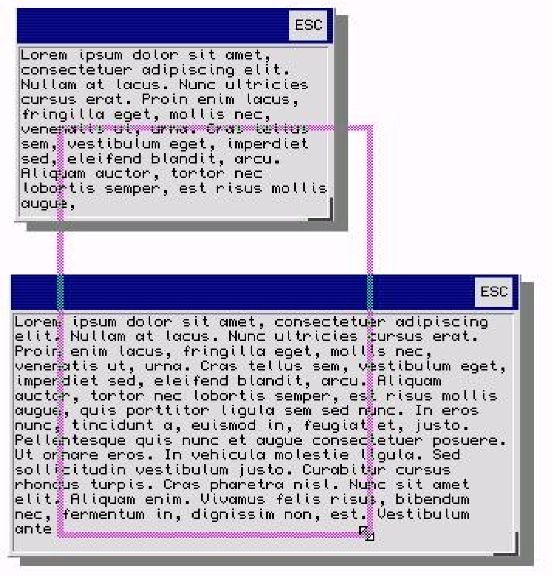
Interactive Resizing

and other bits and bobs..

Some time ago, on the QL-Users' email list, Ralf Reköndt wrote "..how nice [it would be] to use the Windows facility of changing a window size in the lower right edge of a window.." (qlusers 30/11/2008, Program Updates). Wouldnt it just! Thanks to Wolfgang Lenerz and Marcel Kilgus, on later versions of SMSQ/E we now have an interactive window move routine that allows the user to click on the window move button and visibly drag a window to another location on the screen and drop it there. If the platform is too slow to comfortably move the contents with the window, you can opt for just moving a representative outline instead. This only works with a mouse, so if you use the keyboard instead, or if you switch off the feature, SMSQ/E reverts to the old Qram method of moving the window by icon.



```
Lorem ipsum dolor sit amet,
consectetuer adipiscing elit.
Nullam at lacus. Nunc ultricies
cursus erat. Proin enim lacus,
fringilla eget, mollis nec,
venenatis ut, urna. Cras tellus
sem, vestibulum eget, imperdiet
sed, eleifend blandit, arcu.
Aliquam auctor, tortor nec
lobortis semper, est risus mollis
augue,
```

```
Lorem ipsum dolor sit amet, consectetuer adipiscing
elit. Nullam at lacus. Nunc ultricies cursus erat.
Proin enim lacus, fringilla eget, mollis nec,
venenatis ut, urna. Cras tellus sem, vestibulum eget,
imperdiet sed, eleifend blandit, arcu. Aliquam
auctor, tortor nec lobortis semper, est risus mollis
augue, quis porttitor ligula sem sed nunc. In eros
nunc, tincidunt a, euismod in, feugiat et, justo.
Pellentesque quis nunc et augue consectetuer posuere.
Ut ornare eros. In vehicula molestie ligula. Sed
sollicitudin vestibulum justo. Curabitur cursus
rhoncus turpis. Cras pharetra nisl. Nunc sit amet
elit. Aliquam enim. Vivamus felis risus, bibendum
nec, fermentum in, dignissim non, est. Vestibulum
ante
```

Outline resizing is probably the best we can expect

Ideally, interactive resize should be implemented at the system level: Moving the pointer to the edge of a window would change the sprite to an appropriate Resize Sprite and the user could just drag the edge to resize the window and drop it again (by releasing the mouse button) once the desired size had been attained. A spare byte in the Menu Definition could be used to indicate which of the four edges and four corners of the window were resizable.. Dream on.

In the mean time, the following demo program attempts to achieve something similar here and now. Due to issues in the lower layers of the Pointer Environment [PE] it seems to be impossible to achieve quite the same effect as a typical Windows resize, but it works well enough I think.

Just a small point: I dont try to update the window's contents during resizing as this requires a lot of computational power - like a dedicated graphics chip! It is also not trivial to achieve, and it seems unlikely that it could be done in any standard way with the current PE object model, though I could be wrong.

This program, **wresz**, demonstrates mainly three techniques: 1) Interactive outline resizing, 2) unconventional use of Application Windows [AWs], 3) hidden Loose Items [LIs]. Incidental to (2) are techniques for reading AWs by dropping down to discrete routines when AWs are encountered and handling the inherent limitations (ie keystrokes are local to each AW). I make no claims of being either the first or the last word in how the desired effects may be achieved, nor in the perfection of their implementation, but I hope this demo will be of use all the same. The controls are operated as follows:

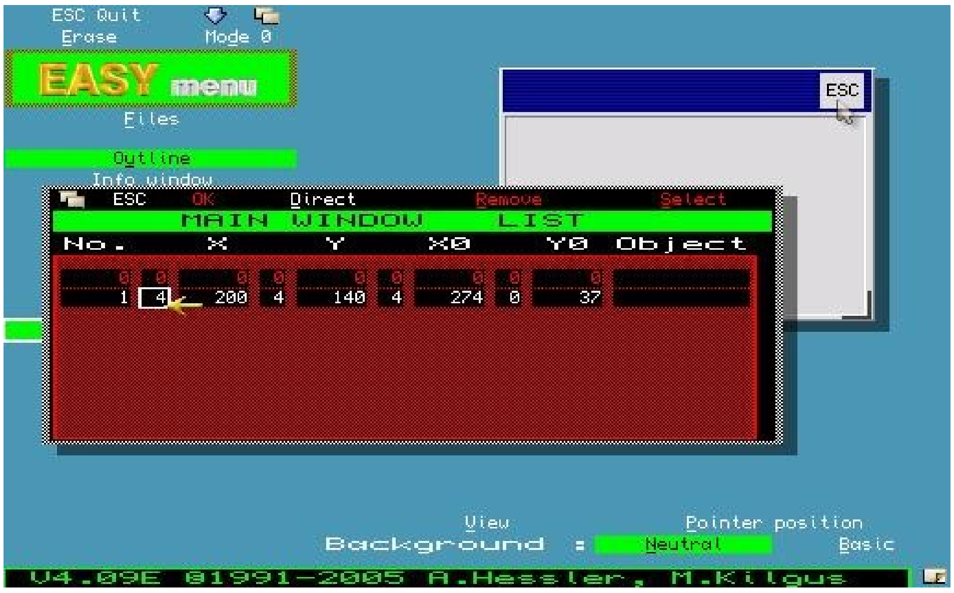
Move: Move the pointer to the titlebar and it changes to a mini window move icon. Click (HIT or DO with the mouse buttons) on the titlebar and interactive move is initiated according to your WM_MOVEMODE settings. You can also move the window by hitting the Ctrl+F4 key [CF4] anywhere in the window. This turns on the internal window move sprite which you move, by mouse or cursor keys, to the desired location and then click again to have the window redrawn there, as in the bad old days.

Resize: Move the mouse to the Bottom Right Corner [BRC] of the main window. The cursor changes to the move corner sprite. Click on the BRC and an outline of your window appears. This can be resized between the minimum allowed size and the maximum available size. Click again and the window is redrawn at the desired size. In the complete version of this program, that I hope will be made available to readers, the text contents of the window is reformatted and displayed at the new size. Resize can also be initiated by pressing the function key Ctrl+F3 [CF3]. This immediately fires up the resizing routine and you can resize using the mouse or cursor keys.

Finally, ESCape [ESC]. Press ESC in any AW without a handler and the program terminates. Press ESC in the Move AW or in one of the Resize AWs and the pointer jumps to the middle of the window. You have to press ESC again to quit. This demonstrates that key presses are local to each AW. Great when you want that behaviour, a nuisance when you dont.

Thanks to Marcel, Albin Hessler's EasyPointer [EZP] toolbox has been updated to incorporate most of the facilities of PE. In other developments, George Gwilt has created an alternative toolbox that lends itself better to the Turbo world view, while Norman Dunbar continues his mission of documenting, and educating us in the ins and outs of PE. This is good stuff and deserves our gratitude and support.

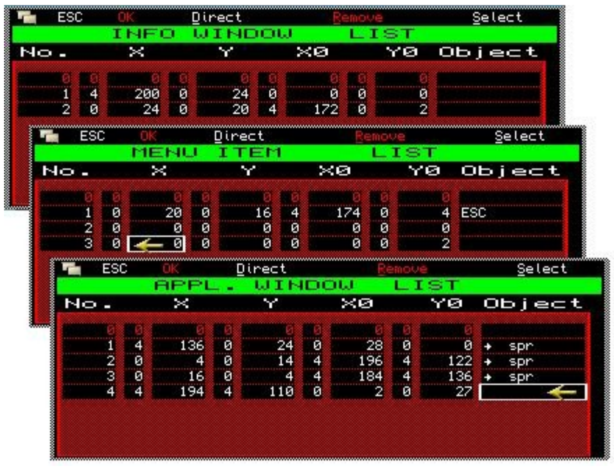
I make no apologies, however, for being a dyed-in-the-wool SMSQ/EasyPointer/QLiberator adherent, as I have been so since their respective beginnings, and they continue to provide good mileage. **wresz** depends on those systems to develop, compile and, to some degree, to run. I leave it to the experts of those other faiths to re-work or produce their own version of what I am trying to show here. If you are acquainted with QDOS/ EasyPEasy/Turbo, etc, with respect to PE programming, you may find it not too difficult to adapt it accordingly. The compiled program should, however run on all systems sporting PE2. As it stands, you will need to be running recent versions of SMSQ/E (v3.13+), EasyPointer (v4.09+) and QLib (v3.35+) to run or develop this code further. You will also need the latest **ptrmen** (V4.08+) toolkit (included with EZP).



Main window. Note the scaling flags at X, Y and X0

For this project youll first need to prepare a suitable menu. The illustration shows my menu design in EasyMenu. A 200x140 pixel window with a titlebar containing an ESC button. However, it isnt quite as simple as it looks! The next picture, a composite, shows its other components. There are two Information Windows [IWs], the first being the titlebar, the second creates a border around the ESC item, to avoid messing up the display when the pointer outlines it. Next there are three LIs: The ESC item, and two **hidden LIs**. As you can see, the latter have no dimensions and are placed in the top left corner [TLC] of the window. Although I have made them 0x0, I have offset their positions to avoid confusing the application (and the programmer!). I can no longer remember whether this is strictly necessary in the former case, as EZP and Wman have evolved over the years. The ESC item's attributes and selection key is what youd expect. The two others' selection keys are CF4 and CF3 – or the standard keystrokes for Move and Resize, respectively.

Finally, there are four Application Windows. The first one is the Move item and covers part of the titlebar and is pixel-aligned to it so as to be invisible. You cannot have a LI covering an



No.		X		Y		X0		Y0	Object
0	0	0	0	0	0	0	0	0	
1	4	200	0	24	0	0	0	0	
2	0	24	0	20	4	172	0	2	

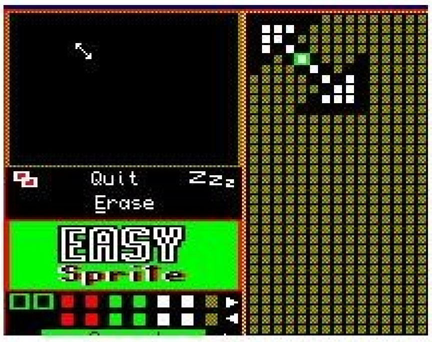
No.		X		Y		X0		Y0	Object
0	0	0	0	0	0	0	0	0	
1	0	20	0	16	4	174	0	4	ESC
2	0	0	0	0	0	0	0	0	
3	0	0	0	0	0	0	0	2	

No.		X		Y		X0		Y0	Object
0	0	0	0	0	0	0	0	0	
1	4	136	0	24	0	28	0	0	spr
2	0	4	0	14	4	196	4	122	spr
3	0	16	0	4	4	184	4	136	spr
4	4	194	4	110	0	2	0	27	

Composite of three EZP element menus

AW (or visa versa) so the AW runs from origin 0x0 to the edge of the ESC LI. It is borderless with attributes the same as for the titlebar IW. However, it has its own sprite. You could either stipulate the sprite to be the standard Move Window sprite or, as I have done, make a miniature Move sprite of your own to go with it. If you have more buttons on the titlebar you will have to adjust the AW accordingly and you may perhaps need more than one AW with the same attributes and sprite to fill in the gaps. As you will see later on in connection with the Resize AWs, their function is easy to group as a single item.

The next two AWs are the Resize bars. These are two thin, borderless AWs of about 4x14 pixels located at the BRC of the main window. Actually, one is slightly longer than the other so as to reach right into the corner without overlapping the other. Four pixels wide may be too skinny for some; you may consider it rather fiddly to find it with the pointer. I have made my resize bars visible here, but they could just as well have blended in with



Sample corner resize sprite

the main window border, only revealing their existence when the pointer sprite changes to the corner resize sprite (see illustration).

Many resize scenarios are possible, of course. Using the same technique, you could provide the facility to allow every edge and every corner to be stretched, as most Windows programs do, or just the bottom and/or right edges. But the most useful compact version, to my mind, is the BRC one demonstrated here.

The final preparation youll need to make is to create an **APPA file**. EZP programmers will know what these are and how to create them, but others may require a short explanation: APPA files consist of a collection of all the EZP components your program needs; your menu definition(s), sprites and, if you want, the ptrmen toolkit extension(s) your program uses. EZP includes a program to build these files from the output files produced by EasyMenu and EasySprite. The APPA file needed here must contain the menu definition plus the corner sprite described above. They can then be accessed by the program as **APPA0("*wresz*")** (the menu) and **APPA0("*CornB*")** (the corner sprite).

Below is the code to make it all work. Only salient points are commented and the comments relate to the block immediately above unless otherwise stated:

```
1 REMark $$chan=6
2 REMark $$stak=360
3 REMark $$heap=1600
4 REMark $$asmb=ram1_rz_Wresz_app,0,60
5 :
6 rem Interactive Move and Resize Demo
7 rem by pjwitte jan 2oo9. V1a01
8 :
9 rem Requires SMSQ/E + ptrmen
10 rem Compile with QLIB
11 rem Compiled should run under QDOS
12 :
```

Standard header showing files to be included. During testing these files should be **LRESPR**ed.

```
13 rem Menu definitions
14 awmv% =  1: rem Move
15 awcr% =  2: rem Corner right
16 awcb% =  3: rem Corner bottom
17 awin% =  4: rem Main app win
18 liqu% = -1: rem LI quit
19 limv% = -2: rem Hidden LI Move
20 lirz% = -3: rem Hidden LI Resize
```

These are the object numbers relating to the menu elements. Ill use the mnemonics to refer to the relevant object. Thus **awmv** refers to the Move AW. (Here **awrz** => **awcr** and **awcb**.)

```
21 mvec% = 1+2+8:rem Return conditions
22 irt%  = 48: rem Immediate return
23 dim pv%(16): rem Pointer Record
24 :
25 rem Misc definitions
26 esc% =   27: rem ESCape
27 kcf3% = 241: rem Key CF3 - Wresz
28 kcf4% = 245: rem Key CF4 - Wmove
29 wwd = 0: rem -> Window Working Def
30 :
31 sp_winbg% = 513: rem Window backg
32 sp_winfg% = 514: rem Window foreg
33 ww_xorg = 36:rem WWD offset $24
34 ww_yorg = 38: rem WWD offset $26
35 :
36 rzbt% =   4: rem Resz border thick
37 rzbc% = 228: rem Resz border colour
38 minx% = 200: rem Min window size
39 miny% = 140: rem   (from menu)
40 sizx% = minx%: rem Start size = min
41 sizy% = miny%
42 posx% = -1: rem Start position
43 posy% = posx%
44 k = 0: stat% = 0: rem GLOBal
45 :
46 x% = 0: y% = 0: rem Scratch
47 :
```

Some definitions, variable and constant.

```
48 rem Program start
49 SetWin
50 :
51 REPeat main
52  k = MCALL(#cw\ k, stat%)
53  sel on k
54   = liqu%: mclear#cw: close: stop
55   = limv%, awmv%: MoveWin
56   = awcb%, awcr%: ReszWin
57   = lirz%: ReszWinBRC
58  endsel
59 END REPeat main
60 :
```

Main program loop. This is a standard menu call (**`MCALL`**). It triggers when the user interacts with one of the objects, whether they be loose items (LIs) or application windows (AWs). The ESC button, referred to as **liqu** (LI quit), is quite standard. However, the window move routine, `MoveWin` is triggered both by the hidden LI, **limv**, and a user click on the titlebar AW, **awmv**. The `MoveWin` routine will sort out whether the CF4 function key has been pressed or whether this is an interactive move.
Note that the handling of LI keystrokes, have to be processed in each AW handler or else they are just ignored: Only if the pointer is outside any AWs with a handler, do they get processed in the main loop.
Should your pointer hover about one of the BRC AWs, the PI will take care to display the corner sprite. If you then click on one of those AWs, `ReszWin`, the **awrz** handler, is reached. Finally, the hidden LI, **lirz**, gets it if the CF3 key gets pressed. This drops directly into the interactive resize routine, `ReszWinBRC`.

```
61 deffn ReadAW%(awno%)
62 loc k%
63 rem GLOBal cw, mvec%, pv%
64 rem Read an App Window
65 :
66 rdpt#cw; mvec%: pval#cw; pv%
67 if (pv%(2) + 1) <> awno%: ret esc%
68 k% = pv%(6)
69 if k% = esc% then
70  rem Centre pointer in awin%
71  rdpt#cw; irt%, posx% + sizx% / 2,
    posy% + sizy% / 2
72 endif
73 ret k%
74 enddef ReadAW%
75 :
```

This is a general AW scanner, called by the AW handlers. It returns if the pointer moves out of the relevant AW or if ESC is pressed. If ESC is detected, it centres the pointer in the program window and returns. This behaviour is designed to get the pointer out of the AW and put it somewhere central where any further keystroke commands may be listened for – in this case only Quit.

```
76 defproc SetWin
77 rem GLOBal cw, cd, .siz%, .pos%
78 rem Open main window, get position,
   attach & format display window
79 :
80 cw = fopen("con_")
81 cd = fopen("con_")
82 mdraw#cw; appa0('wresz'), posx%,
   posy%, sizx%, sizy%
83 wwd = mwdef(#cw)
84 posx% = peek_w(wwd + ww_xorg)
85 posy% = peek_w(wwd + ww_yorg)
86 mwlink#cw, awin%, #cd
87 wm_paper#cd; sp_winbg%
88 wm_ink#cd; sp_winfg%
89 rem Display some text
90 enddef SetWin
91 :
```

This routine merely opens the consoles and draws the program window. It also opens a channel that is linked to **awin**; any window IO sent to that channel [#cd] will appear in **awin**. A call to display some text was removed from line `89` as the routine would unnecessarily lengthen the listing, but thats where the action could go.

```
92 defproc MoveWin
93 loc aml, k%
94 rem GLOBal cw, k, posx%, posy%
95 rem Move by WM_MOVEMODE or by key
96 rem Read Move AW
97 :
98 if k = limv% then
99  WinMove: rem Move by key
100 else
101  rep aml
102   k% = ReadAW%(k)
103   sel on k%
104    = esc%: exit aml
105    = 1, 2: wmov#cw; -1: exit aml
106    = kcf4%: WinMove: exit aml
107    = kcf3%: ReszWinBRC: exit aml
108   endsel
109  endrep aml
110 endif
111 posx% = peek_w(wwd + ww_xorg)
112 posy% = peek_w(wwd + ww_yorg)
113 enddef MoveWin
114 :
```

This is the window move handler. Firstly, if CF4 was pressed while the pointer is in any other location than **awmv**, a window move, using EZP's internal routine, ie saving the relative positions of all the window components, is performed. Otherwise, the pointer **is** in **awmv**, and the AW handler is called to read events in that AW. If user presses a HIT or DO while in **awmv** he means to move the window interactively using Wman. Finally, what if the pointer is in **awmv** and the user presses CF3 (line `107`)? He wants to resize, of course, so interactive resize is initiated immediately. The new x/y positions are fetched directly from the Window Working Definition [WWD].

```
115 def proc WinMove
116 loc x%, y%
117 rem Use Wman routine
118 :
119 x% = -1: y% = x%
120 rdpt#cw; irt%, x%, y%
121 x% = x% - peek_w(wwd + ww_xorg)
122 y% = y% - peek_w(wwd + ww_yorg)
123 wmov#cw: rem Move, keep layout
124 rdpt#cw; irt%, x%, -y%
125 enddef WinMove
126 :
```

This wrapper for **`WMOV`** saves and restores the pointer position. I find it annoying that the pointer jumps to the TLC every time I do a window move by keystroke. Using **`WMOV`**`#cw;-1`, which deploys the internal Wman routine, leaves the pointer in situe an restores it on exit, but it also messes up the window furniture and I have as yet found no work-around. Here I read the current position (line `120`), make the positions relative to the window origin, perform the move, and finally restore the position.

```
127 defproc ReszWin
128 loc wsl, k%
129 rem GLOBal cw, k, posx%, posy%
130 rem Read BRC AWs
131 rep wsl
132  k% = ReadAW%(k)
133  sel on k%
134   = esc%: exit wsl
135   = kcf3%: ReszWinBRC
136   = 1, 2: ReszWinBRC
137     rem Restore pointer
138   rdpt#cw; irt%, (sizx%), -(sizy%)
139   = kcf4%: k = limv%: MoveWin
140     exit wsl
141  endsel
142 endrep wsl
143 enddef ReszWin
144 :
```

The handler for **awcr** and **awcb**. Pretty much as for `MoveWin`. As it doesnt know which of the two corner AWs it will be reading, it supplies the universal variable **k**, returned from **`MCALL`**, to the AW reader, which works for either.

```
145 defproc ReszWinBRC
146 loc rzl, cz, ymx%, px%, py%, xl%
147 loc ox%, oy%, cx%, cy%
148 rem GLOBal cw, pv%, sizx%, sizy%
149 rem GLOBal posx%, posy%, rzbt%,..
150 rem Interactive resize routine
151 :
152 rem Store current pointer position
153 cx% = -1: cy% = cx%
154 rdpt#cw; irt%; cx%, cy%
155 :
156 rem Make relative
157 cx% = cx%-peek_w(wwd + ww_xorg)
158 cy% = cy%-peek_w(wwd + ww_yorg)
159 :
160 rem Work out max screen space,
161 rem  draw window & set outline
162 flim#cw; xmx%, ymx%, ox%, oy%
163 xmx% = xmx% - posx%
164 ymx% = ymx% - posy%
165 ox% = sizx% - rzbt%
166 oy% = sizy% - rzbt%
167 close#cd: close#cw
```

```
168 cz = fopen('con_')
169 outl#cz;xmx%,ymx%,posx%,posy%,0,0,0
170 :
171 rem Draw initial window box
172 over#cz; -1
173 :
174 rem left, right, top, bottom
175 block#cz; rzbt%,sizy%,0,0, rzbc%
176 block#cz; rzbt%, oy% - rzbt%, ox%,
    rzbt%, rzbc%
177 block#cz; ox%, rzbt%,rzbt%,0,rzbc%
178 block#cz;ox%,rzbt%,rzbt%,oy%,rzbc%
179 :
180 rem Set appropriate resize pointer
181 rem  sprite and put in BRC
182 sprs#cz; appa0('CornB')
183 x% = posx% + sizx% - rzbt%
184 y% = posy% + sizy% - rzbt%
185 px% = sizx%: py% = sizy%
186 :
187 rem Draw outline interactively
188 rep rzl
189  rdpt#cz; mvec%, x%, y%
190  pval#cz; pv%
191  if pv%(5) <> 0: exit rzl
192  if pv%(6) = esc% then
193   py% = sizy%: px% = sizx%
194   exit rzl
195  endif
196  rem Pointer x/y-position
197  px% = pv%(3): py% = pv%(4)
198  rem Test limits
199  if px% < minx%: px% = minx%
200  if py% < miny%: py% = miny%
201  if px% > xmx% - rzbt%: next rzl
202  if py% > ymx% - rzbt%: next rzl
203  if px% <> ox% or py% <> oy% then
204   rem Blank and draw t, b, l & r
205   block#cz; ox%, rzbt%, 0,0, rzbc%
206   block#cz; px%, rzbt%, 0,0, rzbc%
207   block#cz; ox% - rzbt%, rzbt%,
      rzbt%, oy%, rzbc%
208   block#cz; px% - rzbt%, rzbt%,
      rzbt%, py%, rzbc%
209   block#cz;rzbt%,oy%,0,rzbt%,rzbc%
210   block#cz;rzbt%,py%,0,rzbt%,rzbc%
211   block#cz; rzbt%, oy% + rzbt%,
      ox%, 0, rzbc%
212   block#cz; rzbt%, py% + rzbt%,
      px%, 0, rzbc%
213   ox% = px%: oy% = py%
214  endif
215 endrep rzl
216 :
217 close#cz
218 rem Set new size
219 sizx% = px%: sizy% = py%
220 SetWin
221 if cx%<=sizx% and cy%<=sizy% then
222  rem Restore pointer
223  rdpt#cw; irt%, cx%, -cy%
224 else
225  rdpt#cw; irt%, (sizx%), -(sizy%)
226 endif
227 enddef ReszWinBRC
228 :
```

The final routine here, is also the most complex. It is arrived at either by clicking one of the resize AWs or by pressing the CF3 key anywhere in the window. First the current pointer position is saved in case the routine was reached by keystroke. Then the old program window has to be thrown away and a new window opened, wherein the resize outline, or box, will be drawn. The size of this window need not take up the whole screen, only that part that encompasses the maximum extent that any new window can be redrawn, starting at the old window's origin. A box outline of the old window is drawn for starters (lines `171+`) and the pointer sprite is set to be the corner sprite. The loop (`188+`) reads the pointer, returning on a keystroke, key down or pointer moved event. (**mvec% = keystroke + key down + pointer moved**). Sadly, Wman does not recognise the **key up** event when initially reading a new channel (it always assumes that key up is the initial state). This means you cant just let go of the mouse button to simulate a “drop” as in *drag and drop*; you have to click a second time to terminate the resize operation. This may be a bug, and may therefore one day get fixed. Line `191` says that if the user clicks a second time (the click that got us here was processed elsewhere), to terminate resize with the current size. `192` says that if ESC is pressed resize is aborted and the size reverts to the starting size. If none of these events occurred, the size is read (`197+`) and tested against the limits to see whether it is legal. If the size passes all the tests and is different from the old size (`203`), the old outline is blanked (un-**XOR**ed out) and a new box is drawn at the new size. This continues until the user is satisfied and terminates (or aborts) the operation. Finally, the drawing window is discarded and a new program window drawn with the new size. The pointer is placed at its old position, provided it still fits inside the window, otherwise it is placed at the BRC.

It is not difficult, but it is extremely fiddly to achieve the effect you want, as the tiniest tweaks can alter the behaviour of the interface. Im looking forward to seeing more PE programs that use interactive resize!