parameter information.

This will be non zero for a string. Actually the type word should be \$0201 for a string and \$0002 otherwise. If the type is in fact something else we rely on an error being signalled when we attempt to put the string on the maths stack.

When the string has successfully been read in we try and open the file with the name given. If this has not worked, we try again having added DATAD\$ to the start of the string. Note that we set D3 = 0 for the OPEN. This is the "old exclusive" form of OPEN which is how EX opens files. The owner of the channel is the job we have just created. Again this is what EX does.

DATAD\$ is added by the subroutine ad_dat. This checks that there is space on the maths stack for the addition of DATAD\$. It then subtracts the rounded up length of DATAD\$ from the maths stack and places DATAD\$ there. If DATAD\$ is of even length the new filename is set. Otherwise we must move the old filename back by one byte so that it is properly joined up with DATAD\$.

8. Put the Parameter list on the Stack

The rounded up length of parameter list to be set on the stack is now in the top word of D5. If this length is zero there is no parameter.

If there is a parameter we ignore its type code and attempt to set it on the maths stack by CA_GTSTRG which will signal an error if it is not a string.

9. Activate the job

The job is activated by a call to MT_ACTIV which requires the job ID to be set in D1, the priority in D2 and timeout in D3.

We set priority to 16 and timeout to 0, which corresponds to EX rather than EW.

The Code

Here is the code which will set EXEG as a keyword which is capable of executing executable Things with a set of channel IDs and a parameter string on its stack.

The syntax is

```
EXEG tname(, filename or #channel)[;parameter string]
```

fname is the Thing name (with or without quotes or apostrophes)

() means optional repeated

[] means optional once only

The Listing

```
; exeg4_asm
        IN      WIN1_LIB_FNPROC_ASM macro for defining procs and funcs
        IN      WIN1_LIB_HED1      macro for setting strings

sys_thgl equ    $b8      address of start of Thing linkage blocks
bv_ntbas equ    $18      base of Name Table
bv_ntp  equ    $1c      current top position in Name Table
bv_nlb基础 equ    $20      base of Name list
bv_vvbas equ    $28      base of variables area
bv_chbas equ    $30      base of basic channels
bv_chp  equ    $34      current top of channels
bv_rip  equ    $58      maths stack

*****
*   The following code links in EXEG as a keyword using LRESPR   *
*****

START    LEA      DEFINE,A1
         MOVEA.W  BP_INIT,A2
         JMP      (A2)

define
proc_start
pf_name  exg,EXEG  The code for EXEG is at exg
proc_end
fn_start
fn_end

*****
* 1. We first find the number of channels (nc) and the length of *
*      the parameter string rounded (lc).                        *
*****

exg      moveq    #1,d7          lc = 0 (+ 1 for rounding)
         moveq    #-8,d5
         add.l    a5,d5
         sub.l    a3,d5
         asr.l    #3,d5          t (number of pars - 1)
         bmi     bad_parm ----- must be > 0 pars
         beq     exg_2          no parameters for the Thing
         bfextu   -15(a5,a6.1){1:3},d0 second last separator
         subq.w   #1,d0          comma? . .
         beq     exg_2          . . yes (no parameter string)
exg_1    subq.w   #1,d0          semicolon? . .
         bne     bad_parm ----- . . no
         subq.w   #1,d5          nc = t - 1
         move.l   -4(a5,a6.1),d0 offset to string
         add.l    bv_vvbas(a6),d0
         add.w    (a6,d0.1),d7    lc = par length (+ 1 for rounding)

; D5 = nc: D7 = lc + 1
*****
* 2.      Now get the Thing name                                *
*****
```

```

exg_2    bclr      #0,d7          lc rounded up to even
         move.w    (a3,a6.l),d0   1st word of par info
         andi.w    #$ff0f,d0     strip separator
         cmpi.w    #$201,d0      string variable? . .
         beq       exg_3         . . yes
         subq.w    #2,d0         must be this? . .
         bne       bad_parm ----- . . it isn't!!
         bsr       n_to_stack     get name from name list to maths stack
         bne       bad_parm -----
         bra       exg_4
exg_3    bsr       gstring        get ordinary string
         bne       bad_parm -----

```

; The Thing's name is on the maths stack

```

*****
* 3.      We must find the Thing's linkage block      *
*****

```

```

exg_4    moveq     #mt_inf,d0      System information
         trap      #1             AO -> system variables
         move.l    sys_thgl(a0),d0 Thing list
         bra       thok
thok1    move.l    (d0.l),d0       next thing linkage block
thok     beq       bad_parm ----- either no list or not found
         lea       $2a(d0.l),a2   -> name
         bsr       cp             compare names
         bne       thok1         not this one

```

```

*****
* 4.      We must locate the Thing                  *
*****

```

```

         movea.l   $10(d0.l),a0    -> thing
         move.l    4(a0),d0        type
         subq.l    #1,d0           is it executable? . .
         bne       bad_parm ----- . . no

```

; AO -> Thing

```

*****
* 5.      We must create this as a Job              *
*****

```

```

         movea.l   8(a0),a2
         adda.l    a0,a2           -> code
         move.l    $C(a0),d2      code length
         move.l    $10(a0),d3     dataspace . .

```

; We need to increase the dataspace by $nc*4 + lc + 4$

```

         move.l    d5,d0          nc
         asl.l     #2,d0          nc*4
         swap      d5
         move.w    d7,d5          D5 = nc | lc
         add.l     d0,d7          nc*4 + lc
         add.l     d7,d3          adjusted dataspace
         addq.l    #4,d3          4 bytes for the counts
         move.l    $14(a0),d0     Program start
         beq       exg_5         A1 -> 0

```

```

add.l    a0,d0                A1 not zero
exg_5    movea.l    d0,a1

; We now create the Job

moveq     #0,d1                independent job
moveq     #mt_cjob,d0
trap      #1
tst.l     d0
bne       bad_parm    ---->    can't start a job!!
move.l    d1,d6                keep ID in D6.L

```

```

*****
* 6.      Adjust the stack and fill in the code      *
*****

```

```

movea.l    -12(a0),a4          stack to A4
suba.l     d7,a4               adjusted stack for chans and par
move.l     a4,-12(a0)          replace stack pointer
lsr.l      #1,d2               code length in words
bra        exg_6

```

```

; There may be more than 2^32 bytes of program so we count the
; two halves of D2 separately

```

```

exg_7      swap      d2
exg_8      move.w     (a2)+,(a0)+    copy code a word at a time
exg_6      dbf        d2,exg_8
           swap      d2
           dbf        d2,exg_7

```

```

; Now A4 -> prog's stack for filling
;   D5 = nc (number of channels) | lc (length of par string)
;   D6 = Job ID for Thing
;   A3 -> parameters after the Thing's name

```

```

*****
* 7.      Find the channel IDs and put them on the stack      *
*****

```

```

           swap      d5                D5.W = nc
           move.w     d5,(a4)+          count of channels
           bra        ta4

ta10       btst       #7,1(a3,a6.l)    is it #? . .
           beq        ta5                . . no
           bsr        gtin              get channel number . .
           bne        nochan    ---->
           move.w     (a1,a6.l),d1      . . to D1
ta9        move.l     bv_chbas(a6),a0
           mulu.w     #$28,d1
           add.l      d1,a0              -> channel
           cmpa.l     bv_chp(a6),a0
           bge        nochan    ---->    over the top!
           move.l     (a0,a6.l),d1      ID
           bmi        nochan    ---->    gone!!
           movea.l    d1,a0              ID to A0
           bra        ta6

ta5        tst.b      (a3,a6.l)         Not a string type
           beq        ta7

```

```

        bsr      gstring          String to maths stack
        bne      bad_parm3 ----->
        bra      ta8
ta7      bsr      n_to_stack      Name to maths stack
        bne      bad_parm3 ----->
ta8      bsr      ope             try and open the file
        beq      ta6             OK
        bsr      ad_dat          add DATAD$ . .
        bne      nochan ----->
        bsr      ope             . . and try again
        bne      nochan ----->      didn't work

ta6      move.l   a0,(a4)+        set ID on stack

ta4      dbf      d5,ta10        count channels

*****
* 8.          Put the parameter string on the stack          *
*****

        swap     d5              D5.W = 1c (rounded)
        tst.w    d5              par string? . .
        beq      ta11            . . no
        bsr      gstring
        bne      bad_parm3 ----->
        move.w    (a1,a6.1),d0    length of par string . .
        move.w    d0,(a4)+        . . set on stack
        bra      ta12
ta13     move.b    2(a1,a6.1),(a4)+ copy the string to the stack
        addq.l    #1,a1
ta12     dbf      d0,ta13

*****
* 9.          Activate the job                                *
*****

ta11     move.l    d6,d1          ID of Thing job
        moveq     #$10,d2        priority 16
        moveq     #0,d3          EX (zero timeout)
ta14     moveq     #mt_activ,d0
        trap      #1
        rts                    return to BASIC

```

; Sub Routines

; n_to_stack sets the name list entry for (a3,a6) to the maths stack
; uses no reg except that A3 is updated to point to the next parameter

```

ns_reg    reg      d0-3/a0/a2
n_to_stack
        movem.l   ns_reg,-(sp)
        movea.l    bv_ntbas(a6),a0
        move.w     2(a3,a6.1),d0    index to name
        lea        (a0,d0.w*8),a0    -> name table entry
        cmpa.l     bv_ntp(a6),a0
        bcc        nts1 ----->
        move.w     2(a0,a6.1),d1    offset to name list
        bmi        nts1 ----->
        movea.l    bv_nlbases(a6),a0
        add.w      d1,a0            -> name in name list
        moveq      #3,d1

```

	add.b	(a0,a6.l),d1	name length + 2 . .
	bclr	#0,d1	. . rounded up
	move.w	d1,-(a7)	length needed on maths stack . .
	movea.w	bv_chrix,a2	
	jsr	(a2)	. . ensure it is there
	movea.l	bv_rip(a6),a1	maths pointer
	suba.w	(a7)+,a1	point to start of string . .
	move.l	a1,bv_rip(a6)	. . and update stack
	moveq	#0,d1	
	move.b	(a0,a6.l),d1	name length to D1.W
	move.w	d1,(a1,a6.l)	
	bra	nts3	
nts4	addq.l	#1,a0	advance A0
	move.b	(a0,a6.l),2(a1,a6.l)	-> stack
	addq.l	#1,a1	advance A1
nts3	dbf	d1,nts4	
	movea.l	bv_rip(a6),a1	reset A1 to start
	addq.l	#8,a3	-> next parameter
	moveq	#0,d0	
nts2	movem.l	(sp)+,ns_reg	
	rts		
nts1	moveq	#-1,d0	
	bra	nts2	

; gstring and gtin get one string and one integer parameter.
; If an error occurs condition code is NE.
; A3 is updated to the next parameter.
; No registers are used except A2

gs_reg	reg	d0-4/d6/a0/a4	
gtin	movea.w	ca_gtint,a2	
	bra	gs1	
gstring	movea.w	ca_gtstr,a2	
gs1	movem.l	gs_reg,-(sp)	
	movea.l	a5,a4	keep A5
	lea	8(a3),a5	set for 1 parameter . .
	jsr	(a2)	. . and get it
	movea.l	a5,a3	update A3
	movea.l	a4,a5	replace A5
	movem.l	(sp)+,gs_reg	
	rts		

bad_parm3	bsr	remv	
bad_parm	moveq	#-15,d0	bad parameter
	rts		

nochan	bsr	remv	
	moveq	#-6,d0	invalid channel
	rts		

remv	move.l	d6,d1	ID of Thing job
	moveq	#0,d3	No error
	moveq	#mt_frjob,d0	force remove job
	trap	#1	
	rts		

; cp compares a string at A2 with a string at (A1,A6.L) regardless of case.
; Returns cc EQ if found else NE. No regs used.
; NOTE. The string length must not be zero.

cp_reg	reg	d0-2/d3/a1-2	
cp	movem.l	cp_reg,-(sp)	
	move.w	#\$DF,d3	prepare for ANDing
	move.w	(a2)+,d0	length
	cmp.w	(a1,a6.l),d0	same? . .
	bne	cp_end	. . no
	subq.w	#1,d0	
cp1	move.b	(a2)+,d1	
	move.b	2(a1,a6.l),d2	
	eor.b	d2,d1	set bits which differ
	and.w	d3,d1	ignore case
	addq.l	#1,a1	cond codes unaltered
	dbne	d0,cp1	carry on till unequal
cp_end	movem.l	(sp)+,cp_reg	
	rts		

```

; ad_dat adds DATAD$ to the start of the name at (A1,A6.L)
; A1 is updated and set to BV_RIP(A6)
; On error D0 is set to "not found". CC are set
; no other registers are used

```

reg_dat	reg	d1-4/a0/a2	
ad_dat	movem.l	reg_dat,-(a7)	
	moveq	#mt_inf,d0	System information
	trap	#1	A0 -> system variables
	move.l	sv_data(a0),d0	-> DATAD\$
	beq	not_found ——>	
	movea.l	d0,a0	Pointer in A0
	moveq	#0,d1	
	move.w	(a0)+,d1	length to D1.L for BV_CHRIX
	move.w	d1,d4	keep in D4.W
	move.l	a1,bv_rip(a6)	store current value of stack
	movea.w	bv_chrix,a2	check there's room
	jsr	(a2)	
	movea.l	bv_rip(a6),a1	new position (possibly)
	move.w	(a1,a6.l),d3	original length of filename
	movea.l	a1,a2	keep position in A2
	suba.w	d4,a1	new value of A1 . .
	btst	#0,d4	odd? . .
	beq	ad_dat4	. . no
	subq.l	#1,a1	set back to even address
ad_dat4	move.l	a1,bv_rip(a6)	store new position
	move.w	d3,(a1,a6.l)	set new . .
	add.w	d4,(a1,a6.l)	. . length
	addq.l	#2,a1	skip length
	bra	ad_dat5	
ad_dat6	move.b	(a0)+,(a1,a6.l)	insert DATAD\$
	addq.l	#1,a1	
ad_dat5	dbf	d4,ad_dat6	
	cmpa.l	a1,a2	are we at the same place? . .
	beq	ad_dat7	. . yes
	bra	ad_dat1	
ad_dat2	move.b	1(a1,a6.l),(a1,a6.l)	move old name . .
	addq.l	#1,a1	. . back . .
ad_dat1	dbf	d3,ad_dat2	. . one byte
ad_dat7			
	movea.l	bv_rip(a6),a1	reset A1
	moveq	#0,d0	
ad_dat3	movem.l	(sp)+,reg_dat	
	rts		


```

not_found moveq    #-7,d0
          bra      ad_dat3

ope       moveq    #0,d3                OPEN (old exclusive)
          move.l   d6,d1                ID
          trap     #4                  A1 relative to A6
          movea.l  a1,a0                -> name (rel to A6)
          moveq    #io_open,d0
          trap     #2
          tst.l    d0
          rts

```

Unusual Instructions

The above code uses three types of instruction which requires a 68020+ for its execution.

a. While searching through the linkage blocks for the Thing's name the instructions

```

thok1     move.l   (d0.l),d0            next thing linkage block and
          lea      $2a(d0.l),a2        -> name

```

appear.

The effective addresses (d0.l) and \$2a(d0.l) have no base register, thus allowing a data register to be used as if it were an address register.

b. A bit field instruction is used to extract a separator towards the start of EXEG. Such instructions allow a contiguous set of bits from 1 to 32 bits to be manipulated. The start of the set of bits is defined by an offset from an effective address. The size of the bit field can be set to any number from 1 to 32. If a data register is used to define the offset values from -2^{31} to $2^{31}-1$ can be set. Otherwise the offset is restricted to a number between 0 and 31.

The instruction used here is:

```

bfxetu   -15(a5,a6.l){1:3},d0 second last separator

```

The offset is 1 and the size 3 as you can see from the numbers inside the curly brackets.

bfxetu stands for Bit Field EXtract Unsigned. The three bits extracted are placed in the bottom three bits of D0. The remaining bits of D0 are set to zero.

c. In the subroutine n_to_stack we need to locate the n - 1th entry in the Name Table. Given the number n - 1 in D0 we require eight times D0 to be added to the base address of the table. The instruction used is:

```

lea      (a0,d0.w*8),a0 -> name table entry

```

You can see that the index register D0 has been multiplied by 8. Values of 2, 4 and 8 are allowed.

Negotiation, Negotiation, Negotiation

by Geoff Wicks

I had a moment of absolute horror when I saw the hotel room for the first time. It was just 8.5 metres by 4.5 metres. Even the hotel said the maximum capacity was 30 people, and that was with all of them seated theatre style. There was no way we could hold a show in that room. All I

could do was to wait for the Quanta Committee to arrive and take it from there.

I went for breakfast and was allocated a table next to a familiar face. John Mason was unconcerned about the problem, but then he had done

these versions I used to use EX NET_PEEK expecting PROGDS\$ to be set to the directory containing the program. One day, I was experimenting with C68, for which I had set PROGDS\$ to the directory WIN1_C68_. It annoyed me when I typed EX NET_PEEK and got the infuriating message "Not found". I had to laboriously type EX WIN1_SYS_NET_PEEK to get the program to load. I then remembered that another program, much used by me, could be loaded by using EXEP. Typing EXEP QD always produced another version of QD whatever the setting of PROGDS\$. I determined there and then to make NET_PEEK an executable Thing, just like QD.

Two problems had to be solved first though. I wanted to be able to call NET_PEEK either as an ordinary executable program by EX or as an executable Thing by EXEP. To set NET_PEEK as a Thing I needed to LRESPR it. The first problem was how to program NET_PEEK so that it could distinguish between being LRESPRd to become a Thing and being loaded by EX. The second problem was how to make NET_PEEK into a Thing.

Problem 1 – LRESPR or EX

The first problem is easy to solve. If NET_PEEK has been LRESPRd the program ID will be that of either master BASIC or that of a daughter BASIC, if SMSQ/E is in operation. If the program ID is zero then NET_PEEK has been LRESPRd via master BASIC. If the long word just before the position to which A6 points is "SBAS" then we are in a daughter BASIC. In all other cases it is reasonable to assume that the program has been started by EX.

As it happens, I do not allow an LRESPR by a daughter BASIC. This is because if the daughter BASIC is removed, then so will the NET_PEEK Thing. To give NET_PEEK immortality we need master BASIC to do the LRESPRing.

Problem 2 – How to Make NET_PEEK a Thing

The manuals tell us that we need a linkage block defining NET_PEEK as a Thing and that we also need a Thing Header.

Linkage Block

The linkage block consists of 42 bytes of information followed by a string containing the name of the Thing. The long word at position \$10 is a pointer to the Thing itself. The manual states that the "linkage block must be in the common heap. ." This block is linked into the Thing list by sms.lthg, the description of which also states that the linkage block "must be allocated in the common heap . .

Thing Header

The format of an executable Thing's header is:

Item	Address	Length	Value
THH_FLAG	\$00	4 bytes	"THG%"
THH_TYPE	\$04	long	1 (for executable code)
THH_HDRS	\$08	long	offset to start of header
THH_HDRL	\$0C	long	size of header
THH_DATA	\$10	long	dataspace
THH_STRT	\$14	long	offset to start of code or 0

Notes

1. The offsets are calculated from the start of the Thing Header.
2. If a program does not alter its code it is said to be re-entrant. In this case it is possible to have several instances of the program in force at one time with all of them using the same code. Each of the programs will have its own entry in the Program Table and its own Header followed by the few bytes at the start of the program containing its name. After this comes the dataspace. The alternative is to have multiple copies of the whole program. The Thing Header supports both these alternatives. For the first, THH_DATA contains the length of program to just after its name and THH_START points to the single copy of the whole program. In the second case, THH_DATA contains the length of the whole program and THH_START is set to zero.

Bearing in mind that the Linkage Block had to be in the "common heap" I decided to put it at the end of my program NET_PEEK just after the Thing Header, since presumably the program would be put in the common heap by LRESPR.

End of Slight Digression

If you remember, I was just looking into LRESPR. Well, I decided to try out LRESPR on NET_PEEK to make it a Thing. This had worked well on Q40, Q60 and QPC2 to name a few platforms. Without any problems UQLX accepted the LRESPR command. However, when I typed EXEP NET_PEEK I got an error message. I think it was "Not found" but I'm not sure. My mind had just then become completely blank.

Luckily I was able to load NET_PEEK by using EX. I used that version to trace through all the Things to make sure that NET_PEEK was indeed linked into the Thing list. Since it was in the list I couldn't understand why EXEP did not work. In an attempt to find out I looked more closely at NET_PEEK's linkage block and at its Thing Header. After a bit I noticed that the item THH_HDRL, which should have been a relatively small number was incredibly large! I noticed in fact that it was the absolute address of the Linkage Block. What was this doing in the middle of NET_PEEK's Thing Header?

The code at the end of my program was as follows.

```
; This is the NET_PEEK Thing

N_THING  DC.L      "THG%",1
         DC.L      HEADZ-N_THING      ; Pointer to header
         DC.L      PRS-HEADZ          ; Size of header
         DC.L      D_SPACE            ; Amount of dataspace
         DC.L      STARTA-N_THING     ; Pointer to start of code

; This is the linkage block

TLINK    DCB.W      19,0
         DC.L      "1.00"
         HED1      <"NET_PEEK">, TLINK1
```

I found that although the size of header calculated as PRS-HEADZ above was correctly set before the Linkage Block was linked into the Thing list, it had been altered by the time linkage was complete. I traced the offending code to a subroutine called th_newth in the SMSQ/E source code. It resides in the file util_thg_usage_asm. This code quite definitely sets the address of the Linkage block into a long word six bytes earlier than the start of that block. As you can see, in my program that is just where the size of header is set in the Thing Header.

Explanation

From the point of view of an application programmer all this might seem extremely odd. But a systems programmer will see it differently. The application programmer should know that when a program is removed any of its channels still open are closed and any space allocated to it is returned to the heap. These useful facts allow application programmers to save code relying on the operating system to clean up after them. They do not need to know how the cleaning up is done.

The systems programmer has to be aware of the details of the cleanup. After all he has to do the coding of this. So how is it done? When a job is removed the operating system must go through the channel table closing all those channels owned by that job. It must also be the case that all allocations from the heap are examined and those owned by the job being removed returned to the heap. Of course, if the allocated area happens to contain a Thing linkage block it will be pretty disastrous if the area is returned to heap without previously being unlinked.

Each area allocated from the common heap has a 16-byte header. I can only find one reference to this header in any of the relevant manuals. That is the Appendix R on page 336 of Dicken's QL Advanced User Guide. However, keys_chp in the SMSQ/E source code perhaps shows more clearly what this header contains. There are two types of block, those which are owned by a job and those which are free space. The contents of the four long words of the header are slightly different in the two cases and appear to be as shown here.

Free Space

Address	Value
\$00	Length of area
\$04	Relative pointer to next free space
\$08	-1 (SMSQ/E), 0 (QDOS)
\$0C	0

Allocated space

Address	Value
\$00	Length of area
\$04	Pointer to driver linkage or 0
\$08	ID of owner job
\$0C	Address of flag byte set when space released or 0

All device drivers have a long word containing the link to the next driver and, three long words later, the address of the "close" routine. It seems clear that if space allocated to a driver's linkage block is to be returned to the heap it should first be unlinked. This can be done by the "close" routine. Indeed, when a job is removed the code returning allocated space owned by the program first calls a "close" routine if there is one.

The Thing linkage block is treated just like a device driver. Thus, when a linkage block is linked into the Thing list, the address of the linkage block is placed six bytes before the start of the block, which is what I was complaining about earlier, and also the address of a "close" routine is set three long words on from the start of the linkage block. It is this which ensures that the Thing linkage block is unlinked before the space in which it has been living, is discarded.

But all this is on the assumption that the linkage block is the first, or only, item in the allocated space. It is a pity that the manuals do not mention this important fact. I suggest that the wording in the manual should have been that the linkage block "must be the first or only item in space allocated from the common heap . . .".

Final Comments

First

I did not want to alter NET_PEEK so that it would create a Thing linkage block in its very own space. Instead I moved the linkage block down a bit by adding the line

```
PSEUDO_H DC.L 0,0,0,0 ; Pseudo header for Thing linkage
```

just before the Thing linkage block.

This allowed the linking of the linkage block to occur without damaging NET_PEEK's Thing header.

Since I did not allow NET_PEEK to be LRESPRd from any BASIC but the master, I knew that the owner of the linkage block would never be removed so that the unlinking would never need to occur by space being returned to the heap.

Second

Having altered NET_PEEK in the way I have mentioned I loaded it into UQLX and found that it both LRESPRd to a Thing and also came to life by EXEP NET_PEEK. I assume that the very large space erroneously assigned to the header by the Thing linking had made it impossible to set NET_PEEK as a program started from the executable Thing called NET_PEEK.

Last

This all started because of the cry "LRESPR doesn't work". Well, so far that has not been explained. I conjecture that the use of an early ROM in the particular UQLX meant that LRESPR failed because some jobs had already been loaded which means that the resident space is no longer available. Of course with SMSQ/E we do not have that problem.

Slight Digression

One of my programs which I use frequently is **NET_PEEK**. This program analyses the contents of ram, both of the machine on which it is running and on any other machine linked into a network. Analysis includes such things as a list of jobs, a list of channels, the contents of a job's registers and the particulars of any open channel. It also, if running on a 68020+, will disassemble instructions. In short I find it indispensable. Indeed I often have more than one version of **NET_PEEK** running. To load these versions I used to use **EX NET_PEEK** expecting **PROGD\$** to be set to the directory containing the program. One day, I was experimenting with C68, for which I had set **PROGD\$** to the directory **WIN1_C68_**. It annoyed me when I typed **EX NET_PEEK** and got the infuriating message 'Not found'. I had to laboriously type **EX WIN1_SYS_NET_PEEK** to get the program to load. I then remembered that another program, much used by me, could be loaded by using **EXEP** Typing **EXEP QD** always produced another version of **QD** whatever the setting of **PROGD\$**. I determined there and then to make **NET_PEEK** an executable Thing, just like **QD**.

Two problems had to be solved first though. I wanted to be able to call **NET_PEEK** either as an ordinary executable program by **EX** or as an executable Thing by **EXEP**. To set **NET_PEEK** as a Thing I needed to **LRESPR** it. The first problem was how to program **NET_PEEK** so that it could distinguish between being **LRESPR**d to become a Thing and being loaded by **EX**. The second problem was how to make **NET_PEEK** into a Thing.

Problem 1 – LRESPR or EX

The first problem is easy to solve. If **NET_PEEK** has been **LRESPR**d the program ID will be that of either master BASIC or that of a daughter BASIC, if **SMSQ/E** is in operation. If the program ID is zero then **NET_PEEK** has been **LRESPR**d via master BASIC. If the long word just before the position to which **A6** points is 'SBAS' then we are in a daughter BASIC. In all other cases it is reasonable to assume that the program has been started by **EX**.

As it happens, I do not allow an **LRESPR** by a daughter BASIC. This is because if the daughter BASIC is removed, then so will the **NET_PEEK** Thing. To give **NET_PEEK** immortality we need master BASIC to do the **LRESPR**ing.

Problem 2 – How to Make NET_PEEK a Thing

The manuals tell us that we need a linkage block defining **NET_PEEK** as a Thing and that we also need a Thing Header.

Linkage Block

The linkage block consists of 42 bytes of information followed by a string containing the name of the Thing. The long word at position \$10 is a pointer to the Thing itself. The manual states that the 'linkage block must be in the common heap.' This block is linked into the Thing list by **sms.lthg**, the description of which also states that the linkage block 'must be allocated in the common heap.'

Thing Header

The format of an executable Thing's header is:

Item	Address	Length	Value
THH_FLAG	\$00	4 bytes	"THG%"
THH_TYPE	\$04	long	1 (for executable code)
THH_HDRS	\$08	long	offset to start of header
THH_HDRL	\$0C	long	size of header
THH_DATA	\$10	long	dataspace
THH_STRT	\$14	long	offset to start of code or 0

Notes

1. The offsets are calculated from the start of the Thing Header.
2. If a program does not alter its code it is said to be re-entrant. In this case it is possible to have several instances of the program in force at one time with all of them using the same code. Each of the programs will have its own entry in the Program Table and its own Header followed by the few bytes at the start of the program containing its name. After this comes the dataspace. The alternative is to have mul-

multiple copies of the whole program. The Thing Header supports both these alternatives. For the first, THH_DATA contains the length of program to just after its name and THH_START points to the single copy of the whole program. In the second case, THH_DATA contains the length of the whole program and THH_START is set to zero.

Bearing in mind that the Linkage Block had to be in the 'common heap' I decided to put it at the end of my program NET_PEEK just after the Thing Header, since presumably the program would be put in the common heap by LRESPR.

End of Slight Digression

If you remember, I was just looking into LRESPR. Well, I decided to try out LRESPR on NET_PEEK to make it a Thing. This had worked well on Q40, Q60 and QPC2 to name a few platforms. Without any problems UQLX accepted the LRESPR command. However, when I typed EXEP NET_PEEK I got an error message. I think it was 'Not found' but I'm not sure. My mind had just then become completely blank.

Luckily I was able to load NET_PEEK by using EX. I used that version to trace through all the Things to make sure that NET_PEEK was indeed linked into the Thing list. Since it was in the list I couldn't understand why EXEP did not work. In an attempt to find out I looked more closely at NET_PEEK's linkage block and at its Thing Header. After a bit I noticed that the item THH_HDRL, which should have been a relatively small number was incredibly large! I noticed in fact that it was the absolute address of the Linkage Block. What was this doing in the middle of NET_PEEK's Thing Header?

The code at the end of my program was as follows.

```
; This is the NET_PEEK Thing
N_THING  DC.L      "THG%",1
         DC.L      HEADZ-N_THING    ; Pointer to header
         DC.L      PRS-HEADZ        ; Size of header
         DC.L      D_SPACE          ; Amount of dataspace
         DC.L      STARTA-N_THING   ; Pointer to start of code

; This is the linkage block
TLINK    DCB.W     19,0
         DC.L      "1.00"
         HED1      < "NET_PEEK">, TLINK1
```

I found that although the size of header calculated as PRS-HEADZ above was correctly set before the Linkage Block was linked into the Thing list, it had been altered by the time linkage was complete. I traced the offending code to a subroutine called th_newth in the SMSQ/E source code. It resides in the file util_thg_usage.asm. This code quite definitely sets the address of the Linkage block into a long word six bytes earlier than the start of that block. As you can see, in my program that is just where the size of header is set in the Thing Header.

Explanation

From the point of view of an application programmer all this might seem extremely odd. But a systems programmer will see it differently. The application programmer should know that when a program is removed any of its channels still open are closed and any space allocated to it is returned to the heap. These useful facts allow application programmers to save code relying on the operating system to clean up after them. They do not need to know how the cleaning up is done.

The systems programmer has to be aware of the details of the cleanup. After all he has to do the coding of this. So how is it done? When a job is removed the operating system must go through the channel table closing all those channels owned by that job. It must also be the case that all allocations from the heap are examined and those owned by the job being removed returned to the heap. Of course, if the allocated area happens to contain a Thing linkage block it will be pretty disastrous if the area is returned to heap without previously being unlinked.

Each area allocated from the common heap has a 16-byte header. I can only find one reference to this header in any of the relevant manuals. That is the Appendix R on page 336 of Dicken's QL Advanced User Guide. However, keys_chp in the SMSQ/E source code perhaps shows more clearly what this header contains. There are two types of block, those which are owned by a job and those which are

free space. The contents of the four long words of the header are slightly different in the two cases and appear to be as shown here.

Free Space

Address	Value
\$00	Length of area
\$04	Relative pointer to next free space
\$08	-1 (SMSQ/E), 0 (QDOS)
\$0C	0

Allocated space

Address	Value
\$00	Length of area
\$04	Pointer to driver linkage or 0
\$08	ID of owner job
\$0C	Address of flag byte set when space released or 0

All device drivers have a long word containing the link to the next driver and, three long words later, the address of the "close" routine. It seems clear that if space allocated to a driver's linkage block is to be returned to the heap it should first be unlinked. This can be done by the "close" routine. Indeed, when a job is removed the code returning allocated space owned by the program first calls a "close" routine if there is one.

The Thing linkage block is treated just like a device driver. Thus, when a linkage block is linked into the Thing list, the address of the linkage block is placed six bytes before the start of the block, which is what I was complaining about earlier, and also the address of a "close" routine is set three long words on from the start of the linkage block. It is this which ensures that the Thing linkage block is unlinked before the space in which it has been living, is discarded.

But all this is on the assumption that the linkage block is the first, or only, item in the allocated space. It is a pity that the manuals do not mention this important fact. I suggest that the wording in the manual should have been that the linkage block "must be the first or only item in space allocated from the common heap ...".

Final Comments

First

I did not want to alter **NET_PEEK** so that it would create a Thing linkage block in its very own space. Instead I moved the linkage block down a bit by adding the line

```
PSEUDO_H DC.L      0,0,0,0    ; Pseudo header for Thing linkage
```

just before the Thing linkage block.

This allowed the linking of the linkage block to occur without damaging **NET_PEEK**'s Thing header. Since I did not allow **NET_PEEK** to be **LRESPRd** from any BASIC but the master, I knew that the owner of the linkage block would never be removed so that the unlinking would never need to occur by space being returned to the heap.

Second

Having altered **NET_PEEK** in the way I have mentioned I loaded it into UQLX and found that it both **LRESPRd** to a Thing and also came to life by **EXEP NET_PEEK**. I assume that the very large space erroneously assigned to the header by the Thing linking had made it impossible to set **NET_PEEK** as a program started from the executable Thing called **NET_PEEK**.

Last

This all started because of the cry "LRESPR doesn't work". Well, so far that has not been explained. I conjecture that the use of an early ROM in the particular UQLX meant that **LRESPR** failed because some jobs had already been loaded which means that the resident space is no longer available. Of course with SMSQ/E we do not have that problem.