```
710  IF MTEXT$(#3,key)<>""
720   NEXT main
730  ELSE
740   IF awnum=1
750    names$(position)=buffer$
760   ELSE
770    selection$(position)=buffer$
780   END IF
790   MAWITEM #3,key,,buffer$
800   buffer$=""
810   drag%=NOT(drag%)
820   SPRS #3,0
830  END IF
840 ELSE
850  IF MTEXT$(#3,key)<>""
860   IF awnum=1
870    buffer$=names$(position)
880    names$(position)=""
890   ELSE
900    buffer$=selection$(position)
910    selection$(position)=""
920   END IF
930   MAWITEM #3,key,,""
940   drag%=NOT(drag%)
950   SPRS #3,1
960  ELSE
970   NEXT main
980  END IF
990 END IF
1000 END DEFine DRAG_DROP
1010 :
1020 DEFine PROCedure SHOWTIME
1030 LOCal number
1040 number=MAWNUM(#3\3)
1050 open_over #4,ram1_showtime
1060  print #4,"In week "&number
1070  print #4,"the following players were selected:"
1080  print #4,selection$
1090 close #4
1100 END DEFine SHOWTIME
```

The Extended Environment in SBASIC: Programming with QPTR

Wolfgang Lenerz

This is the new Guide to using QPTR in SuperBASIC. The purpose hereof is to enable you to program the "Extended Environment" - so called the (Pointer Environment) very easily with the QPTR extensions (which you must obtain seperately). Contrary to what a first impression may let you believe, the Extended Environment, and QPTR at the same time, are not complicated or difficult, but just complex, notably because there are so many new concepts to assimilate at once. But it is actually sufficient to know and respect its "philosophy" to see - and understand - the logic behind it.

I sincerely hope this Guide will be useful to you.

Introduction

This is an explanation of the concepts and keywords needed to ***program*** applications using the QPTR SuperBASIC extensions. For some aspects, we will use examples derived from QPAC II, it is thus hoped that the reader is familiar with this software...

Before starting on the course proper, some terms might need an explanation:

The **Extended Environment** essentially is just a "new" method to interact with the user of an application:
Interaction means, on the one hand, display of information on the screen (in windows) and, on the other hand, obtaining the user's response to this information (often, but not always, through a 'pointer'). For example, a file copier displays information (the name of files on a disk) and obtains the user's response (i.e. selecting which files to copy). The Extended Environment (which I'll abbreviate as EE from here on) can handle that aspect of a program, but the rest of the program will remain (nearly) unchanged: in the example just used, it is still you, the programmer who will have to program the copying operation itself.

An **application** is simple a program.

A window is said to be **managed** when it is part of an application written specifically to take advantage of the facilities offered by the EE: QPAC II has managed windows, QUILL has windows that are not managed.

The EE changes not only several aspects of the QL's windowing system, but also the QL's multitasking. Here, the concept of a window is enlarged to mean not only the means through which an application will communicate with the user, but also the means to determine whether an application will multitask or will be suspended.

The best way to understand that is to imagine that, for the EE, all window are "stacked" on a pile (one considers that an application has but one main window). The window that is on top of the pile is that which is entirely visible on the screen. This window is said to be **unlocked** which means that it will accept input if you type something in it and if the mouse pointer is over it. If you now hit CTRL + C, then the window on the top of the pile will get transferred to the bottum of the pile, and the window that was just under that one will be on top of the pile. Now, if the pointer is in that window anything you type (or any click of the mouse) will be directed to that window and is thus taken into account by the application to which that window belongs. The window on top of the pile will be called the upper window. The other windows, which are underneath it, do not accept keystrokes. It is then said that thay are **locked**.

It is possible for two or more windows to be on top of the pile at the same time, and to be visible entirely (if they are small enough...). Both windows will then be unlocked. However, if anything is typed on the keyboard, the keystrokes thus generated go only to the window in which one can see the pointer.

The concept of ***locked windows*** is important: An application whose window is locked will be suspended (i.e. it stops working) until its window becomes unlocked IF this application either attempts to write to the screen or is waiting for user input. Example: You are working in Abacus, and ask it to recalculate a large speadsheet. As soon as Abacus starts to do that, you switch to Basic. Abacus will continue to work on the spreadsheet, until it has to display the recalculated sheet. Then it will stop cold, waiting for you to switch back to it (thus unlocking it window). Until then, Abacus is suspended.

The word **pointer** can have two meanings: first of all it can mean the concept of a pointer as used in all programmming languages, i.e. a variable pointing to something. Also, it can mean a pointer (cursor) on the screen, moved about by the mouse or cursor keys. One doesn't generally use the word "cursor" because that normally ony means a rectagular square (blinking or not), whereas a pointer can have about any shape you desire. Normally, it should be quite clear from the context which meaning of the word pointer is used, without any possibility of confusion.

The mouse pointer can be used to "hit" objects or "do" these objects: a "Hit" is either a click of the left mouse button or tapping the space bar. A "Do" is either a click with the right mouse button, or tapping the ENTER key.

An application will have a main (or "***primary***") window through which it communicates with the user. Generally, this window will be divided into "***sub-windows***". These sub-windows are but subdivisions of the main window. Thus, in QPAC II, for example, the primary window of the "Files" menu is the entire window visible. The part of the window which displays the file names is a sub-window.
Some sub-windows are a bit special in that they can have 'objects' (such as the file names in the QPAC II files menu). The state of the objects can change when hit or "done" and can even produce an action. The sub-windows are called ***application sub-windows***.

Programs using QPTR can also be **compiled**. However, you have to use the QLiberation Software's QLiberator for this, as the "Turbo" compiler cannot cope with functions and commands which return changed parameters (even thought this is explicitly foreseen for Basic keywords). As an importnt number of QPTR keywords use this feature, programs written with them cannot be compiled with "Turbo".

A program written for the EE will most likely follow the following procedure:

- Definition of window(s)
- Display of windows onscreen
- Waiting for user input
- Act on user input
- (perhaps) Re-define windows and display it
- Wait for user input etc...

This is in fact not far from 'classical' programming: QUILL doesn't do anthing else than display its windows, wait for user input, act on that etc...

Each of these stages will be discussed. The most difficult and important is the first stage, the definition of the window.

Part One: Defining the Window

We wish of course, to define windows which are managed. To this end, there are rules to be obeyed, the definition must be made in a determined manner, which may seem complicated at first. To obtain this global definition of the window, there are several **levels of definition** through which you will successively have to pass: You must first define the main window, then the different sub-windows and lists.

I - LEVEL I: Definition of the primary window

A - Some new concepts

A certain number of new concepts must be set out before we can examine the new keywords.

1) The primary window
The primary window of an application is terribly important. Put simply, it is the first window to be opened for an application - but it determines the graphical aspect of the entire application. This is why it is called the ***primary window:*** Primary not only because it is the first to be opened, but also because it primes all the others.

The primary window is paramount: your application is not allowed to open any other window outside the primay window!

2) The sub-windows
As we shall see, and as was already mentioned, the primary window itself is generally broken down into sub-windows. There are different kinds of sub-windows, each doing it's own bit. NONE of the sub-windows' sizes may exceed the size of the primary window, they must all be opened within their primary. Even if you attempted otherwise, the EE would not let you (!).

There are three types of sub-windows:
- The ***information sub_windows*** which just display some information, as their name suggests.
- The ***application sub-windows*** - they can be so diverse that it is difficult to give a precise description.
- The ***menu sub-windows***: these are a special case of application sub-window, containing "objects".

3) Menu items
In addition to sub-windows, primary windows may also have "loose menu items" (sometimes also called simply "menu items"). These can change state or produce an action when hit or "done", and, if the pointer moves over them, a border is drawn around them.

To understand these three components, let's look at the QPAC II "Files" menu:
We can see the first sub-window, containing the file names, at a first glance. This is a menu sub-window (the file names are its objects).
We also notice the menu items, such as F3: Commands, F5 All etc...
The "stripes" around the device name, are drawn within an information sub-window. Likewise, the data on the device (free sectors/ total sectors) are displyed within an information sub-window.
It should be noted that none of these sub-windows is a "window" in the QL sense (i.e. having its own SCR or CON channel), even if they behave like such. This can be seen from the "Channels" menu: the "Files" menu has but one screen channel open...

4) Secondary windows
Sometimes it is necessary to have additional windows which are true QL windows, with their own channel. These will generally be secondary windows.

Secondary windows are defined, and behave, exactly like primary windows (i.e. they have their own sub-windows, menu items etc...) BUT these secondary windows are all confined within the primary window whose size they may not exceed. Simply put, an application may not display anything outside its primary (but it is possible to make the primay bigger, if need be).
An example of a secondary window: in the QPAC II Files window, hit F3. This opens another window, which itself has menu items. This other window is a secondary window, it has its own CON_ channel, as you can see when checking through "Channels".
It is of course possible to open a new secondary window within a first secondary window (no, they're not called tertiary windows...). This can be useful if you wish to have a cascade of menus: a first menu leads to a second one, which in turn leads to another etc... (it is not, however, considered to be good programming style to use too many cascading menus).
Whilst any secondary window is, of course, limited to the size of the primary, a secondary window within another secondary window is NOT limited to the size of the first secondary window - else, successive menus would have to get progressively smaller!
In brief, an EE application has two kinds of windows: one primary window (possibly with sub-windows) and, possibly, one or several secondary windows. Each may have its own loose menu, and sub-windows.
The difference between a menu, whether it is a loose menu or the objects in a menu sub-window on the one hand, and a sub-window,on the other hand, is the fact that clicking on an item in a menu will lead to a result. This may just be to select the item (e.g. F4 - view in the QPAC II Files menu) or lead to some kind of action (e.g. F3 in the QPAC II Files menu). Clicking on a sub-window in itself generally produces no results (there is one exception to which we will come later)..

5) The working definition

To construct a primary window, you will need to build up a "**working definition**" of this window. Let's take an example with "normal" SuperBASIC. You can open a window just by typing: "OPEN#3,con_". You have then opened a window. However, to really define this window, you would then define its size and position (WINDOW#3,x,y,z,p), its colours (border, paper, ink) etc... Thus you will build up an exact definition of your window, with all your parameters.

Likewise, in the EE, you make a definition of the window according to your parameters. Here, however, this defintion, i.e. the "working definition" is more complex and it is compulsory - you cannot do without it.

B - Making the Working Definition

The working definition of the primary window is built up by the following function: **MK_WDEF** (MaKe Working DEFinition).

```
workdef= MK_WDEF (wdef%,wattr%,wptr,ltab,
inftab,apptab)
```

"workdef" then becomes a pointer to the working definition of the window. The parameters to this function are as follows:

-› * **wdef%** is an array containing the "physical" definition of the window.
In other words, it is a 4 element integer array (DIM wdef%(3)). Its elements are, in this order:
- window x size
- window y size
- x position of pointer when the window is drawn
- y position of pointer when window is drawn.

The pointer position is given as the number of pixels starting from the upper left hand corner of the window, which is considered to be at coordinates (0,0).

-› * **wattr%** is an array containing the window "attributes". These "attributes" are simply the following: window paper (& strip) colour, size and colour of the window border and size of the shadow beneath the window, in the following order:
- size of shadow
- size of border
- colour of border
- paper colour

So, there again, this is an integer array with 4 elements (DIM wattr%(3)).
The last three parameters should be clear to anyone concerned. The "size of the shadow" is given in pixels (but is multiplied by 2 by the software, to have even numbers). The shadow counts for the size of the window: On a normal QL, you could not have a window 512x256 pixels wide plus a shadow, this would make the window too large. A shadow size of 2 is generally thought to be sufficient.

-› * **wptr, ltab, inftab, apptab** are level II "pointers" (i.e. they are explained in level II):
  - wptr is a pointer (generally obtained by SPRSP) towards a sprite definition.
  - ltab is a pointer to a loose menu items list, as returned by the MK_LIL function: ltab=MK_LIL (level II parameters).
  - inftab is a pointer towards an information sub-window list, as returned by the MK_IWL function: inftab=MK_IWL (level II parameters).
  - apptab is a pointer towards an application sub-window list, as returned by the MK_AWL function: apptab=MK_AWL(Level II parameters).

Each of the last 4 pointers may be set to 0. In this case, it is considered that the list to which it points does not exist: if inftab = 0, there are no information sub-windows.

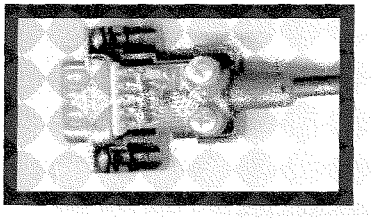
ATTENTION: It is important to respect the types of variables: if a variable is expected to be an integer, or an integer array, the variable MUST be of the correct type. Else, at best, the function in which it is used will give up with an error, at worst very bizzarre things may happen...

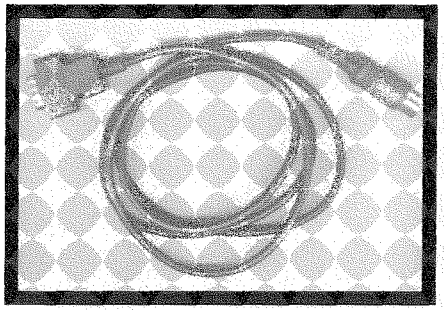
We'll continue with level II functions in the next instalment of this series.

Sinclair QL CSYNC Inverter

Marcel Flipse

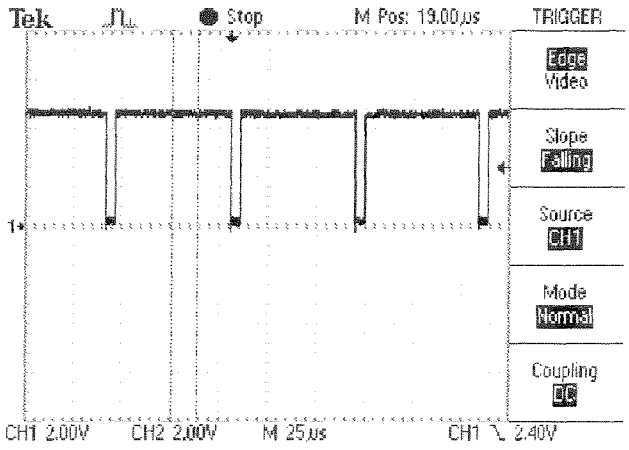
The QL cannot be connected to a CGA monitor directly. This is because the QL has an active-low Csync pin. A CGA monitor expects an active-high signal. This document shows how to make a very little circuit board, which inverts the Csync pin. No additional power supply is needed. The PCB is small enough to fit inside a DB-9 connector.



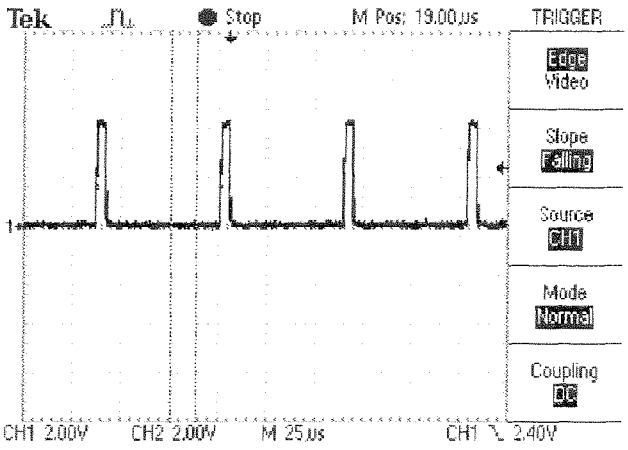


Here are some pictures of the cable.

The circuit is straightforward. The Csync signal is fed through a single gate NAND, which acts as the inverter. The NAND gate is powered by the Csync signal itself. Energy is stored in a 10uF tantalum capacitor, to buffer the time the Csync signal is low. A 100 ohm resistor is added to limit the inrush current during power-up. Some additional resistors and SOT-23 transient suppressors are added for extra protection of the QL.



Here you can see the 'original' Csync signal, measured directly at the 8-pin DIN connector at the rear of the QL.



Inverted signal

SMSQ/E machine you will be perfectly happy with either. The Q60 is a great choice for someone who just wants the best possible QL system or a QL and Linux, especially if you dislike Windows.
By the time you read this review, D&D will have sold out the entire first production run of Q60s and planning if not already selling the next run. With a machine of this calibre and with the dedication of people like Dennis Smith and Derek Stewart and the full support of designer Peter Graf, I am 100% convinced this machine will be a certain success, it really deserves to be. I have no hesitation in recommending this computer - every QLer should have one! If I'd had enough money to hand when it came to the time to hand it back, I'd have bought this machine without hesitation.

Programming with QPTR - Part 2

Wolfgang Lenerz

Continuing on from last time's instalment, here is the new part of the series on how to use QPTR. As usual, any comments are welcome.

II - LEVEL II: Definition of the lists and sprites

If you want a window to look at least somewhat interesting, you will have to dress it up a bit - so the Level II pointers should not all be 0, but should, indeed, point to something. This is what is done by the level II functions: Level II defines the (pointer and other) sprites and sub-window lists.

A - The Sprites: "wptr"

Contrary to games computers, here a "Sprite" is just a kind of image visible on the screen, which is not "independently animated". The most typical example would of course be the mouse pointer. This is a sprite, directed over the screen by a mouse or the cursor keys. It can be an arrow, or a cross (as in FiFi) or almost anything. A sprite can also be an image that is not mobile - once it is drawn it remains where it is. The mouse pointer sprite is actually exceptional in that it can move around the screen. For example of a more normal sprite, the icon used to make a window move around the screen is, in itself, a sprite (when hit, the pointer changes to that sprite).

So, the pointer used by the application is a sprite. Each primary and secondary window can have its own sprite - as can application sub-windows. In QD, the sprite is in the shape of a cursor (blinking or not), in Disktool, it is in the shape of a disk, in FiFi it has the shape of a cross etc... You will notice that the pointer sprite "looses" its specific shape as soon as it leaves an application's primary window: as soon as you put the pointer over another application, it takes the shape given to it by that application - provided, of course, that the application has managed windows and is unlocked (of course, several applications may have the same pointer sprite). The pointer over an unmanaged and unlocked window is either an arrow or a "K", depending on whether or not the application is waiting for a keystroke. Locked windows always have another default pointer, a padlock. One cannot change these default sprites.

If each application can have its own sprite as pointer, it means that each application must define this sprite. If it doesn't (wptr=0) a pointer by default will be used, i.e. the famous little arrow.

The sprite definition is built in an area of memory which must previously have been reserved by the RESPR or ALCHP (if you have Toolkit II) functions. wptr is then simply the address of this memory area:

`wptr=`*`ALCHP`*`(size)` or `wptr=`*`RESPR`*`(size)`

Now it "only" remains to find out how much memory you should reserve (this is not a fixed amount, it varies from sprite to sprite) - and then you have to fill the memory area with the data for the sprite you wish to have.

The size of this memory area depends strictly on the size of the sprite: a small sprite will need less memory than a large sprite - which seems quite logical. For the time being, sprites are limited to 64 pixels in each direction. This may seem small, but is actually not bad.

Sprites are 'printed' to the screen in a similar way to characters, i.e. imagine a grid of columns and rows. Each element, corresponding to one pixel on the screen, can be either on or off - but here, you can not only determine whether the pixel is on or off, but also in what colour is should be 'on'.

The size of the sprite thus depends on the number of columns and rows. Suppose we want

to define a sprites in a 10 by 10 grid (10 lines with 10 rows - 10x10 pixels). To define the sprite, we read these rows and columns into an array. The array will be a normal SuperBasic string array, which, with a great leap of imagination, we shall call "sprite$" in the examples. For a 10 by 10 sprite, this array must be DIMensioned as follows:

```
DIM sprite$(9,10)
```

or more generally:

```
DIM sprite$(rows-1,columns)
```

where rows and columns are the number of lines and columns respectively. The "rows-1" is because the first dimension of a sprite is sprite$(0). Thus, by using DIM sprite$(rows-1,columns) we do get an array with the required number of lines and columns.

That still doesn't tell us what value the 'size' should be. This can be obtained with the SPRSP function (SPrite Reserve SPace), which is used as follows:

```
size= SPRSP (columns, rows)
```

where, again, rows and columns are the number of columns and lines. Note the reverse order of the parameters: columns first, rows second (this is the other way round in the DIM statement).
So, attention:

*** do not state SPRSP (rows, columns), nor SPRSP (columns, rows-1) - it's (columns, rows)!.**

*** you must double the number of columns if the sprite is a mode 8 sprite, because, indeed, each pixel is twice as large in that mode...**

Thus, to reserve sufficient memory, you should proceed as follows:

```
size= SPRSP (columns,rows):
address= RESPR (size)
```

or:

```
address= RESPR ( SPRSP(columns,rows))
```

to save on a variable (of course, RESPR can be replaced by ALCHP).

Once enough memory is reserved, the sprite needs to be defined. This is most easily obtained by using the **SPSET** (**SP**rite **SET**) command:

```
SPSET address, ori_x, ori_y, mode, sprite$
```

-› * **address** is the address obtained by the RESPR, as mentioned above;

-› * **ori_x** and **ori_y** are the x and y "origins" within the sprite. It may seem curious that a sprite has origins, as the sprite (if used as a pointer), may freely move about the screen and thus its origin changes every time. Actually, these are the origins within the sprite: A sprite can be quite large, but there must be one point as of which you consider that the sprite is inside of, say, an item or a window: this is dertermined by the origin of the sprite. Suppose you have a sprite in shape of an arrow, you may wish that the point of the arrow should be the origin of the sprite, as most people will use that to point to the vaious options... So you set the origin of the sprite to be the point of the arrow.

-› * **mode** is the colour mode in which the sprite is to be drawn: 4 or 8

-› * **sprite$** is the array we have defined above (rows-1,columns).

Of course, this array must have been filled in before using the SPSET command. This is fortunately quite easy: Each row of the array is made up as follows, using a white arrow outlined in black as an example:

```
90  DATA '   a   '
100 DATA '  awa  '
120 DATA ' awwwa '
130 DATA 'awawawa'
140 DATA '  awa  '
150 DATA '  awa  '
160 DATA '  awa  '
170 DATA '  awa  '
180 DATA '  aaa  '
```

Thus our array is filled in by a program such as follows:

```
10 RESTORE 80
20 READ rows,columns
30 DIM sprite$(rows-1,columns)
40 FOR n=0 TO rows
50   READ mydata$
60   sprite$(n)=mydata$
70 END FOR n
80 DATA 8,7  : rem the number of rows & cols
90 DATA  '   a   '
100 DATA '  awa  '
```

```
120 DATA ' awwwa '
130 DATA 'awwwwwa'
140 DATA ' awa '
150 DATA ' awa '
160 DATA ' awa '
170 DATA ' awa '
180 DATA ' aaa '
```

In line 20, the number of rows and columns is read in (the DATA in line 80). After that, the array is DIMmed and the loop reads the strings from lines 90 to 180, which are used to fill in the array. There only remains to explain the meaning of these strings:

Let's start with line 90. Each character in this string stands for ONE PIXEL. Line 90 is thus the uppermost row of the sprite. It is composed of three spaces, an 'a' and again three spaces. Each character has a special meaning: A space means that this pixel will be "transparent": it will let shine through whatever lies beneath this pixel of the sprite. An 'a' means that the pixel will be black. The letters for the other colours are:

a - black
u - blue *
r - red
m - magenta *
g - green
c - cyan *
y - yellow *
w - white
space - "transparent"

The colours marked with an asterisk ('*') can only be used for mode 8 sprites.

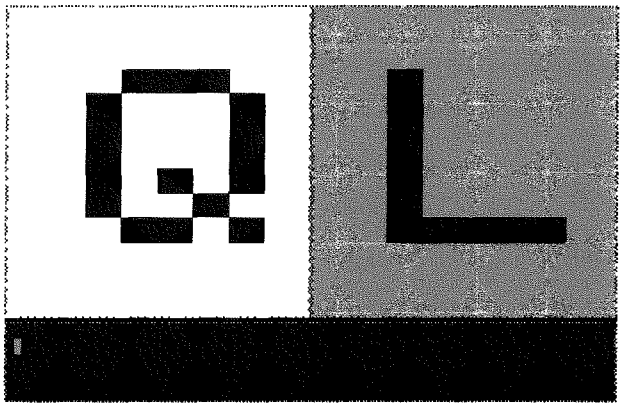
In our example, we can thus see that line 100 is composed of two transparent pixels, a black pixel, a white pixel, a black pixel and, again, several transparent pixels. In fact, the black pixels encase the white pixels. And so on for the other lines - and now we have defined the sprite. As of now, whenever we need the address of a sprite, 'wptr' will be a valid address we can use.

More next time!

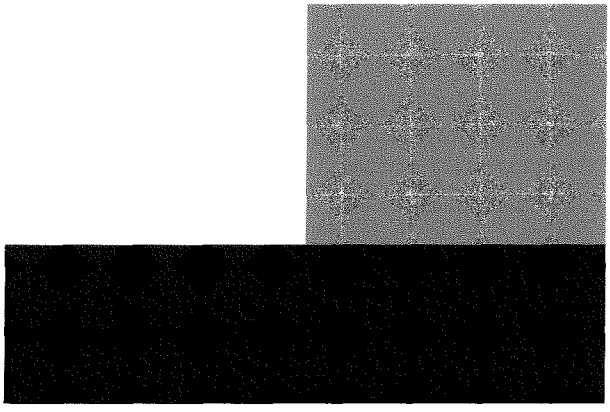
QL Logo

Dilwyn Jones

Some time ago the idea of finding a logo for the QL was floated among the QL community. Various suggestions were made and as far as I know no real consensus was arrived at. Since then, I've kept a page about this idea on my website and there has been a slow but sure contribution of ideas. Many of these might be suitable for T-shirts, mouse mats, magazine logos, anything which might help promote the QL. Some of the ideas contributed are traditional QL symbols such as the famliar red, white and black QL screen, others are much more colourful and perhaps more representative of the modern QL world.
My hope was that we could come up with something everyone would associate with the QL, in much the same way as the penguin symbol is with Linux. As far as my original idea was concerned, the best symbol of the QL is either a QL picture, or the red and white startup screen, or the letters 'QL', or the logo moulded on the original QL case! So here is my first proposal. As far as I'm concerned, anyone can use this to make a QL T-shirt or whatever - it's a GIF file of 512x256 pixel dimensions just like the startup QL screen, with the letters QL added in the chunky QL screen fount. See figure 1.



Branko Badrlijka has sent me his suggestion, a plain and simple QL monitor screen which makes for a very small graphics file which is easily resized without affecting detail.
He also suggests that a moderately thick black border may aid appearance on certain backgrounds.



Paragraph word processor from F Lanciault.

BMP2PIC - file conversion program from Phoebus Dokos, converts windows BMP files into QL PIC files.

PhotoQL - Roberto Porro's graphics conversion program.

Pnm2picr (Q40) - available from the Q40 web site, I don't know much about this program.

PIC2BMP - conversion program from Jerome Grimbert.

Q-Colour. Colour picker and display system from Wolfgang Uhlig, includes the colour "skins" extensions from Wolfgang Lenerz.

Sprite Editor - from Jerome Grimbert.

QL3D1 - from Mark Swift?

PSA conversion from George Gwilt, converts partial save area files between Q40/Q60 mode 33 style graphics to mode 32 style graphics.

PCBCad from Malcolm Lear. PCB/design program.

Screen Snatcher - grab copies of the screen picture, works on both traditional QL mode 4/8 graphics plus the new modes.

PicView - image file viewer for QL screens and PIC files.

QCDEZE from Duncan Neithercutt is a CD-ROM handler front end which uses GD2 graphics on Q40/Q60 systems.

Pan and Scroll Toolkit from Wolfgang Lenerz, available on the Phoebus Dokos website.

QDT - the QL desktop system from Jim Hunkins.

Anyone know of any more? I may update this article from time to time.

Programming with QPTR - Part 3 The Level II pointers, continued

Wolfgang Lenerz

Again we continue our exploration of QPTR. You may remember that to make a window under the Extended Environment with the Sbasic QPTR toolkit, you need a working definition, and that the working definition is obtained by the function ***MK_WDEF***, thus:

```
workdef = MK_WDEF(wdef%,wattr%,wptr,ltab
,inftab,apptab)
```

Here, wtpr, ltab, inftab and apptab are 'level II pointers'. In the last instalment, we stopped at these level II pointers, and more specifically after having explained "wptr", the pointer towards a sprite definition, sprite which will be used in the window for the mouse pointer. We now know how to define sprites.

So let's have a closer look at the other Level II pointers, and first 'ltab', the loose items table, or pointer towards the loose menu itmes list.

B- The "loose menu" items list

or: Of menus and items.

The concept of a "menu" probably does not need much more explanation: a menu is just a set of options proposed to the user, who makes his choice either by hitting a key corresponding to the option, or by clicking on the option with the mouse.

An "**item**" of a menu is simply one of the choices of that menu. In older programs (such as Quill), a menu is displayed as a regular list, such as:

F1 = action 1
F2 = action 2
F3 = action 3
F4 = action 4

and so on. This kind of menu, whilst regular, is also boring as it is generally bundled closely together in the window, and it is difficult to show, at the same time, other information in addition to the menu choices. It would be nice to have the menu items anywhere in the window, instead of having them in a regular grid as shown above.

This is what a 'loose' menu allows us to do. As the term implies, the items of such a menu are 'loose', i.e. can be anywhere in the window, they don't have to be in a rectangular grid - but they are still part of the same menu. The advantage is that, whilst the items are part of a regular structure (and thus easily recognizable), they are also placed where they can be used to best effect. The structure of the menu is regular in that the items will have the same appearance, but the items do not appear one after the other in the window. This is why it is a "loose menu".

When you define a loose menu for the window, you will have to define what each menu item in this menu is and does. Also, as the items are part of the same menu, they will have some properties in common (their general appearance). Some other properties will depend on each item

(such as the key which actions each item - it would be unfortunate if that were the same for each item). You must determine all of that.

The common properties are those that define the general aspect of the items: what colour is used as the "paper" or "ink" to display the items, what type of border they will have, etc. Take for example the QPAC2 "Files" window: the menu choices offered (Command, View, All, ESC and so on) have the same general aspect (same colour, same ink, same paper, same border colour when the pointer is in them etc...): they are all items of the same loose menu. They all change similarly when actioned or when the pointer moves over them. So they all change "status" in a similar manner. Let's take a closer look, first, at item "statuses".

1 - Item Statuses

Actually, loose menu items can have four different **statuses**: The first status, is simply that of a normal item which you can hit or do. It is said that this item is "**available**". The second status is where, for any reason, you can neither "**hit**" nor "**do**" the item: the item is "**unavailable**" and cannot produce any action. The third status is that of an item over which the pointer is just hovering: this is now the **current** item, and a border is drawn around it - if you HIT or DO in the window, it will be this item that is actioned. The last status is that of a **selected** item, which is what happens when you hit an item and it stays emphasized - such as the View item in QPAC2-Files.

The definition of the four statuses will be common to all items of a loose menu. This seems logical, and avoids confusion: if red paper with black ink meant that one item was selected, but meant that another item was unavailable, this would confuse the user to no end. Thus, the definition of the colours used for these different statuses are the same for all items. This provides the regularity which enables the user to recognize loose menu items instinctively as such.

To each status for the loose items thus correspond "**item attributes**". The item attributes are common to all items, and define the paper, border and ink colours for each status.

Some other aspects, however, may be different for each item. In fact, it is as if each item had a "window" with a content. Thus, for each item, you should indicate what the size and position of its "window" should be, and also its content and type (text or sprite). You also define its "selection key" and so on. The selection key is the key you hit to have the corresponding item produce an action - F4 for the "View" item in Qpac2.

2 - Making the loose menu items list

All the data for these items is grouped together in a list called "loose menu items list" or "Loose Items List" (LIL). This list contains the common definitions for all items, and the different information for each item, one after the other. To make this list, you should use the function ***MK_LIL*** (**M**a**K**e **L**oose **I**tem **L**ist):

```
ltab = MK_LIL (lattr, lsiz%, lorg%, ljus%,
key$, ltyp%, lstr$, lspr, lblb, lpat)
```

ltab is the result of this function and is a pointer to the loose items list. The parameters are as follows:

-› * **lattr**. This is an array of dimension DIM lattr(3,3). It contains the item *attributes*. These are the different colours/borders which show the different statuses of the items. As mentioned above, these are common to all items.

- lattr (0) (i.e. lattr (0,0), lattr (0,1), lattar (0,2) and lattr (0,3)) contains, in this order, the size and colour of the border of the current item, in lattr (0,0) and lattr (0,1). lattr(0,2) and (0,3) are unused.

- lattr(1) contains the paper and ink colours for unavailable items in lattr (1,0) and lattr(1,1). The two other elements of lattr (1) (i.e. lattr (1,2) and (1,3)) point to a "blob" and a "pattern" (more of which later): in general, though, **they are left empty**. For my part, I don't think I've ever encountered a program where they weren't left empty i.e. 0 (just putting myself out on limb here, of course).

- Next, same thing for available items(lattr 2,0 et 2,1)

- Next, same thing for selected items (lattr 3,0 et 3,1)

-› * **lsiz%**, **lorg%**, and **ljus%** are integer arrays of dimension DIM (n-1,1) where n is the number of items in the menu (numbering starts at 0). Thus, if you have three loose menu items, you'll DIM the arrays DIM (2,1). Element 0 (i.e 0,0) and (0,1) then contains information

about the first loose item (item 0) element 1 of the array contains information about loose item nbr. 2 and so on. Just what the information held in the elements is, is explained here:

a) **lsiz%** contains the x and y sizes of each item. One could say that these are the size of the "window" for each menu item. It is this window whose paper colour will change when the item becomes selected. Each element of this array contains, in element (n,0) the X size of the window, and in element (n,1) the Y size of the window.

b) **lorg%** contains the "origin", i.e. the x and y position of this "window" for each item. The position is given as the top left corner of the "window", in pixels, **and relative to the origin of the (primary or secondary) window containing the loose menu items**. Each element of this array contains, in element (n,0) the X position of the window, and in element (n,1) the Y position of the window.

c) **ljus%** is the x/y justification of the content of the item with respect to its "window" (i.e. if the item contains a text, is the text centered, is it flush to the left, or to the right?) The "window" for a loose item can be larger than its content, and then it is important to state where the content should be. For example, the 'F6 Sort' item in the QPAC2 Files menu generally has a window that is larger than its content, which can be seen when you move the pointer over it: the border around the item is larger than the content of the item. With the ljus% parameter you indicate the number of pixels from where the content of the item should be drawn or printed, with respect to the top left corner of the item's "window". If this parameter is 0 in any of the directions (x or y), then the item will be **centered** in that direction. Each element of this array contains, in element (n,0) the X justification of the content of the window, and in element (n,1) the Y justiciation of the content of the window.

–› * **Key$** is a string that contains the selection key for the items. The selection key for an item is the key to be pressed to hit/do the item. KEY$ is one large string made up of the selection keys for each item, so that key$(0) = the selection key for item 0, key$(1) = the selection key for item 1 and so on. Thus Key$ is composed as follows:

```
key$=chr$(n1)&chr$(n2)&...&chr$(nx)
```

i.e. exactly ONE keypress character per item, until x items. The first is for the first item, and so on. You can also write:

```
key$="A"&"B"&"C" etc...
```

If you do not wish an item to have the possibility to be hit/done with a keypress, use CHR$(0) in the string for the keypress for this item.

The character in question MUST be put in UPPER CASE, (i.e. either "A" instead of "a" or CHR$(65) instead of CHR$(97)). It doesn't matter, later on, whether the uses presses the key in upper of lower case, but here at the definition stage, you MUST give it in upper case.

There are also some special characters:

CHR$(1)= Hit= SPACE/left mouse button (not to be used as selection key)
CHR$(2) = DO = ENTER/Rightmouse button (not to be used as selection key)
CHR$(3) = Cancel = ESC
CHR$(4) = Help = F1
CHR$(5) = Move window = CTRL F4
CHR$(6) = Change size = CTRL F3
CHR$(7) = Wake = CTRL F2
CHR$(8) = Sleep = CTRL F1

Thus, if you have an item the action of which will move the window (it should then have the standard sprite for that, as well), the key$ for this item should be CHR$(5), and thus, each time you hit the standard CTRL-F4 combination to move the window, this item will be actioned. You COULD conceivably use any other key, but it really is better if you use the standard keypresses for these standard items!

All of these actions should be quite clear, except perhaps wake and sleep: Try CTRL F1 and CTRL F2 in QPAC 2, and you will notice that sleep puts the program to sleep as a button, CTRL F2 wakes it up again, and refreshes the menus.

Of course, you are not required to provide for buttons, wake or even window move events in your programs. If you do provide for this, however, it is suggested that you use the standard keypresses for the items concerned.

–› * **ltyp%** is an array of dimension DIM (n), i.e. one single element per item. This array de-

termines the item type. There are four types:

0 = the item is a string
2 = " " " " sprite
4 = " " " " blob
6 = " " " " pattern

Once the type is determined to be one of the four above, you can then add nothing, 256, -256 or other negative numbers to it. This changes the behaviour of the item:

- If nothing is added, different actions result depending on whether the item is "hit" or is "done": when the item is "done", the program comes back from reading the pointer (as we shall see later) but if you only "hit" the item, the item will only change status beteween selected and available a(and back) and that is all.

- If you add 256, the item, even when it is "hit", will cause a return from the read pointer loop, as if it was done. Thus, there is no difference between hitting and doing (!). Also, the item's status is immediately reset to available.

- If you add -256, a hit and a do are, again, the same, but the item is not reset immediately to available.

- You add other negative numbers, but only to text items. If you do that, you will cause a letter in the item (if it is a text!) to be underlined automatically. This is covered in more detail a bit later.

-› * **lstr$,lspr,lblb,lpat** are the arrays containing the content of the items: lstr$ contains strings, lspr contains pointers to sprites, lblb points to pointers for blobs, lpat points to addresses for patterns (we shall see the definition of blobs and patterns later - they are very seldomly used for loose items). Thus, if you have determined (by type%) that the first item is a string, the first element of lstr$ contains this string.

The arrays for these pointers are DIMmed to DIM (n), with lstr$ being dimmed to (n,max_length_of_string) as is usual for string arrays. They parsed - and must be filled - for each corresponding item, as referred to by the ltyp% of the item. Let's suppose we want 3 items, the first one a text item ('HELP'), the second one a sprite items (window move), the third again a text item ('ESC') and the fourth another sprite. We will then DIMension the ltyp% array for four elements: DIM ltyp% (3). The contents of ltyp% will be:

ltyp%(0)=0 (string)
ltyp%(1)=2 (sprite)
ltyp%(2)=0 (string)
ltyp%(3)=2 (sprite)

We will then DIM lstr$(3,10), lspr(3),lblb(3) and lpat(3).

lstr$(0) will contain "HELP", lstr$(2) STAYS EMPTY (0 string), lstr$(2) will be "ESC" and lstr$(3),stays empty again.

lblb and lpat will remain empty (all elements set to 0). Likewise, lspr will be empty except for lspr(1) and lpsr(3) which will each contain a pointer to a sprite (as explained in the last instalment of this series).

The MK_LIL will automatically choose the correct items from the correct arrays, depending on the type of the item. This can be one the worst problems with the QPTR function, i.e. fill in these arays wrongly...

Next time, we deal with automatic underlining of a letter in a text item and information sub-windows.

New Q-Word Game coming soon

Phoebus Dokus informed us about a new project from RWAP, Geoff Wicks and himself. Here is a short description, and have a look at the screen shots! As it is supposed to be ready for XMas, that's the last chance to report about it before its release. We hope to have a review for you as soon as the product is released (hint, hint, for both RWAP and reviewers!)

Q-Word is a word puzzle game that's a fusion between Tetris, Scrabble and your Sunday newspaper's "Find the hidden words" puzzles. Q-Word runs on hi-resolution, hi-colour screens and its the first QL commercial game of its kind to use digital sound (as either a CD sountrack on QPC-QXL or via SSS Q40/Q60/Amiga). There is also planned support for Q-Midi (Via the NET ports on regular QL/QXL and Aurora and in the future via serial on all platforms or Standard Midi UART on uQLx). Q-Word is based on the Q-Typ dictionary, therefore it practically can be used in all languages a Q-Typ dictionary exists. The follow-

that this happens because of the Spring Equinox being set in March.

I hope that this is of some interest to someone!

Footnotes:
(1) To be absolutely correct -term wise-: "The One, Holy, Orthodox Apostolic and Catholic Church" (Orthodox: "The one that preaches the CORRECT truth" from Orthos: Correct, Doxa: Belief, Rite and Catholic: The one for ALL- from OLA=All, Everything) as it is its full title - I am sure someone will have some use for this trivial information:-)
(2) This was adopted by the -then- Hellenic Kingdom a little after the Olympic Games due to some funny circumstances with foreing correspondence from the Games... ie the letter arriving at a date in say the UK before it was sent :-) (I've seen some of those in my years working for Vlastos Philatelic Centre and it was rather interesting as the first thought that comes to mind is that Mr. Spock is right... Time Warp IS possible :-)

Programming with QPTR - Part 4 - The level II pointers

Wolfgang Lenerz

Last time I left you with the promise to explain automatic underlining of text items. So here it is:

3. Automatic underlining of a letter in a text item

You will probably have noticed that in many cases a letter in a text loose menu item is underlined (generally, but not always the first letter). This serves to indicate to the user that this letter is the selecton key for this menu item. For an example, you can look at the "Command" menu in the QPAC 2 Files program.

This of course is a very nice possibility and, provided you have QPTR version 0.08 or higher, you can also make use of this in your own programs.

As was mentioned last time, to obtain this automatic underlining, you have to add something to the type of the item. Remember, this works only with text items - and you can only underline one letter per item, of course.

In principle, to obtain automatic underlining, you subtract 2 from the item type to underline the first character of the item, 4 to underline the second character in the item text, 6 for the third and so on - in fact, you subtract twice the position of the letter in the item text.

In practice, however, this will generate an error if you use an underlined text item and add -256 to it (to obtain a return even when the item is "hit" and not "done"). The combination of a negative item type and a negative addition to it, makes QPTR hiccup and refuse the item type.

Hence, to obtain underlining in a text item where you also want to use the -256, you should use the following item types:

254 = text with first letter underlined
252 = text with second letter underlined
250 = 3rd letter

and so on. I think you can see the progression.

If you want to use this possibility, though, you should slightly change the RD_LOT procedure that comes with QPTR (and the use of which is, of course, highly recommended). I have made these changes, and here you can find the procedure as it stands now:

```
DEFine FuNction RD_LOT (lattr,nitem)
 LOCal count(3)
 LOCal item, ltyp, a$, lsk$
 LOCal ldef%(nitem-1,6), lptr(3,nitem-1)
 LOCAL lstr$(nitem-1,85)
 lsk$=''
 FOR item = 0 TO nitem-1
  READ ldef%(item,0), ldef%(item,1)
  READ ldef%(item,2), ldef%(item,3)
  READ ldef%(item,4), ldef%(item,5)
  READ a$: lsk$=lsk$ & a$
  READ ltyp
  ldef%(item,6)=ltyp:ltyp=(ltyp MOD 256)/2
  IF ltyp>10 or ltyp<0:ltyp=0
  IF ltyp
   READ lptr(ltyp,count(ltyp))
  ELSE
   READ lstr$(count(0))
  END IF
  count(ltyp)=count(ltyp)+1
 END FOR item
RETurn MK_LIL(lattr, ldef%(TO, 0 TO 1),
ldef%(TO, 2 TO 3), ldef%(TO,4 TO 5), lsk$,
ldef%(TO, 6), lstr$, lptr(1), lptr(2),
lptr(3))
END DEFine RD_LOT
:
```

As you can see, the changes concern the handling of ltyp...

Ok, now the handling of menu items has no more secrets for you.

C - The information subwindow definition list

As mentioned in previous instalments of this series, menu items all have a certain action, the do something. This is not true for "information sub-windows" - they are there only to DISPLAY some sort of information, or used just to draw borders within the window. If you look at the "command" window in the QPAC2 Files program, you can see that the window is divided into three parts: the upper part, containing the name of the window, a middle part framed by a green border (it contains some loose menu items) and the lower part with commands that are not included within the border. This border was drawn with an information sub-window, whose only function here is to draw that border.

Contrary to loose menu items, information sub-windows do not have to have common attributes. They can be as disparate as you wish them to be. Moreover, the content of each information sub-window can be completely different, not only from the content of other information sub-windows, but even from another part of the content of that same information sub-window.

Thus, when building the list of the information sub-windows, this list will be substantially different to that for the loose menu items. In fact, we will have several lists: one general master list, containing pointers to the information sub-windows, and then one list per information sub-window.

Here with the level II pointers, we are only concerned with the master list, which contains information for each sub-window, as well as pointers to other information. The information contained in this master list is concerned with the "physical definition of each sub-window (size, origin et al). The pointers to other information point to information about the content of each sub-window.

To build this master list, we use the following function: ***MK_IWL*** (***M***a***K***e ***I***nformation Sub-***W***indow ***L***ist)

```
inftab = MK_IWL(iwdef%, iwattr%, infolist)
```

where:

–› * **iwdef%** is an array containing the physical description of the windows. It has a dimension DIM (n,3) where n is the number of information subwindows-1. For each array element z, the array contents are:

- window x size (z,0)
- window y size (z,1)
- window x origin (z,2)
- window y origin (z,3)

The origins are the top left corner of the window with respect to the top left of the primary (or secondary) window containing the information sub-window.

–› * **iwattr%** is an array with the attributes of the sub-windows. It is again an array DIM (n,3) where n is the number of information sub-windows -1. For each array element z, the array contents are:

- Shadow "depth" - this is actually ignored for information sub-windows and should be left at 0.
- border size
- border colour
- paper colour

of the information sub-window, in that order.

–› * **infolist** again is an array, but not an integer array. It is an array of pointers towards the lists containing the content of the information sub-windows. These pointers are obtained with a level III function (MK_IOL), which we shall look at later. There is one such list per information subwindow (or else the pointer is left at 0).

D - The application subwindow list.

Here again, this is a master list. It is, again, different from what has gone before. Actually, it contains no other information than pointers towards application sub-window definitions. Indeed, for each application sub-window, we must establish one definition. The pointers to these definitions are united into this single master list.

Like information sub-windows, application sub-windows do not necessarily have common characteristics, they can be very different from each other. This is why the master list contains only these pointers to the application sub-window definitions.

To build this list of application subwindows, we shall use the function ***MK_AWL*** (***M***a***K***e ***A***pplication sub-***W***indow ***L***ist)

```
apptab = MK_AWL(appsubwin(n))
```

appsubwin is an array containing the pointers to the application sub-window defintions. For "n" application sub-windows, you will DIM this array (n-1). It will be filled in with pointers supplied by the MK_APPW function:

```
appsubwin(0)= MK_APPW (level III parameters)
appsubwin(1)= MK_APPW (levem III parameters)
```
etc...

If your window does not have any application sub-windows, apptab is just 0.

This finishes level II - so let's continue right away into level III.

The Level III Pointers

The Level III commands and functions are used to fill in the contents of the sub-windows (information sub-windows and application sub-windows). As was already mentioned, the content of the primary window is made up of the loose menu and the two types of sub-windows, its contents are thus defined by them. The loose menu is alraedy entirely defined in levels I & II so there now only remains to fill in the content of the sub-windows.

A - The information sub-windows

The "physical" definition (i.e. size and origin) of these windows was already given in Level II. Here in level III, we only define what is ***in*** the sub-window. The content of such a subwindow is made up of "***objects***". An object may be anything: a text, a sprite, a "pattern" or even a "blob". For example, if the sub-window is to contain the words "Joe was 'ere", the object is the string "Joe was 'ere", and it is an object of type text. We have already met objects: the content of a loose menu item can be a text, a sprite, a "blob" or a "pattern" - this is, in fact the "object" of this loose item. The same is true for information sub-windows but an information sub-window can contain several objects whereas a loose menu item can only contain one object.

To use the above example, if the information sub-window is supposed to contain the string "Joe was 'ere", this text could be the object of the sub-window. But I could also say that the word "Joe" is the first object of the information sub-window, the word "was" is object number 2, and "'ere" object number 3. The window thus would have three text objects.

Agreed, in the above example it would not make much sense to have three objects where one would do the trick (and even so: see below). However, you could have a text in front of, or next to a small sprite. Then you would have to define two objects, one a text, the other a sprite.

By now, you will have guessed that you will need to build up a list of information sub-window objects. This is achieved with the function ***MK_IOL*** (***M***a***K***e ***I***nformation sub-window ***O***bjects ***L***ist):

```
listobj1 = MK_IOL (isize%, iorg%, imod,
itype%, istrg$, ispr, iblb, ipat)
```

Here, listobj1, the result of the function, is a pointer to the list of the objects.

The parameters to this function are not very complicated (hereafter, "n" is the total number of objects in the information sub-window to put on the list):

-› * ***isize%*** is an integer array of DIM isize% (n-1,1). For each object x, isize% (x-1,0) is the x-size and isize% (x-1,1) is the y-size of this object (remember, numbering starts at 0).As usual, the sizes are given in pixels.

-› * ***iorg%*** is an integer array of the same DIMensions and contains the x and y origins of the object within the information sub-window. (0,0) is the upper left hand of the information sub-window.

-› * ***itype%*** is again an integer array, but of DIMension itype%(n-1). It contains information on the type of object (same as for loose menu items). Here again, you can provide for automatic underlining of any letter in a text object, by varying the type parameter just like for loose items: (254=1st character is underlined, 252 = 2nd character is underlined and so on).

-› * ***istr$***, ispr, iblb and ipat are string arrays (istr$) or floating point number arrays (the others) and they contain, just like for loose items, the objects themselves, i.e. the strings (istr$), sprites (ispr) blobs (iblb) or patterns (ipat). Each object can be of any type.

-› * ***imod*** is a floating point array and contains possible additional information on each object:

* If the object is a sprite, there is no additional information.

* If it is a blob, you must insert here the address of a "pattern", and if it is a pattern, give the address of a blob. Generally, instead of refering to blobs and patterns, you might consider using sprites.

* If the object is a text, you must give the ink colour of the text, and the size of the text (like in the CSIZE command). This data is combined as follows:

Ink * 65536 + Csize_x * 256 + Csize_y.

Thus, if the object is to be a string which is to be printed in red and big letters (i.e ink=2, csize=3,1), this becomes:

2 * 65536 + 3 * 256 + 1 = 131841.

Thus for this object, imod (x-1) would contain 131841.

It follows that if I want a string ("Joe was 'ere") where Joe would be printed in big red letters, the rest in normal colours, I would need two objects, one for "Joe", the other for the rest.

Strangely, in the parameter list, the imod parameter precedes the type% parameter, even though it is the type% parameter that determines what the additional information is - but that's the way it is.

You should build up a list for each information sub-window (unless the sub-window is empty - then the pointer is 0).

You will thus write:

```
listobj1= MK_IOL(...)
listobj2= MK_IOL(...)
```

and so on, one for each sub-window. Once the lists for the sub-window have been made, then you must regroup the pointers to the list in another array, as follows:

```
DIM infolist(n-1)

infolist(0)= listobj1
infolist(1)= listobj2
...
infolist(n-1)=listobjn
```

The infolist array is then one of the parameters to the MK_IWL function, which, as we have seen, is a LEVEL II function explained earlier.

OK, that's it for today.

More in the next instalment, where we'll look at some more level III parameters.

TK2 on MAC QL Emulator

by Al Boehm

About how to install TK2 on the MAC Q-muLator:

The Q-emuLator web page has changed. It is now:

http://users.infoconex.com/daniele/q-emulator.html

However, that won't help with the MAC version since that page is still being updated.

It's been some time since I ran Q-emuLator for MAC and I am still looking for the paper manual which is probably within 8 feet of where I am sitting. As soon as I find it, I will give you more definitive instructions. If I don't find it, I will email Daniele for info.

As I recall, there are two steps to installing the TK2_ext.

1. Get a copy of TK2 (Tony Tebby has OKed free use of TK2 on emulators). If you have a hard time finding a copy of TK2, I will send you it via email. It's not very large.

2. Use the the CONFIGURE menu to install the copy and then save the configuration. I remember this was pretty straightforward but I do need that manual to be exact.

Editor's comment: if YOU are using the latest MAC QL Emulator, why not write about it? Other readers may be very interested in your experiences? I still get asked by Mac users and can't refer to anything recent. Also, if you run QPC under RealPC or Virtual PC on the Mac, please tell us and others about it. Best, if you use both and let us know the advantages and disadvantages of each system.

Programming with QPTR - Part 5 - The level III pointers

Wolfgang Lenerz

We finished the level III pointers for information subwindows last time. Now it's time to explain those for application subwindows:

B - Application subwindows

There are two kinds of application subwindows. First, there are "**menu application subwindows**". These contain elements which behave like loose menu items: they can be selected, clicked and actioned. A typical example would be the QPAC2 "Files" menu - the names of the files which are displayed are part of an application subwindow: they can be selected, and, if you DO them, they produce an action. The "menu items" of the menu application subwindows are displayed in a grid. This makes a nice contrast with the loose items.

The second type of application subwindow is the "**simple**" application subwindow. This does not contain a menu, in fact it is empty. Since it doesn't contain anything, it is easier to define than a menu application subwindow.

As we have seen for the LEVEL I definitions, there is a list of application subwindows, composed of pointers (addresses) to the subwindow definitions. This thus must mean that application subwindows also have definitions... and, indeed, there is one definition per application subwindow. This definition is built with the ***MK_APPW*** (***MaKe APP***lication sub-***W***indow definition) function:

```
appsubwin = MK_APPW(awdef%, aattr%, aptr,
akey$, x_ctrldef, y_ctrldef, xoff%, yoff%,
x_spac, y_spac, xindex, yindex, linelist)
```

Phew!

Let's start by the easiest bit: The first four parameters. The first two of these (i.e. awdef% and aattr%) are identical to the first two parameters of the ***MK_IWL*** function which was described in the last instalment of this series. They determine the "physical" definition of the window (adef%) and the window attributes (aattr%) as follows:

-› * **awdef%** is an array containing the physical description of the application subwindow. It has a dimension DIM (3). The array contents are:

- window x size (element 0)
- window y size (1)
- window x origin (2)
- window y origin (3)

The origins are the top left corner of the window with respect to the top left of the primary (or secondary) window containing the application subwindow.

-› * ***aattr%*** is an array with the attributes of the subwindows. It is again an array DIM (3). The array contents are:

- Shadow "depth" - this is actually ignored for application subwindows and should be left at 0.
- border size
- border colour
- paper colour

of the application subwindow, in that order.

-› * ***aptr*** is the address of a pointer sprite for this application subwindow. Thus, each application subwindow may have a pointer sprite that is different from the main window pointer sprite!

-› * ***akey$*** is the "selection key" of the application subwindow - this is used to bring the pointer directly into the application subwindow. Moreover, if the application subwindow is a menu application subwindow with a scroll bar hitting this key will bring the pointer:

- first to the centre of the application sub-window, if the pointer was not already in the application sub-window.
- then , if you hit it again, onto the scroll bar (if any!)
- then back to the centre of the application subwindow
- and again onto the scroll bar - and so on...

Just like for loose menu items, this selection key must be passed to the MK_APPW function in upper case. Generally, the TAB key (chr$(9)) is used, if there is only one application subwindow.

It is possible to define application subwindows with these first four parameters only. In this case, we have a simple application subwindow, and the call to RD_PTR (see below) will come back each time the pointer has moved or a key was hit (provided, of course, the pointer was in the application subwindow!).

If, however, you wish to define a menu application subwindow, you must fill in more parameters which will be explained below.

IV - LEVEL IV: Defining rows and columns

Before starting on this, let's see what a menu application subwindow consists of. This is one of the most complex aspects of QPTR programming - again, it is not difficult, there are just many parameters to learn (and remember)... However, if there are many parameters, this also means that you will have a large freedom to set up these windows (else the parameters wouldn't be of any use).

A - The components of a menu application subwindow

As we have seen above, the first parameters of an application subwindow are normal: size and origin of the subwindow, colour and size of its border, pointer, "paper" colour and "selkey". These parameters shouldn't be complicated.

Apart from that, an application subwindow is nearly entirely composed of "objects", i.e. the items of the menu. As mentioned, these are similar to loose meny items, but are arranged in a grid of rows and columns.

If need be, one my also add the scroll/pan bar and scroll/pan arrows. You can clearly see this in the "Files" menu of QPAC 2 , where all of these elements are visible. The "objects" are, of course, the filenames.

Just as a reminder: when the window can be scrolled up and down, then this is a "scroll". If the same is possibel for left to right, then this is a "pan".

1) The objects

As mentioned, the objects are items, quite similar to loose items. Here again, you must make a list of these objects and specify the type of each object (text, sprite etc...). In most cases it will be text, but not necessarily so, as Jérôme Grimbert shows in these hallowed pages of the august magazine (see his series on XMenu).
For each object, you also specify a possible selection key, the content type (i.e. the text), the content itself and the position in the grid. As you can see, quite a long number of parameters. This list is made up of Level V parameters which will be detailed later, and is built with the MK_AOL function.

Let's suppose for now that the list has already been built and that "objlist" is the result of the MK_AOL function.

Since the objects of an application subwindow behave similarly to loose menu items, the current item is also surrounded by a border, like the current item in a loose menu. The objects can be selected, thus changing their status, and if an object is "done", it may produce an action.

Here again, like for menu items, you will have to determine the **attributes** of these objects: the colours for the different statusses, and the colour and size of the current menu item border. These are common for all items of an application subwindow. Of course, different application subwindows may have different colours (I'm not sure whether that would be a good design practice, though).

This is where the similarity with loose menu items ends, as here we do not have "loose" items, but "bound" items - they are bound to each other and part of a grid of rows and columns.

2) Columns and rows

Since the items are part of a (hopefully regular) grid, we must specify how these objects are to appear in the grid. Whilst this is necessarily in rows and columns, you can specify how many rows and columns there are to be. The columns for each row need not be identical.

In most cases, the most important element is the row. You must determine which object(s) can be found in which row. Thus you must establish a **row list** which clearly states what row contains which objects, e.g. show that it contains objects a to b. The next row then contains objects c to d etc... If there is only one object per row (e.g. The QPAC 2 "Files" menu with "Statistics" switched on) this is not really complicated: you just indicate that object 1 is in row 1, object 2 in row 2 , 3 in 3 and so on.

If there are two objects per row, you will indicate that objects 1 and 2 are in row 1, objects 3 and 4 in row 2, 5 and 6 in row 3... - you get the picture.

(This row list is made with the MK_RWL function, commented below).

So, by now we will have indicated the content and parameters of each object, and in which row each object is going to go. Now you have to determine the size of each row, by determining the size of each column.

Each column as two sizes: the 'hitsize' and the 'spacing' between objects. There is one of each per column in the row.

The hitsize is the maximum size of an object in one column of the row - this actually defines the column size. It is this size that will change colour according to the status of the item. Again, look at the "Files" menu - if you click on a filename, it is not only the paper under this filename that changes colour, but the whole area that goes up to the second column (if any), and this, whatever the length of the filename may be. It is also that area which is outlined by the border when you bring the pointer over it, showing that this is the current item.

The "spacing" determines the number of pixels between the beginning of the hitsize of the first column and that of the hitsize of the next column, if any (and then the next columns, if any etc...). Clearly, the spacing must be at least as large as the hitsize, and ideally a bit larger (so that the border around the current item can be shown).

Let's presume that we have four rows with three columns each. And let's further suppose that the objects in the second column will be longer than those in the first column. I could then define the hitsizes and spacings as follows:

column one : hitsize 50, spacing 54
column two : hitsize 70, spacing 74
column three: hitsize 40, spacing 44.

The numbers correspond to the sizes in pixels: the first column will have a hitsize of 50 pixels and a total space (spacing) of 54 pixels. There will thus be at least 4 pixels between the object in that column and the object in the next column. You should make sure that the column is at least as long as the largest object that can go into it. If not, the object will be cut (if it is a text) or even not drawn at all (if it is a sprite).

You determine the hitsize and spacing for each column of each row. By doing this, you build up what is called the "**spacing list**". There is no need to have each column in each row to be the same size as that of the rows above and below. You don't even have to have the same number of columns in each row. Again, I consider it to be good programming practise to have a regular grid. It does make presenting the data easier.

It seems obvious that if you add up the spacings of each row in each column, you should get the size of the subwindow. It is possible, however to exceed that size, in which case the application subwindow becomes "pannable".

Likewise, if the combined height of all rows exceeds the height of the window, the window becomes scrollable.

3) Sections

An application subwindow can be cut up into several independently scrollable (or pannable) "sections". Each section can be scrolled independently, but they all show potentially the same data.

Sections are not necessary for application subwindows. If you take the QPAC2 Files menu for example, there are no sections. Let us suppose there were, though. If you have so many filenames that the window becomes scrollable, you could cut up the window into two sections. The window would be split up horizontally into two sections, there would be scroll bars for each section.
In principle, each section has its own scroll bars/scroll arrows (and pan bars/pan arrows of course). That way, you can see, at the same time, the start and the end of your data (in this case, the filenames.

It is important to realise that all sections may use the same rows and columns, and thus you can see all of the data in each section - you just have to scroll through it.

The user doesn't have to use the same different sections. In general, when the user brings the pointer to the scroll bar (NOT the scroll arrows) and "does" on the place where the two sections come together, the sections are joined and become one.

4) The control definition

The **control definition** tells the pointer Environment:

* how many sections there are
* how many rows there are in each section
* at what row each section starts
* where each sectionstarts in the window

OK, we have now seen the different elements that make up the application subwindow. So let's start defining them.

B - The parameter

First of all, you may have noticed above that two parameters to the **MK_APPW** function were left unexplained:

-› * **xoff%** This parameter just gives the number of pixels between the left border of the window and the first object on the left of the window. This applies the the first column of all rows of the applicaton subwindow. If left at 0,the first object will be right up against the left hand side of the application subwindow.

-› * **yoff%** is the distance, in pixels, between the uppermost visible row and the upper border of the application subwindow.

Ok, that's it for this time. Next time, we'll continue looking at level IV parameters.

QTrans Review

John Perry

QTrans is a quaintly named file copy and transfer utility. While QL file handling programs are ten a penny, this is one of the ones which does stand out from the others. This review is of version 1.03 which had just been released at the time of writing this review. In fact, it was the latest in a flurry of releases. For a start it's pointer driven. It's claimed to be a precursor to a full blown GUI (Graphical User Interface) for QL systems. And it has quite a few novel features, like the dual file listing windows enabling you to see simultaneously the list of files on both the drives you are copying files from and to. In addition it has all sorts of commands and facilities for just about any file action from viewing and printing to searching and trashing.

Trashing? Well, one of the novel features of this program is the Trash Can. This is a facility which lets files be deleted, but done so in a way that allows you to undelete later.

It's a fairly rudimentary form of 'recycle bin' or similar facility found on other computers. It is not a true 'delete' action but rather files are put into a special folder on a hard drive rather than being deleted as such. In fact, there is a choice of Delete or Trash commands, meaning you can choose how files are deleted (provided you remember to use the correct command of course!)

The author suggests you give this folder a short and unusual name such as WIN1_*_ or a single letter name if you prefer something easier to remember. I opted for WIN1_*_ as it made it less easy for me to use a command from BASIC, for example, to accidentally delete something from this folder! In use, it worked well enough even if the Trash directory seemed to fill up at an alarming rate the way I go through files! In fact, for each file trashed, it seems to create two files, one with the original filename, and another much shorter file with a filename suffix of _T which contains details of where the file came from and the original file dates - yes, it even preserves file dates if that's important! The content of the Trash directory can be viewed just like any other directory on your hard drive and restoring files is as easy as copying files normally. Either navigate to the Trash Can folder or simply hit or do on the little icon of a bin, then select the Untrash command and it'll offer the choice of whether to restore the file to the original directory it came from or to the current path, which is what it calls the directory content shown in the other window.

I've realised I'm letting my enthusiasm get ahead of me here, so let's start with a screen dump from the program to show you the basics of what this program is all about.

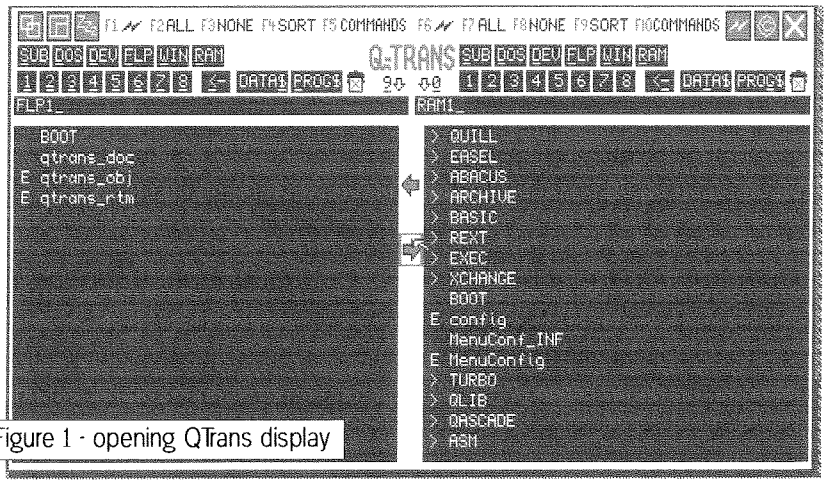


Figure 1 - opening QTrans display

So you can see where we have single stepped through the above code, and we are just about to jump to label 'T24_NOT_T30' because this instruction is not a type_30. Except, we know that it is an ADDX instruction because that is what I was testing, and ADDX is a type_30, so what have I done wrong?

I have tested bits 7 and 6 and found them both to be zero (because the Z flag was set after I stepped through the ANDI.W $C0,D0 instruction. This means that the jump should not be taken to T24_NOT_T30 because I have not yet ascertained that the instruction is not an ADDX. With bits 7 and 6 set to 00, I could be looking at ADDX or ADD. I should not be taking the jump until I have further tested the value in bits 5 and 4 as per my algorithm above.

This could be why the ADDX is being decoded as ADD, because I have the wrong condition in my test. In order to fix this, I have to change the source code, re-assemble and try my test again. I do this without the QMON2 first of all and if it still fails, I can use QMON2 to try and find out why again. I need to give the current job a 'G' instruction and then I can ESC from the decoding and exit the program.

I shall go do that and report back. Hang on here for a bit .....

Ok, I'm back. I made the change from 'BNE.S' to 'BEQ.S' and it worked fine. So it looks like I have correctly identified the bug. I need more testing though to make sure I cover all possible op-codes. I have followed up my ADDX testing by passing test files which have ADD, ADDA, ADDQ and ADDI instructions, along with assorted SUB variants and all appears to be working well.

So there you have it, an example of how I manage to get my code wrong and how I can use the tools available to try to sort it out. As I mentioned earlier, QMON2 is available from Jochen for a small fee, but only if you understand German manuals.

Laurence (Lau) Reeves has a different version of QMON2, written by himself, which fixes some bugs but I don't know if this is widely available or if it comes with a manual. Perhaps he could be persuaded to part with it or make it available - who knows. I'm not sure if he ever wrote a manual for it though.

See you next time.

Programming QPTR in SBASIC

W. Lenerz

Second Part - Displaying Windows

OK, by now I don't really know what part of this series we're in anymore... (Kudos to Herb Schaaf for keeping his numbers up!). However, we've arrived at the second section of this little walk-through.
Once we've defined our window, it's time to put it up on the screen. Do not forget that the first window to be opended is very important - it is the primary window, and all other windows (the secondary windows) must be within that primary window. The keywords for bringing windows onto the screen may be grouped into several sections: first, how to display the window in itself (I), then changing something within the window (II) and, last but not least, opening channel(s) within the window.
In the following explanations, I shall try to keep variable names coherent with what has gone before, whenever the same variables are to be used.

I - Displaying the window

There are two purposes for this. The main purpose, of course, is to display the content of the window. Second, one wants to make the window "managed" by the Pointer Environment. Indeed, only a window properly managed by the Pointer Environment may profit from all of the advantages granted by that Environment.
There are two keywords for displaying windows in the Pointer Environment. There is also a keyword to make an already existing window a "managed" window. Attention, we're talking about displaying the main (primary and secondary) window itself, not the sub-windows. There is no keyword to display the subwindows specifically - they are displayed automatically with the primary or secondary window.

A - making an existing window "managed"

The command "OUTLN" (OUTLiNe) makes an existing window managed and makes the Pointer Environment aware of the window. The window concerned is one opened with a normal "OPEN" command.

The syntax of OUTLN is as follows:

OUTLN [#channel,]xs,ys,xo,yo

The parameters are the same os for a normal open command: the window x and y sizes and the window x and y origins. "channel" is channel #1 by default.
Please note that, when working in S*Basic, the normal Basic windows (channels #0,#1 and #2) are not managed by the Pointer Environment in an automatic way.
However, for a successful programming session with QPTR under S*basic, channel #0 (Basic's "primary" window) must be managed by the Pointer Environment. Thus, channel #0 MUST be managed by the Pointer Environment at the start of the programming session - just use OUTLN for that.
If you don't do that, you will get many a bad surprise, as, notably, the pointer will not be read correctly, and your loose items will seem not to function correctly.

B - Displaying the window

Putting the window up on the screen is achieved with two commands: ***DR_PPOS*** and ***DR_PULD***, standing respectively for "***DR***aw ***P***rimary and ***POS***ition" and "***DR***aw ***PUL***l***D***own window". These are commands, not functions. They are very similar, being responsible for displaying the window on the screen and making it managed by the Pointer Environment. The difference is that DR_PPOS is used only for primary windows, and DR_PULD is used for secondary windows (also called pulldown windows, hence the name of the command). Moreover, DR_PPOS can use a channel parameter, while DR_PULD doesn't (the channel is opened automatically by that command).

The entire parameter list for the commands is:

DR_PPOS [#channel,] workdef, xpos%, ypos%, liflags%, appflags% [, xctrldef%, yctrldef%]

As mentioned above, the optional channel parameter does not apply to DR_PULD.

-› * ***Workdef*** is the working definition as returned by MK_WDEF

-› * ***xpos%*** and ***ypos%*** are integers which determine, in a very roundabout fashion, the position of the window. Indeed, obtaining the window's initial position is a bit counter-intuitive: xpos% and ypos% do not determine the x and y position of the upper left hand of the window as could have been expected. In fact, x and y determine the place where the POINTER will be on the screen once the window is drawn. The window is then drawn around this pointer position in such a way that the pointer is located at a predefined location within the window!

Indeed, we saw earlier that one of the parameters of the ***MK_WDEF*** command is the initial pointer position of the pointer within the window. Thus, when the primary window is drawn,the sequence of events, for the positioning of the window, is as follows:

The pointer is set to the xpos%,ypos% position given as parameter to the DR_PPOS command. Then the initial pointer location of the pointer within the window is looked up. After that, the primary window is drawn around the pointer in such a way that the pointer is located exactly where it should be within the window. As mentioned above, a pretty roundabout way of handling things...

Of course, determining where the window will effectively be drawn is easy, and can be calculated as xpos%-x and ypos%-y, where xpos% and ypos% are the parameters to the DR_PPOS command, and x and y are the parameters to the MK_DEFK function.

As can be expected, this pretty complicated way of positioning the window does have a reason - it is possible to set the xpos% and ypos% parameters to -1. In this case, the window will be drawn in such a way that the pointer is not moved at all. This is to make sure that windows can appear where the pointer is, so that the user's focus (which is generally on the pointer) doesn't need to change.

As a general way of doing things, this makes sense. The only difficulty arises when one wants a window to appear at a predefined position. I personally find the calculations to be made to ensure that the window appears at a certain position too complicated. So what I generally do when a window must appear at a certain position, is to set the initial pointer position within the window to 0. That way, the xpos% and ypos% parameters to DR_PPOS do determine the point where the window will be placed. After that, I just set the pointer position within the window with another QPTR command....

-› * ***liflag%*** is an integer array of DIMension (n-1) where n is the number of loose items the window contains. The array is used as a flag array, where each element of the array is a flag containing the statuses of the items - you might remember that each item can have several statuses (selected, available, unavailable and current item). When the window is drawn (and also later when the pointer is read) you will have to determine what status each item is to have - some items may be unavailable initially, or selected etc... This, of course depends entirely on the program. For a file copier, for example, the "Copy" item might remain unavailable for as long the the user hasn't chosen source and target directories.

Each loose menu item corresponds to one element in the array: liflag%(0) is for the first item, liflag%(1) for the second and so on. The values these flags can have are pretty simple, as follows:

-› 0 : the item is available
-› 16 : the item is unavailable
-› 128 : the item is selected.

There is no special value to indicate the current item, because the Pointer Environment itself automatically determines what item is the current item and then draws the border around it, and this depending on the pointer position. Thus, if you wish for any particular item to be the current item as soon as the window is displayed, you must set the pointer to such a position that it is "within" this item.

As the DIMension of the liflag% array depends entirely on the number of loose items, it is recommended to DIM this array at the same time one establishes the loose menu item list, because at that time one knows exactly how many loose items there are in the window.

-› * ***appflag%*** is the same thing as liflag%, but for the menu application window(s): there again, each object of a menu application window may have several statuses (the same ones as for loose menu items). There is one array per application subwindow, and they are DIMmed as follows: DIM appflag%(row-1,sec-1) where ***row*** is the number of rows for all of the objects and ***sec*** is the number of sections. If there only is one section, then one uses DOM appflag%(row-1,0).

-› * ctrldefx% is, again, an integer array of Dimension crtldefx% (maxsec%,2) where maxsec% is the number of sections as defined in the x control definition (horizontal). The values of this array are a bit special, as follows:

(0,0) contains the number of control sections (i.e. maxsec%).
(0,1) contains 1 if the control definition just changed, else 0.

Then, for each control section i:

(i,0) contains the start pixel position.
(i,1) contains the number of the first row shown.
(i,2) contains the number of rows in this section.

-› * ctrldefy% is, you will have guessed, the same thing for vertical sections and columns, instead of horizontal sections and rows.

Please note that the two last parameters are optional: if there is no control definition, they may be omitted or simply set to 0. However, there will be as many flag and definition arrays as there will be menu application subwindows (of course, they are not necessary for simple application subwindows). If you have several application subwindows, you may omit the flag arrays for those where you don't need them.

These are all of the parameters for the two commands, ***DR_PPOS*** and ***DR_PULD***. Both commands are similar, they display a managed window on the screen. As was mentioned above, the main difference between these two commands is that ***DR_PPOS*** is used for the primary window, whereas ***DR_PULD*** is used for secondary windows. ***DR_PPOS*** can use a CON channel (which must have been opened beforehand, the default channel being #1).

The problem with that is that **you have no channel number for secondary windows**. Indeed the ***DR_PULD*** command opens a window and a channel by itself (a channel of type CON) - but this channel is NOT accessible from S*Basic. There is no Basic channel number corrsponding to the window opended by ***DR_PULD***. This is different for ***DR_PPOS*** which can use a channel in which all the normal operations (PRINT etc) can be made. Thus, ***DR_PULD*** opens an **inaccessible** channel.

Moreover, there may be a certain number of problems when compiling. Indeed, in S*basic, one can practically not use the ***DR_PPOS*** command, as that would mean opening a primary window. But, as we have seen, window #0 of S*basic should be the primary window for the S*Basic job, and a job cannot have two primary windows. Channel #0 was made the primary window with the OUTLN command. Thus, in interpreted basic, you will rarely use the DR_PPOS keyword.

There are two consequences to this:

First, if the program is to be compiled later on, one should include some lines along the following:

```
IF compiled
  OPEN#1,"CON_"
  DR_PPOS (parameters)
ELSE
  DR_PULD (parameters)
END IF
```

Thus, you open a channel #1 (which for nearly all commands is an implicit channel) in a compiled program, and then you open a primary window. In interpreted Basic, you open a secondary window. Second, there may be a positioning problem when displaying secondary windows: indeed, like for primary windows, the positioning of the secondary window is achieved via the xpos% and ypos% parameters, which were described above. However, for secondary windows, these parameters contains the true coordinates (no muckling about with the pointer position here) where the window will be displayed. This however means that you cannot know where **within the primary window** the secondary window will open - the user may have moved the primary window from its original position.

Of course, there is a solution, as you can use the pointer positioning. If xpos% and ypos% are given as -1, the secondary window will open at the pointer position. You can thus set the pointer within the primary window to a certain position and then open the secondary window.

Ok, this is it for this time - in the next instalment we'll cover changing a window once it has been displayed. Is there anybody out there reading this series at all?

3D Perspective Animation - Part 3: Trees

Stephen Poole

In QL Today of march 2003, I mentioned trees as part of 3D Perspective Animations, but omitted the code to draw these from the article. This article will set that right. First of all, I must apologise to readers for not having divided my various 3D programs into modules, which would have meant less typing each time. This is because I did not expect to print so many articles from the start, otherwise I would have planned things out better. Mea Culpa!
In 1988, a french magazine printed an article entitled ' Growth of Plants' for the Atari ST. This 'interesting' program allows you to enter strings of characters which control the mathematical properties of plant-growth, but, patently, the code doesn't work, even on the ST. (I have probably lost hundreds of hours typing in programs from magazines that rarely work, even after a considerable ammount of debugging. One wonders if the magazines possess the necessary computers to test them on. ***[Screenshot attached, to show that we run the programs - Editor]*** But nowadays published programs are rather a rarity). I have also a book called 'Patterns in Nature' which describes plant growth, but unfortunately does not describe the formula for graphic output. So I had to write my own method from scratch, and decided to draw bifurcation diagrams viewed at an angle, with leaf-production by random plotting, and the result was satisfactory enough for my needs. For more details on simple binary-trees, see the forthcoming article in Quanta.
This program has been eventful for me as for the last 20 years I have been working uniquely on a monochrome monitor. So I promised Jochen I would link up my JS to our television set and experiment with 4 and 8 glorious colours (to improve the otherwise psychedelic output). However, I still prefer the look of the trees on my monochrome monitor, as 4 or 8 colours don't give sufficient graduations for my liking. No doubt GD2 is the answer... Remember that these trees are flat, so they must be drawn at an angle. They could be

Programming in Sbasic with QPTR - Part 6

Wolfgang Lenerz

This time we continue to look at the level IV parameters used to make menu application subwindows - indeed the most daunting aspect of the Pointer Environment.

One of the first parameters is the rowlist, which we make with the ***MK_RWL*** (***M****a****K****e*** ***R****o****W*** ***L****ist*) function:

```
rowlist = MK_RWL (objlist,rows(n,1))
```

The parameters for MK_RWL are as follows (where "n" is the number of rows - 1):

-› * ***objlist*** is the "object list". This will have been obtained by the MK_AOL function, explained later in Level V.

-› * ***rows*** is an array DIM rows(n-1,1) where "n", as mentioned above, is the number of rows wished. This array is filled in by determining for each row, which object is to be the start object of the row and which is to be the end object. Let's suppose, for an example, that we wish our objects to be in 4 rows with three columns each, and in the order the objects are found. The array will thus contain: rows(0,1)=0 (start of row 1) and rows(0,1)=3 - end of row 0. The row 0 thus contains objects 0,1 and 2. Next, rows(1,0)=3 and rows(1,1)=6 - row 1 thus contains objects 3,4,5. As you can see, for the end marker we use the next element: to state that object 5 is the last object of row 1, we set rows(1,1) to 6.

  For each row, you MUST give as many objects as there are columns for the row! It is unfortunately not possible to specify simple x elements starting from y, where x is the number of columns. Nor is it possible, say in row 2, to have object 2, followed by object 18 followed by object 6. On the other hand, you can specify that row 1 has objects 5 to 7, row 2 has objects 1 to 3 and row 3 has objects 3 to 5 - overlapping is thus possible. The object number used here is simply their place (index) in the list of objects that you have built up in level V: the first object in the list is object number 0, the second is object 1 and so on.

The next parameters that need explaining are ***x_spacing*** and ***y_spacing***. They contain the "spacing list". This is obtained by the ***MK_ASL*** (***M****a****K****e*** ***A****pplication subwindow* ***S****pacing* ***L****ist*) function, as follows:

```
x_spacing = MK_ASL (size%(n,1), indsize%,
indspacg%)
```

where:

-› * ***Size%*** is an array DIM size%(m,1) where m is the number of columns (not rows!). For each element i, size%(i,0) contains the hitsize and size%(i,1) contains the spacing of object i-1.

-› * ***indsize%*** and ***indspacg%*** are optional parameters: they are used for the "index bars", something which nobody has ever really understood. They are best left at 0, even though they are explained later on.

Of course, defining one spacing list is not enough - we will only have defined the object sizes in one dimension (the x axis), but what about the other dimension, the y axis? Defining the spacing and size for one dimension is not sufficient, we know the sizes from left to right but not those up/down. So we must build a second spacing list, for the columns this time. This list is build up in a similar manner to the x spacing list above:

```
y_spacing = MK_ASL (size%(n,1),indsize2%,
indspacg2%)
```

where n is, this time, the number of rows minus 1.

Right, we have built the spacing list - now we have to establish the "control defintion". This tells the Window manager how many "sections" there are in the window (in each direction) and at what row (or column) each section starts.
A "section" is just a collection of rows (or columns) that can be scrolled independently. It is as if you cut the window into 2 (or 3,4,5...) independent parts, each with its own scroll arrows. Many windows only have one single section, but several are possible.
If all of your rows and columns fit inside the window at once you don't really need sections and, such a control definition isn't really useful and it can be left at 0. In Sbasic, the control definition also allows you to determine the colour and size of the scroll arrows, in addition to the sections themselves.
The control defintion is made with the ***MK_CDEF*** (***M****a****K****e*** ***C****ontrol* ***DEF****inition*) function:

```
x_ctrldef= MK_CDEF (secmax%,arrcol%,
barcol%,barseccol%)
```

x_ctrldef is then one of the parameters for ***MK_APPW***.
The parameters for MK_CDEF are as follows:

–› * ***secmax%*** is the number of sections.

–› * ***arrcol%*** is the colour for the scroll arrows.

–› * ***barcol%*** and barseccol% are the colours of the index and section bars – again, leave these at 0.

With this, you have build a control defintion (here, for the x axis). The same applies if you want to have vertical sections:
`y_ctrldef = MK_CDEF (secmax2%, arrcol2%, barcol2%,barseccol2%)`

Contrary to the spacing lists, you do not have to have a control definition for each dimension. If you do not have a control definition for any direction, the pointer may be left at 0.
Later, we will also have to initialise a definition control array, as follows:
`DIM cty%(secmax%,2)`

This will be used in the read pointer loop.
cty%(0,0) contains the number of current sections: Even if you have provided for the possibility of 2 sections, there may be only one to start with, or there may once have been two but the windows have been joined later.
cty%(0,1) is ‹› 0 if the control definition has changed.

Then, for each section i, elements (i,0), (i,1) and (i,2) remain. The contain the following information:
* (i,0) is the y position, in pixels, of the start of the section within the window.
* (i,1) contains the number of the start row (i.e. the first visible row).
* (i,2) contains the number of rows visible in the section.

Indexes

The two last parameters for MK_APPW, i.e. the x and y indexes concern the index bars, and they may be left at 0. If you do fill them in, they must contain the hitsize and spacing lists for the indexes (just like the ones for the window). Here are some details about the indexes. Menu application subwindows may be provided with "indexes" which are printed outside the menu application subwindow – for example the number of rows and columns in a spreadsheet. To do this, you must fill in all of the parameters concerning the indexes. I presume that Qspread (supplied by JMS) does use these indexes – and if it does, it must be the only application ever to have done so.

V – Level V: Defining the Object List

A - The object list

As we saw above, it behoves us to create an object list, which contains all of the objects of the menu application subwindow. This list is set up with the ***MK_AOL*** (***M***a***K***e ***A***pplication subwindow ***O***bject ***L***ist) function.
`objlist = MK_AOL(olattr, oljus%, olkey$, oltype%,olstr$,olspr,olblb, olpat)`

These parameters have the same meaning as for the ***MK_LIL*** function (see my earlier instalments in this series). However, there is no parameter defining the window or the sizes (we have already seen above how the sizes and spacings are defined). Nor do we define the origin of the object, which seems quite natural as the object is part of a regular and organised menu. Moreover, the object doesn't necessarily stay at a fixed position in the window, as it is possible to split an application subwindow into sections, and join them together later on. In addition, the menu may be scrolled or panned, and thus the object does not stay in a fixed position with respect to the window. However, we must define the attributes (same attributes for all objects) and then the justification, selection key, type and content for each object – these parameters should all be pretty clear by now. In the "files" subwindow of QPAC2, the type is of course a string and the content of the object is the name of the file. Actually, the type will generally be a string, but not necessarily so, as Jérôme Grimberts examples in these pages have shown!

B - "Blobs" and "patterns"

A **blob** is a structure that defines the shape of a visual object. This is similar to tracing a character on the screen: with a character editor, one can define what pixel must be "on" and what pixel must be "off". However, the character is only visible when it is printed on the screen with any INK on any PAPER (or, rather, STRIP). This is similar for blobs, except that you are not limited to the size of one character.

However, a blob has no colour, it just states that this pixels is on but not that another pixel is not on. It does not say what colour the pixel is to be, that will be defined by the colour pattern.

Thus, without a pattern, the pixel would be invisible, because it would be transparent, having no colour. The blob is like a mask which lets colour shine through or not.

A pattern is just the contrary – it is the definition of a structure with colours, but without a particular shape. By combining a blob (a shape without colour) and a pattern (a colour without a shape) we obtain something that is visible on the screen. A pattern without a blob can't be seen because it has no shape. Only the combination of the two produces something visible. A sprite is an example of a blob combined with a pattern, as it defines, at the same time, a shape and the colour of each pixel within that shape.

Let's re-use the example of the arrow which we had used for the sprite. It was something like this:

```
 90 DATA '   a   '
100 DATA '  awa  '
120 DATA ' awwwa '
130 DATA 'awawawa'
140 DATA '  awa  '
150 DATA '  awa  '
160 DATA '  awa  '
170 DATA '  awa  '
180 DATA '  aaa  '
```

This arrow can also be used as a blob because it defines a shape. It is just that for the blob, it makes no difference whether the colour in it is "a" or "w" or anything else. The only thing that counts is whether the pixel is transparent (' ' = off) or not (any colour definition – "w", "a", "r", "g" etc means that the pixel will be on). The colour itself is then filled in with the pattern. When the above data is used as a sprite, the pattern is made up from the colour information contained in the arrow data. But if the above is used as a blob, the colour information just tells us whether a pixel is on or off.

Now we shall apply a pattern to this blob:

```
DATA 'rrrarrr'
```

This mean that the arrow will be red, except for the pixel in the middle, which will be black. This pattern is applied to each row of the blob in turn and it is the combination of both that produces a visible object on the screen.

But why do it in such a complicated way when, as when have seen for sprites, everything could conveniently be made up in a single block? Well, that's just why: if everything is in a single block, you have to redefine everything if you want to change just one colour. If, for a sprite, I want everything to be red instead of black, I'd have the redefine the entire sprite. With a blob and a pattern, I just design the blob and several patterns and thus I can change colours as I want to, by using different patterns with the blob...

As a pattern may be defined in a single line, this is pretty fast! But a pattern may also be much more complicated and there may be one pattern line per line in a blob. This is what happens for sprites. In the above example with the arrow, QPTR makes up a blob and a pattern from the information contained in the data: a blob makes up the shape of the object, and there is a pattern with as many lines as there are lines in the blob.

For the basic programmer using QPTR, pattern and sprites are defined exactly like sprites – you should just make sure that the sprite origin is 0,0 because, of course, blobs and patterns don't have origin (and if you don't understand why not, even though a sprite has one, I'd recommend re-reading the section on sprites!)

You will be happy to know that this concludes the first big section of this series. By now, we have seen all there is about defining windows. In the next instalment, we'll be able to start on actually making the window appear on the screen.

Just a word of advice. I Have tried to cut up the window information into different levels, starting at the top level, and then working down. When you set up your window, you would, of course, do it the other way round: first you build the lower levels and then you work your way up, since you often need the lower level pointers and parameters for the higher level ones.

```
 9      window resize        "
10      sleep
11      wake
12      f1
13      f2
14      f3
15      f4
16      f5
17      f6
18      f7
19      f8
20      f9
21      f10
22      f11
23      f12
24      cf1
25      cf2
26      cf3
27      cf4
28      cf5
29      cf6
30      cf7
31      cf8
32      cf9
33      cf10
34      cf11
35      cf12
36      cursor
37      winking cursor
```

NOTES

1. Sprites 8 to 37 are new system sprites.
2. sprites 6 and 7 are "mouse pointers" and sprites 8 and 9 are "window sprites".

Concluding Puzzle

1. A colour value in a working definition is $0220.
2. The 33rd entry in the system palette linked to the program is $0220. 3. What happens?

A PS - the Twice MT_RECHP Bug

The 3.xx versions of SMSQ/E do not adhere to the original QL memory layout. Free memory used to be found between SV_FREE and SV_BASIC. In the new versions of SMSQ/E the space between SV_FREE and SV_BASIC is limited to about 840K, the real free memory being elsewhere.

However, in v3.01, if MT_RECHP is called twice with the same address, the memory seems to revert to the old SV_FREE to SV_BASIC area. A large amount of memory will then seem to have disappeared!

Programming QPTR in SBasic - next part

Wolfgang Lenerz

Obviously not.
(And if you're wondering what this means, look at the end of the last instalment!)

II – Altering Windows

Several commands exist to change or alter either a primary or secondary window entirely or only in part (i.e. a sub-window or item).

A - Removing the window

First of all, a command to take the window away entirely, which surely must be the most drastic alteration...

DR_UNST `workdef`

where workdef is the working definition of the window, as obtained by *MK_WDEF*. The command will remove the window entirely, including all of the subwindows (but not the secondary windows, which should, however, have been removed before) and will also remove the window from the screen. If the window was opened via the *DR_PULD* call (i.e. it is a secondary window), then the implicit and inaccessible screen channel open by that command is also closed automatically – actually, this is the only legitimate way of closing this channel (unless it is done by some "external" operation, such as QPAC2's "channels" menu). If the window was opened with *DR_PPOS*, then the corresponding channel is NOT closed, and should be closed later on if need be. If you try to remove the primary window when secondary windows are still open, bizarre things will happen, so try not to do that – always close secondary windows first and the primary window last.

B - Changing the window

The size, position, content and certain attributes of windows (and sometimes sub-windows) may be changed.

1 – Changing the size or position of a window

With the *CH_WIN* ("**CH**ange **WIN**dow") command you can change the size or position of the window. This command can only be used with secondary or primary windows but not with any sub-window and is used as follows:

CH_WIN workdef `[,xsize%,ysize%]`

When you use this command without the two optional parameters, the window will change position, i.e. move about the screen. Under QDOS, the pointer changes to the "move window" sprite, you move it around and hit Space/Enter to signify to where you want the window to be moved.. Under more recent versions of SMSQ/E, it is also possible to move the window itself, or its outline,

around the screen. The movement of the pointer sprite/window content/outline is automatically handled by the Pointer Environment, the programmer doesn't have to do anything in particular.
Using this command with parameters will result in a change size operation,. The parameters are:

–› * workdef is the working definition

–› * xsize% and ysize% are optional return parameters. As mentioned above, when omitted they signify that the window should only be moved and the programmer doesn't have to concern himself with this (other than calling the command), all is handled by the Pointer Environment. However, a few things should be considered when using this command, even in "move" mode.

If you move a (primary or secondary) window, all sub-windows are automatically moved with it. Since sub-windows are defined relative to the main window, this is as should be.

However, if you move a primary window, the seconday windows are not moved at the same time. And this can result in quite some unforeseen consequences. Hence, never allow the user to move a primary window when secondary windows are still open. Look, for example, at the QPAC2 "Files" menu – when the F3 commands menu is opened (this is a secondary window), you cannot use the items in the primary window, and thus cannot move the window about the screen. To do that, you first must close the seconday window.

Moreover, if you have opened a channel over a subwindow or an item (more about which later in this series), the channels ARE NOT moved with the window – thus, after each move operation, you should re-open them again over the sub-window or item.

When xsize% and ysize% are not omitted, this means a change size operation. The pointer will change to the usual change size sprite which you can move about the screen to click and signify how much you want the window to change size. At the click, command will pass back to the program. Remember that xsize% and ysize% are RETURN parameters. These variables then contain, upon return from this command, the displacement (+ or –) of the pointer, in pixels, from the moment the command was invoked until the user's click. For example, if the pointer was at (100,100) at the time the command was invoked and if the pointer is then brought to (210,100) and then the user clicks, xsize% will contain 110 and ysize% contains 0. If the pointer was brought to 50,110, xsize% will contain –50 and ysize% 10. And so on.

It is then the programmer's responsibility to re-draw the window entirely, taking into account the changed size as expressed by the user. It is not obvious how to achieve this – in fact, the best way is to remove the window entirely, make a new working definition and put the new window up on the screen. In my opinion, this is one of the most feeble aspects of the Pointer Environment, other operating systems (even Windoze) do it better than that, sometimes even clipping the window automatically.

2) – Changing the pointer

At some time, it might be interesting to change the pointer of a primary or secondary window. One can even change the pointer for an application sub-window (but not for any other sub-window). This change is achieved with the *CH_PTR* (***CH***ange ***P***oin***T***e***R***) command:

CH_PTR `workdef,win_nbr%,new_ptr`

–› * *workdef* is the working definition of the window.

–› * *win_nbr%* shows the number of the window or application sub-window to be changed: 0 for the first application subwindow, 1 for the second etc... If you want to change the pointer for the entire window and not only an application sub-window, use –1.

–› * *new_ptr* is the address of the new pointer to be used, as returned by ***SPRSP***. If this parameter is 0, then the default pointer is used. For a primary or secondary window, the default pointer is a small arrow. For an application sub-window, the default pointer is the pointer used by the primary (or secondary) window enclosing it.

3 – Changing the content of a sub-window, object or item

The following command allows us to change an object in a subwindow, whether it is an information subwindow or an application subwindow. One can also change the content of a loose menu item with this command: *CH_ITEM* (***CH***ange ***ITEM***).

CH_ITEM workdef, win_nbr%, obj_nbr%, type%, key$, value

–› * *workdef* is the working definition, as usual.

–› * *win_nbr%* is the number of the sub-window to be changed. Here, the following rules must be observed:

  –› If win_nbr% is –1, it signifies a change in the main window, i.e. a change in a loose menu item only.

  –› If it is a negative value n other than –1, it means an information sub-window, calculated as follows: ABS (n) –2. Thus –2 means information subwindow 2–2 = 0. –3 means information subwindow 3–2=1, and so on...

  –› If it is a non negative value n, it means the application sub-window n+1: 0 is the first application sub-window, 1 the second etc...

–› * *obj_nbr%* contains the number of the object (or loose item) to be changed. The list starts at 0, as usual.

–› * *type%* is the NEW type of the object (text, sprite, blob or pattern, using the usual values).

–› * *key$* contains the NEW selection key for the item or object (obviously, this is not used for objects in an information sub-window which have no selection key). Use an empty string ("") if you want to keep the old selection key, or a nul value string (CHR$(0)) if you do not want the object to have a selection key.

–› * *value* contains the new value. The type of that depends on the type of the new object (as indicated by type%) – this will be a string for text items, or a pointer to a sprite, blob or pattern for those objects that need one.

C - Redrawing part of a window

Once the content of an item, object or sub-window was changed, that (sub-) window containing it must be re-drawn. For loose menu items or menu application subwindow objects this can be done automatically, without using any special command, but there are also commands to do it explicitly.

The implicit way (which does not exist for information sub-windows and their objects and thus only exists for loose menu items or the objects of an application sub-window) is to set the "flag" of that item to a certain value, which shows that one wishes this object to be redrawn.

Indeed, we saw earlier that the *DR_PPOS* and *DR_PULD* commands use "flag" arrays for the loose menu items and for the objects of menu application sub-windows. I even explained how these flag arrays are used to set and show the status of the items when they are drawn initially. These two types of flag arrays are also used by the *RD_PTR* command, which is the main way of reading the pointer, and which was explained in an earlier instalment of this series.

If, before using this command, the value of an element of the array is set to the value of the status wished plus one, then the corresponding loose menu item or menu application sub-window object is automatically redrawn when the RD_PTR command is next called. As we saw earlier, a value 0 in an array element means that the item is available, 16 means it is unavailable and 128 means the item is selected.

Thus, if I want an item that was unavailable to become available, I just have to place 0 + 1 in the correspoding flag array element. The item will then be redrawn with the new status at the next call upon *RD_PTR* And, if the content of that item had changed in between (using *CH_ITEM*), it will be redraw with the new content. You don't even need to change status: an available item (value 0) will be redrawn as available if the value is set to 1.

Now, let's look at the explicit redraw commands:

1) Loose menu items

The command *DR_LDRW* (**DR**aw: **L**oose items **DR**a**W**) is used to redraw one, several or all loose items. It takes the following parameters:

DR_LDRW workdef,lilflag%

–› * *workdef* is, as usual, the working definitions of the window concerned (which contains the loose menu).

–› * *lilflag%* is the same integer status array as for *DR_PULD.*

Of course, before using this command, you should place suitable values into the array, corresponding to the status of the items wished. Then you add 1 to the items statuses – only the items that have this change flag set will be redrawn – with one exception, however:

If **NO** element of the status array has the change flag set, then **ALL** of the items are redrawn. The logic of this is hard to fault – after all, you are only going to invoke this command when **SOMETHING** at least has changed – if nothing is then pointed out via the change flag, then all of them must be redrawn.

Most of the effects of this command can easily be obtained by just setting the change flag in the status array (adding 1 to each status) and calling *RD_PTR*

2) Application sub-windows
To redraw an application sub-window, use the command *DR_ADRW* (**DR**aw **A**pplication sub-window re-**DR**a**W**), as follows:

DR_ADRW `workdef, win_nbr%, appflag% [,ctrldefx%, ctrldefy%]`
Here, all parameters are the same as for the DR_PULD/DR_PPOS commands (except for the win_nbr% parameter): working definition, flag array and the control definition arrays. The win_nbr% parameter contains the number of the application sub-window concerned (starting at 0 as usual).
This is a more practical command than that concerning the loose menu items, because you can also change the control definitions. In that case, you should not forget to set element (0,1) of the changed control defintion to 1, to signal that it has, indeed, changed.

3 - Information sub-windows
Nothing can change status in information sub-windows - there are no items. But an information subwindow can be redrawn entirely and thus a changed content be put on the screen. This is done with the command *DR_IDRW* (**DR**aw **I**nformation sub-window re-**DR**a**W**).

DR_IDRW `workdef,info_nbr`
-› * workdef is of course the working definition.
-› * info_nbr is a bitmap which indicates the window to be redrawn: for each information sub-window, there is one bit. If this bit is 0, then the information sub-window must be redrawn, else it will not be redrawn. Info_nbr is a long word (32 bits) and this command can thus "only" be used for the 32 first information sub-windows (that SHOULD be enough!). Bit 0 is for the first information sub-window, bit 1 for the second and so on. Thus, if info_nbr = HEX$('FFFFFFE') this means that information sub-window nbr 0 should be redrawn.

III - How to set a Channel over a Sub-Window

The main problem with sub-windows is ... that they don't exist! At least not for the normal programmer. As was already mentioned, these windows are not windows in the normal QL sense of the word. They have no channel attached to them, they are internal Pointer Environment subdivisions (not even inaccessible channel as the one opened by DR_PULD for secondary windows).
Actually, this makes sense. A typical Pointer Environment window has many loose menu items, several information sub-windows and often one or several application sub-windows. It would not be reasonable to give each of them its own channel and channel ID - not only would we risk running out of place in the channel table, but also, each channel takes its own slice of memory. So, there are no channels associated with the sub-windows. However, sometimes it is necessary to have a channel that "covers" a sub-window or an item. This is useful, for example, when one is supposed to type something "into" a loose menu item.
The solution consists in opening a normal "CON" channel and setting it over the item or sub-window. Once the operation is finished, the channel can be closed again, if need be.
There are three commands to place channels over each of the two types of sub-windows (information sub-windows and application sub-windows) as well as loose menu items. I have already pointed out that, when the window is moved, these channels do not move with it, and thus, after each change in the window's position (or, indeed size), you should re-set the channels over the sub-window or item concerned.
Of course, the channel to be set over the sub-window or item should be a "CON" channel, opened beforehand.

A - Setting a channel over an application sub-window

This is done with the *DR_AWDF* command:

DR_AWDF `#channel, workdef, app_wdw%`
sets a channel over the application sub-window the number of which is given by app_wdw%. As usual, the count starts from 0. You will, by now, have guessed that workdef is the working definition of the window enclosing the application sub-window and "#channel" is the channel to be used.

B - Setting a channel over an information sub-window

DR_IDF `#channel, workdef, info_wdw%`
sets a channel over the information sub-window the number of which is given by info_wdw%. As usual, the count starts from 0. You will, by now, have guessed that workdef is the working defini-

tion of the window enclosing the application sub-window and "#channel" is the channel to be used.

C - Setting a channel over a loose menu item

DR_NWDF #channel, workdef, item%
sets a channel over the loose menu item the number of which is given by app_wdw%. As usual, the count starts from 0.
It is up to you whether you open and close the channel after each operation, or whether you keep open a genral putpose "con" channel which you set to the sub-window/item each time it is necessary.

OK, that's it for now. More next time.

How to read QL disks on a PC

Jimmy Montesinos

Before beginning

Disks that have been formatted on a QL cannot be read directly on a PC without some special software, such as a QL emulator. Also, normally disk interfaces on the QL will only format DSDD disks (1440 sectors = 720 Kbytes).
If you use the more common form of HSDD disk of (2880 sectors = 1.44 Mb) for your PC, you can put sticky tape across the hole on the left of the disk (not the hole which is used to make the disk read-only). If you do this, the computer will think that the disk is only a DSDD disk.

Preparation of a disk

With the use of a small utility, you can format a disk on the QL, store data on it and later read that data on the PC.

This utility is: QLTOOLS 2.7 and was written by: Giuseppe Zanetti, Valenti Omar, Richard Zidlicky and Jonathan Hudson.
It is possible to download it from:
ftp://ftp.nvg.unit.no/pub/sinclair/mirrors/ql/demon/

Qltools27.nt.zip is for use under Windows 2000 or Windows XP. You might also want to read the following web-page of Richard Zidlicky:
http://www.geocities.com/SiliconValley/Bay/2602/ql.html

After decompressing the file `qltools.exe` onto your PC's hard disk, place an empty disk in the PC's disk drive and from the RUN command in the Start menu, type:
`Qltools \\.\a: -fdd QLFloppy`
(\\.\a: is the description of the top disk drive in a PC that uses Windows NT/2000/XP - there is no space between the full stop and the backslash.)

Later it is possible to format the disk from the QL. Place the disk in the disk drive of the QL and enter the command:
`FORMAT FLP1_QLFloppy`
There is a delay and the QL screen shows:
1440/1440 sectors

This part is needed only if your original QL floppy cannot be read directly by QLTOOLS, which should not happen with most disk interface like Sandy QBoard, GoldCard etc.

To copy files from the QL to this disk

Now is the time to transfer the original files of the QL onto this new disk:
If you have the TK2, you can for example use:
`WCOPY MDV1_ TO FLP1_`
After responding `A` ("ALL" Files) all the files on MDV1_ will be copied to FLP1_ (the floppy disk).

In order to copy all the files of a QL disk to this new disk the best thing is to use the Ramdisk. This can be done with the following instructions:
`FORMAT RAM1_1440`
Then insert the original disk and enter:
`WCOPY FLP1 _ TO RAM1 _`
and answer `A` (All files).
Now insert the disk prepared on the PC and enter
`WCOPY RAM1 _ TO FLP1 _`

To read and use the files on a PC under QPC2

The users who have the best QL Emulator in the world (QPC 2) can directly read the files of this disk using the same instructions as on the QL, such as:

```
DIR FLP1 _
LRUN FLP1_boot
COPY FLP1 _ TO WIN1 _
etc...
```

You do not need to prepare a special disk for this and can use the original QL formatted disk. QPC2 will even allow you to read from and save to a PC formatted floppy disk directly (the standard QL can read these disks with a variety of tools). If you copy an executable file to a PC formatted floppy disk, you have to remember two things:

Programming QPTR in BASIC - third part

Wolfgang Lenerz

Reading th Pointer

Reading the pointer will enable you to get the user's response to the different possible menu actions. This, of course is a paramount part of programming in the Pointer Environment. You have the choice between two different methods of reading the pointer, with three different keywords (two of which are very similar). The first method is the most interesting, even though the second, a directy pointer read, can also be useful.

I - Reading the pointer indirectly

This method makes our programming much easier. It is structured around the ***RD_PTR*** (***R***ea***D*** ***P***oin***T***e***R***) and the ***RD_PTRT*** (***R***ea***D*** ***P***oin***T***e***R*** with ***T***imeout) keywords. When one of these commands is used, the pointer is drawn on the screen (in the shape determined by the working definition). The user can move the pointer via the mouse or the keyboard cursor keys. The commands will only come back to the program when:

1) The user did (and in some cases hit) an item in a menu or an application subwindow (or used the respective selection key) and
2) this action did happen in this window - nothing will happen if the user clicked outside of the window.
3) With the RD_PTRT keyword, a return can also be made when a timeout or "job event" occurs.

The advantage of this command seems obvious: it handles all of the changes in the pointer shape and state, notably if you have specified different pointers for application subwindows: the pointer will automatically change when it is brought over such an application subwindow. Likewise, when the pointer is moved outside of the primary window, it may change shape and become that of another window, or the default sprite (an arrow) or a sprite showing that the window underneath is not a managed window or expects keyboard input etc.

A click outside of the window is not acted upon, and, in fact the command only comes back in case of a timeout or job event (for RD_PTRT) and when the user somehow actioned something inside of the window.

That's very practical for the programmer. When the command returns to the program, return parameters indicate what happended. Thus, there are a LOT of parameters for this command, but they are all pretty logical. We'll start with the RD_PTR command:

1 - RD_PTR

```
RD_PTR workdef, item%, subwin%, event%,
xrel%, yrel%, liflags% {[,appflags%
[,ctrldefx% ,ctrldefy%]]}
```

Quite a mouthful!

The parameters are the same for both RD_PTRT and RD_PTR, and they are as follows:

-› * ***workdef*** is the window working definition. The window can be a secondary or a primary window, according to how workdef is set up. Unfortunately, when there is a primary window and a second window it is not possible to choose in which one of these you want to read the pointer: indeed, if you open a secondary window "over" a primary one (e.g. the "commands menu in the QPAC 2 FILES program) the secondary window locks the primary window over which it is pulled down and which it covers totally or partially. The primary window thus no longer is the window on top and can't read the pointer anymore.

-› * ***item%*** is a returns parameter. It contains the number of the item the user hit or did, and which caused the command to return. The return mechanism is as follows: you may remember that one of the parameters for a the definition of a loose menu item or a menu application sub-window is its type (text, sprite etc) to which one adds 256 or -256: this type will then determine how the item reacts when hit/done:

- If nothing is added to the item type , then this item acts as follows when actioned: if the item is ***hit***, it just changes state - if selected it becomes available and if available it becomes selected, but it DOES NOT cause the RD_PTR command to return. If the item is ***done***, it changes state (to show that it was selected and hit) and then causes the command to return.
- If -256 is added to the items type, both actions (hit or do) will produced the same

result, i.e. a change of state towards selected (or available if the item was already selected) and a return form the RP_PTR command loop.

- If 256 is added to the item type , both a hit and a do will, again, have the same result: the item will cause the RD_PTR(T) command to return to basic, but the item state will automatically be reset to available, without any programmer intervention.

These last two cases (256 and -256) thus cause an "automatic return" from the item when it was hit or done.

-› * ***subwin%*** is also a return parameter. It contains the number of the (application) subwindow in which the pointer was located at the time of the user action. If the pointer was not in an application subwindow but on a loose menu item or anywhere else in the window (an information sub-window, for example) then this parameter will be -1. With this, you can determine and find out whether the user clicked a loose menu item or an item in an application sub-window.

-› * ***event%***, again a return parameter, contains the "event" that caused this return. This "event" may be either the fact of actioning an item/object, or the press of any of the following keys: ESC, F1, CTRL F1, CTRL F2, CTRL F3 or CTRL F4. To each of these keypresses corresponds a certain event, and each event has a code which is thus returned in the event% return parameter. These codes are:

1 = DO : an item was done (ENTER)
2 = CANCEL: ESC was pressed
4 = HELP : F1 was pressed
8 = MOVE : CTRL F4 was pressed (move window)
16= SIZE : CTRL F3 was pressed (change window size)
32= SLEEP : CTRL F1 was pressed (make into button)
64= WAKE : CTRL F2 was pressed (WAKE)
128= HIT on an item with an automatic return.

Thus, the above keystokes will also cause a return from the RD_PTR(T) pointer read loop.

-› * ***xrel%*** and ***yrel%*** are the pointer coordinates at the time when the event caused the return to the program. These coordinates are relative to the upper left corner of the window (of the application subwindow) in which the pointer was when the return occurred.

-› * ***liflags%*** is the same flag array for loose items as that used for the ***DR_PULD*** and ***DR_PPOS*** keywords (see in an earlier instalment of this series). Remember, these flags may have a value of 0, 16 and 128. If you add one to these values when calling the ***RD_PTR*** command, then the item will automatically be redrawn in the appropriate status.

-› * appflags% is the same flag array for application subwindows that is used in the DR_PULD and DR_PPOS commands which we treated in an earlier instalment of this series. Just like for loose menu itmes, if you set any value of these flags arrays to the status +1, the items will be redrawn automatically upon entering this command.

-› * ***ctrldefx%*** and ***ctrldefy%*** are the application sub-window control definition arrays.

Using this command is pretty easy because it only causes a return for well defined events. It can get included in a pointer read loop which will be about as follows:

```
REPeat loop
 RD_PTR ‹parameters›
 post=item%:REMark SELect on floats only in QDOS
 SELect ON subwin
  =-1       : rem loose menu item
    SELect ON post
      =1:do_this
      =2:do_that
      =3:something_else
      ... etc...
    END SELECT
  =0        : rem click in first menu appsub wdw
    SELect ON post
      ....
    END SELect
  END SELect
END REPEAT loop
```

Thus, one reads the pointer and when the return was made, one uses the subwindow and the item to determine, first, in what subwindow the event occurred and, second, what action should be taken for this event.

2 - RD_PTRT, timeouts and job events

The ***RD_PTRT*** keyword is pretty similar to the ***RD_PTR*** keyword. Both use mostly the same parameters, except that the RD_PTRT keyword has one additinal parameter, a timeout, as follows:

```
RD_PTRT workdef, item%, subwin%, event%,
timeout%, xrel%, yrel%, liflags%
{[,appflags% [,ctrldefx% ,ctrldefy%]]}
```

The wordkdef, item%, xrel%, yrel%, liflags%, appflags%, ctrldefx% and ctrldefy% parameters are the same as for ***RD_PTR*** and thus don't need to be described here anymore.

There are two changes with respect to the ***RD_PTR*** keyword:

First of all, there is an additional ***timeout%*** parameter. With this you can indicate that you also want a return from the keyword after a certain time. In usual QL fashion, the timeout is given in 1/50th of a second (me thinks, in North America it is in 1/60th of a second).

Thus, you can also make sure that you return from the read pointer loop after a certain period of inactivity. Mind, though, that the return will be made either because of a "normal" event (including "job events, which I'll explain below) or because of a timeout - whatever comes first!

When a return from a timeout occurs, the event parameter is set to -1, which is a value it doen't normally have. This allows you to distinguish between normal events and a timeout.

Speaking of the event% parameter, this has been modified a bit. It still has all of the functions as for the RD_PTR keyword, but has been extended. You can also use it as an entry paramter for the ***RD_PTRT*** command, to pass it some "job events" on which the keyword will also return.

Job events are a relatively recent addition to the Pointer Environment. They are an easy, legal and (now) documented way for one job (program) to communicate with another. One program can send another an "event". The other program receives the event through the read pointer loop. There are 8 events, contained in one byte, each bit representing one event.

Sending an event is pretty easy, and uses the typical TK II fashion of determining a job:

```
SEND_EVENT job_id,event
```
(the job_id is a composite number: job_tag * 65536 + job_nbr),
or
```
SEND_EVENT "job_name", event
```
or
```
SEND_EVENT jb_nbr,job_tag,event
```

For example:
```
SEND_EVENT "Quill",3
```
will send events 1 and 2 to Quill. This wouldn't mean a lot, since Quill isn't equipped to receive events, but it could be done.

When the program is a pointer program, it will receive the event throught the pointer read loop. In S*Basic, this event may cause a return from the pointer read loop: "May" not "will" - at least not necessarily.

Indeed, the event% parameter will indicate, on entry to the command, what job events the program is ready to receive. If the event% parameter is 0, it is not ready to receive any event. If event% = 1 * 256, it is ready to receive event 1, if it is 3 * 256, the program is ready to receive events 1 & 2 and so on.

As you can see, the event is passed in the high byte of the event% word, thus just multiply the events to indicate by 256. There is one problem: if you want to indicate that you are ready to receive all 8 events, you would normally have to pass 255*256. This will cause an error, so use -1 instead.

On return from the read pointer loop, the job events are contained in the upper byte of the event% word.

There is also a way to get an event without reading the pointer:
```
result%=WAIT_EVENT (events% [,timeout%])
```

This will wait for timour% ticks (if this parameter is not passed, it waits forever) until on of the evnts% passed on entry happens. The event(s) are returned in result%

This was the easy way to read the pointer. Next time, we'll look at a more circumvallated way of doing this.

QL Forever!

The menu colours are almost entirely defined as $2xx system colours, then you can choose palette 0 to 3 and load your favourite theme there (using QCoCo or the Colour Utilities Disk). You can change colours while the program is running. Also in mode 4 you can now use the familiar palettes known from QMenu and Qpac2. There is a small SBasic program to set a system palette to the old Suqcess colours (and make all the other applications look like a Suqcess ;-)

One remaining snag is that the scroll/pan arrows colour can not be set. For that I made a small procedure to set these colours just before the application window contents are drawn, using;

```
MAWSETUP #ch\subw,...
```

set colours in the Working Definition:

```
MAWDRAW #ch,subw
```

Only the application window border colour remains stubborn and is set to a mode 4 "gray" stipple, which should be ok for most colour schemes.

The whole EasyPTR package needs a big overhaul to bring it to GD2 standard but with the help from some experts we managed to find a workaround for most problems, except for the "gray" stipple. Something to do for the next update.

Compared to the previous Suqcess version 1.19 there is not much functionality added to version 2. You can now open a database "Read Only", some bugs were fixed, improvements made in View, in Direct Sort and to a few prompts. Every Loose Item now has a key attached to it. Further changes were mostly in the "looks" department. Suqcess2 only runs under SMSQ/E 3 because of the new colour commands. A trial version can be downloaded from Wolfgang's site:

www.uhlich.nl/ql/

Full versions can be obtained from Jochen Merz Software. They come in English, German and Dutch flavours.

Programming QPTR in S*Basic (Part 10)

(It seems we somehow lost track of the part numbering in previous issues, but 10 should be OK now).

Wolfgang Lenerz

We follow on from the last instalment by examining, this time, a more convoluted way, even if it is a "direct pointer read"

II - READING THE POINTER DIRECTLY

With this command the pointer can be read at any time and the return from the command can be either immediately, or at the occurrence of a certain event, as specified by the programmer. Contrary to the RD_PTR command that we saw last time, there is only one command, RPTR (no "RPTRT"). However, RPTR also takes into account job events. This command (Read PoinTeR) takes the following parameters:

RPTR xabs%, yabs%, end%, winnum%, yrel%, yrel%, return$

-› * ***end%*** is a variable that determines under what conditions this command returns to the programmer. The conditions are determined by setting individual bits in this variable to 1, according to what one wishes. The following table contains the return conditions, if the corresponding bit is set to 1:

Bit set to 1	return if:
0	a keyboard key, or a mouse button is pressed
1	keyb. key or mouse button continues to be pressed
2	the key or button is released
3	the pointer moves away from the given coordinates
4	the pointer is, or moves out of, the window
5	the pointer is in, or moves into, the window
6	NEVER set to 1!!!!
7	"special" mode

Most return conditions may be mixed together at your heart's content: if you set both bits 4 and 5 to 1, then the command will return immediately because the pointer is always either in or ouside of the window!

You may set any individual bit in this variable to 1 by first setting the entire variable to 0 and then adding 2^x to this variable, where x is the position of the bit in the variable. If I add 2^4 (=16) I set bit 4 to 1. So, by adding 48 (=16 + 32 = 2^4 +2^5) I set bits 4 and 5. Of course, you must add this only ONCE for each bit.

The "special mode" which is chosen when bit 7 is set, will lock all windows of all other jobs and show a special sprite, which can be:

- the change size sprite, if bit 1 is also set to 1
- the "move window" sprite if bits 1 and 0 are both 1
- the "empty window" sprite if both bits 1 and 0 are set to 0

When bit 7 is set to 1, all other bits (except 0 and 1) should be set to 1.

This parameter is also used to set the job events on which one wishes the program to return. We discussed the job events last time, please refer to the last instalment of this series.

The job events are included in the high byte of the end% word. To set any of these events, proceed as above (2^x where X is the event number, from 0 to 7) but then multiply that value by 256. (Note: from S*Bazic, you can only set the first 7 events (0-6) and not event nbr 7, as that would be exceed the value of an integer in S*Basic. You would need to use a negative number for that). So, to set job event n° 2, I'd add (2^2)*256 to end end% variable.

-› * ***winnum%*** contains the number for the main window (=-1) or the number of the application subwindow to which the pointer read should apply (especially to know whether the pointer is in the (sub-)window or not).

-› * ***xrel%*** and ***yrel%***, which are return parameters, contain, on return, the pointer coordinates in the window or in the application sub-window in which the pointer was when the command returned.
They are both relative to the origin (upper left hand corner) of this window or application sub-window.

-› * ***xabs%*** and ***yabs%*** are used when bit 3 of end% is set to 1. They then contain the ABSOLUTE pointer position - when the pointer moves from this position, the command will then return.

These parameters also contain, on return, the absolute position of the pointer (in all circumstances). Again, this is relative to the screen origin (upper left hand corner).

-› * ***return$***, another return parameter, contains the character code (chr$) of the key pressed, or one of the following values, with the following meaning:

Key	content of return$ CHR$	Meaning
none	0	no key pressed
SPACE/left button	1	Hit
ENTER/right button	2	Do
ESC	3	cancel
F1	4	Help
CTRL F4	5	Move window
CTRL F3	6	Change window size
CTRL F1	7	Sleep
CTRL F2	8	Wake

Thus, with this command, you can also read the pointer. Its disadvantage is that it doesn't take into account any loose items etc... It is thus more difficult to use than the RD_PTR command and doesn't use all of the facilities offered by the Pointer Environment.

That's it for today. If you've been following this series continuousy, you should now have a firm grasp of the concepts used by the Pointer Environment, and also how to use them from S*Basic.

Next time, we'll look at some additional keywords, which will probably conclude this series.

Programming QPTR in SBASIC - Last part

W. Lenerz

Additional Commands

The QPTR extensions contain some additional S*Basic keywords, as follows:

I - Commands for the mouse and the hotkey system

Several keywords are concerned with the mouse and access to the hotkey system.

A - Accessing Hotkey System II

The hotkey system is closely linked to the Pointer Environment and two commands give you some access to it.

1) Filling the Hotkey buffer

The hotkey buffer (also called "stuffer buffer") is a small buffer that you can fill with strings which you can then get at by hitting the hotkeys ALT + SPACE (or ALT + SHIFT + SPACE) together. This, however, is only possible once the Hotkey job is running, which is achieved via the HOT_GO command of Hotkey System II (if you don't have the HOT_GO command, then you are still using Hotkey System I – an immediate upgrade is really necessary).

As soon as the hotkey is hit, the content of the stuffer buffer will be stuffed (hence the name) into the current keyboard queue (just as if you had used the old TK II Altkey system – please note that Hotkey System II will get rid of the Altkey used by TK II, else too many routines would compete for access to the Altkeys). The effect is that the string appears as if you had input it via the keyboard.

The stuffer buffer can also be filled by other programs: thus QPAC2's FILES menu puts the names of files selected into the stuffer buffer. So does QD with the names of the files saved/loaded. FiFi can also do this, and so can others (I would really like this to be a configurable feature of every program, though). Recent versions of SMSQ/E will also put a string currently being edited with the INPUT command, or by programs using the "edit line" trap, into the stuffer buffer whenever F10 is hit during editing.

With the HOT_STUFF command, you can explicitly put a string into the stuffer buffer. The syntax of this command is:

```
HOT_STUFF a$
```

a$ is the string to be put into the buffer. You can put several strings in there by passing them as parameters separated by commas:
`HOT_STUFF a$,b$,c$,d$....`
the string a$ will be put into the buffer first.

2) Picking a job

You now know that jobs (or their windows) are organized in a stack. The job the window of which is on top of the stack will have its window unlocked. With the PICK function, you can bring a job to the top, where its window will be visible and unlocked. This is like a repeated CTRL +C, but more targeted to a specific job. Instead of just cycling through all jobs as does CTRL + C, you can PICK any specific job you want.

The syntax of this function is:

```
result = PICK ([#channel,] JobID )   or:
result = PICK ([#channel,] key)
```

As usual, if you do not specify a channel number, channel #1 will be taken as default.

The job ID can be specified as "job number, job tag", which is what is returned by the TK II JOBS command. You may also use a single number:
`job_tag *65536 + job_number`

The "key" may be -1 or -2. If you use a key of -1, then the job at the bottommost place will be picked to the top. If you give -2 as key, then the same thing happens, but the window of that job will be marked as unlockable: its output will always be visible as soon as it changes.

B - Mouse commands

1 - Filling the mouse buffer

In a similar way that we have a Hotkey System II stuffer buffer, there is also a mouse buffer – but this is severely more limited. Indeed, the buffer

holds only two characters at the most. It can be filled with the MS_HOT command.

The content of the mouse buffer may be retrieved by clicking both mouse buttons at the same time – this buffer thus is only for those that do have a mouse...

The syntax of this command is:

```
MS_HOT [#channel,],a$
```

where a$ is a string of two characters at the most.

As usual, the channel number will default to #1 if you do not specify it.

If you pass an empty string then clicking both mouse buttons at the same time will no longer have any effect at all.

The interesting thing about the mouse buffer (and this is contrary to the stuffer buffer) is that the mouse buffer is polled before the Hotkey/ Altkey routines poll the keyboard. Practically, this means that you may use the mouse buffer character to set off a hotkey – when you click both mouse buttons, this behaves as if you had hit the corresponding hotkey. To achieve this, though, you must fill the mouse buffer with two characters, the first must correspond to the ALT key (i.e. CHR$(255)) and the second to the Hotkey you wish to activate.

2) Changing mouse speed and wake up

You may change the mouse speed and wake up time.

The mouse speed (or "acceleration") determines how far the mouse pointer moves on the screen whenever you move the mouse on your desk (or whatever). Grossly: if the speed is high, the pointer moves a lot with a feeble mouse movement. If the speed is low, the pointer moves less and you need to move the mouse a lot further to move the pointer on the screen. The speed also commands the gradual acceleration of the mouse pointer when the pointer is moved via the cursor keys rather than the mouse.

The mouse "wake up" is the mouse movement that is necessary to show the pointer on the screen when the pointer isn't already visible, for example if it is in a window that is waiting for keyboard input (blinking cursor). This can be easily seen in a Basic input window. The pointer ,normally isn't visible in that window, it becomes visible when you move the mouse. Try it, you will see what I mean.

The command for this is MS_SPD and its syntax is:

```
MS_SPD acceleration [,wake_up]
```

Both parameters range from 0 to 9 and the wake up parameter is an optional parameter.

You can also use the QPAC II "SYSDEF" menu and see how these two parameters change the behaviour of the mouse.

II - Commands for Blobs and Patterns.

Blobs and patterns were already defined in an earlier instalment of this series, please refer there if in doubt.

There are several commands which make the use and creation of blobs and patterns a bit easier:

A - Pattern creation

Here is a command that is useful to create a pattern of a bit more complicated design. Indeed, you may wish to design an image (for example with a painting program) and convert it into a pattern later on. This is pretty nifty as you don't have to care about how to make a pattern in the more complicated way. The command for this is MKPAT:

```
MKPAT address,buffer
```

-› * ***buffer*** is a buffer holding the painting, which was created, for example, with the PSAVE function (which was already covered in this series). The content of this buffer will be transformed into a pattern which will be put at address.

-› * ***address*** is the address in memory where the image converted into a pattern will lie. You must have reserved this address (for example with RESPR or ALCHP) and have enough space at the address for the resul-

ting pattern (including the header). This address can then be used whenever you need a pattern.

Thus note that you need to know the memory size for the pattern before you start this operation. You can get to know the necessary size by using the SPRSP function which we already have seen in an earlier instalment of this series - just use the x size of the buffer and half of the y size of the "buffer" - and then add 18 to take into acount the header.

The pattern (and the image in the buffer) must be at least 16 pixels wide (and the pattern will normally be cut to a length that is a multiple of 16 pixels).

B - Writing blobs and patterns

Once you have created a blob and a pattern you can "write" them out to the screen, i.e. have them appear anywhere you want. Please be reminded that a blob without a pattern, and a pattern without a blob are invisible.

1) WBLOB

This command writes a blob (Write BLOB) with its corresponding pattern to specific screen coordinates:

```
WBLOB [#channel,]x, y, blob, pattern
```

-› * obviously, **x** and **y** are the screen coordinates where the blob is to be written. 0,0 is the top left hand, and these coordinates are in pixels, relative to the window origin of the channel given as parameter.

-› * ***blob*** and ***pattern*** are, of course the pointers to the memory addresses where you can find the blob and pattern to be written out.

As usual, the channel parameter will default to #1 if it isn't specified. The blobs and patterns are written into the channel window at the specified coordinates. If the coordinates are outside the window, there is no error but the blobs and patterns will not be drawn. Pattern should be a multiple pf 16 pixels wide. Some (pretty old) versions of the Pointer Interface do NOT check whether the parameters are really blobs and patterns - if they aren't there is a good chance that the machine will crash. Hence - make sure!

2) LBLOB

The LBLOB (Line of BLOBs) command allows you to print one or several lines of blobs on the screen:

```
LBLOB [#channel,] xpos, ypos, blob, pattern
```

-› * ***xpos*** and ***ypos*** are the screen coordinates. You may combine them with the TO operator:
`xpos,ypos TO x1pos,y1pos (TO x2pos,y2pos etc)`
just like you would with the S*Basic LINE command.

-› * ***blob*** and ***pattern***, are the same pointers to blobs and patterns as described above.

3) SPRAY

This interesting little command is like WBLOB, but instead of writing an entire blob, it only writes out a random number of pixels of it. This is really only necessary in some kind of painting program, where, instead of drawing a continouous line, you would want to write out a more diffuse line. The "pencil" thus just leave a spray of pixels (hence the name) with a diffuse line.

```
SPRAY x, y, blob, pattern, pixels
```

-› * the first four parameters are like for WBLOB.

-› * ***Pixels***: This parameter gives the (approximate) number of quantity of pixels that will be drawn. However, even if you paint several times over the same place with the same pixel, you will not be sure that the entire blob will be drawn out (after all, you have WBLOB for that!)

This concludes this little series on QPTR. I hope you have enjoyed it more than I have....