QUANTA HELPLINE

Dilwyn Jones

Members wishing to submit helpline requests via email can use the email address helpline@quanta.org.uk or if you prefer to use traditional post, please send the helpline request to me via the address printed inside the front cover of the magazine.

Obviously, we cannot guarantee to answer every query we receive, but we will do our best! Where we have been unable to answer the queries, we may print the help request as an open request in the magazine to ask if any of the readers can come up with a solution. And, of course, if readers feel that they have a better solution than we came up with, or would like to correct any errors we make, please write to us!

Q I have started using QPC2. This version, as in previous versions, does not accept AUTO 100,10. The message is INVALID PARAMETER. When the QL first came out I used AUTO frequently. I use it as the first instruction when writing a program.

A. In fact, strictly speaking, this is not down to QPC as such, more of a change in the way the SMSQ/E operating system (or even more specifically, the SBASIC interpreter) handles program entry and editing.

In SMSQ/E, or more correctly SBASIC, the AUTO command does not exist in the way in which it originally did in the original QL SuperBASIC.

SBASIC accepts the keyword AUTO, but implements it like the ED command in Toolkit 2 – it starts editing at the specified line number. So you can only give one parameter (the line number), like you can with ED. Here is a quote from the SuperBASIC Reference Guide, which explains this:

> ***AUTO - SMS NOTE:***
> *On current versions of SMS, AUTO has been re-coded to be the same as ED, therefore it will not allow a second parameter, and merely places you in ED mode with the cursor at the specified start line number .*

While this may be awkward at first, I suppose the idea is that the ED command gives you a lot more facilities and flexibility.

Q. Is it possible to change the system fonts, so that I can give all my programs a different look? I am aware of the CHAR_USE command to set a different font in a program, but it would be nice to have all programs using the same font.

A. Sadly, this isn't possible in original QDOS as the main system font is held in the ROM and cannot be changed. The designers of SMSQ/E, however, foresaw this requirement and added a command called CHAR_DEF to the SBASIC interpreter.

Each channel can have two fonts, one of which normally covers the usual ASCII range of characters up to code 127 and the other which covers the extended character set including accented characters and symbols with higher ASCII codes.

You can use any standard QL system font, there are plenty out there to download free on the internet from sources such as the Fonts page on my website at http://www.dilwyn.me.uk/fonts/index.html

You can also create your own, using one of the many font editors out there.

Before anyone says anything to correct me, Sinclair and most QL software developers have traditionally used the term "fount" instead of "font".

To change the 'system fonts' used in SMSQ/E, we use a command called CHAR_DEF:

```
CHAR_DEF font1_address, font2_address
```

As you migh expect, the two addresses simply point to the address of a font you have loaded into the common heap memory. To do this, you would test the length of the font file with the FLEN function, use the ALCHP command to allocate that amount of memory (best to add the two lengths together into one block and load the fonts one after the other), then use CHAR_DEF to tell the system where these fonts are.

There are two special values which can be used in place of the address values:

0 = go back to using the ones built into the operating system

-1 = don't change this setting, useful if you only wanted to change one of the two fonts, for example.

So, CHAR_DEF 0,0 resets both character sets to that built into the operating system. CHAR_DEF 0,-1 would reset the first (the one including letters, numbers etc) to that built into the system, while still using the existing second font you loaded.

Here is how to load two fonts.

```
100 REMark set filenames of the two fonts to be used
110 font1$ = 'win1_fonted_serif_font'
120 font2$ = 'win1_fonted_font2_fnt'
130 fontlength1% = FLEN(\font1$)
140 REMark ensure it's an even value
150 IF (fontlength1% MOD 2)=1 THEN fontlength1% = fontlength1%+1
160 fontlength2% = FLEN(\font2$)
170 :
180 REMark allocate memory to hold the fonts
190 font1address = ALCHP(fontlength1%+fontlength2%)
200 IF font1address <= 0 THEN REPORT font1address : STOP
210 font2address = font1address+fontlength1%
220 :
230 REMark load fonts
240 LBYTES font1$,font1address
250 LBYTES font2$,font2address
260 :
270 REMark set the new system fonts
280 CHAR_DEF font1address,font2address
290 :
```

By now, you may have tried running this program and realised nothing has changed if you are using the SBASIC windows #0, #1 and #2. This is because channels already open don't automatically use the new fonts for various reasons, the most obvious being that if the font loading failed, or you accidentally loaded a file of rubbish data, you would no longer be able to see what you are typing in as the letters, numbers and symbols might turn to garbage.

So, the safest thing to do is to open a new channel, such as OPEN #3,scr and test the printed output there with something like LIST #3. If it works as we expect, CLOSE #3 and next we use the original CHAR_USE command to set the fonts for the existing channels:

```
FOR chan = 0 TO 2 : CHAR_USE #chan,font1address,font2address
```

Indeed, this can hold true for programs which have already been started before you load the new fonts. The answer is simple – install the fonts first, before you do anything else such as load other programs, e.g. install the new fonts in your boot programs.

Used in this way, even “front end” programs like Launchpad can take on a new appearance using these new system fonts, as long as the fonts are installed before such programs are started.

You may have come across fonts which include all characters of the first and second QL fonts, some of the ones on my website for example – usually these fonts cover the full range of character codes from 31 (the default chequerboard characters) right up to 191, one of the arrow symbols. To use these simply requires a minor bit of lateral thinking – we load the font as font1 and simply set font 2 to 0. This means that all valid characters will now be printed using the new font 1, but any out of range character codes can still be displayed using the default chequerboard character:

```
CHAR_DEF font1address,0
```

When resetting the fonts back to normal with CHAR_DEF 0,0 you should remember to deallocate any heap area used for the new fonts, e.g. with the command RECHP font1address if you used the program above:

```
CHAR_DEF 0,0 : RECHP font1address
```

It is best to do this with no programs running, as the window channels may not spot the change and carry on trying to use the new system fonts which are no longer there.

Channels such as the SBASIC windows may not realise you have changed the default system font back to normal either. You have to use CHAR_USE commands again to reset them back to normal, but make

sure you have used CHAR_DEF first, otherwise CHAR_USE might end up using the new fonts, since CHAR_DEF is still pointing to them! Confused? I certainly was when I first issued these commands in the wrong order!!!

FOR chan = 0 TO 2 : CHAR_USE #chan,0,0

Q. Is it possible to change the cursor shown by the QL? I'm fed up of just a flashing red rectangle?

A. Not on a QDOS QL, but recent versions of SMSQ/E can do this, by allowing a sprite to be loaded and used as a cursor. Recent versions of SBASIC have new keywords which allow you to load a sprite of 6 pixels across and 10 pixels down.

CURSPRLOAD 'filename' is all that all is required once you have a suitable sprite. You can create 6x10 pixel sprites with most QL sprite editors such as Easysprite. For example, run this program to build a little green diamond shape sprite to use in place of the standard cursor.

```
100 f$ = "ram1_diamond_spr"
110 fl = 104
120 base = ALCHP(104)
130 RESTORE
140 FOR a = 0 TO fl-1
150   READ byte
160   POKE base+a,byte
170 END FOR a
180 SBYTES ram1_diamond_spr,base,fl
190 RECHP base
200 PRINT f$;' created.'
210 STOP
220 :
230 DATA 1,0,0,0,0,6,0,10,0,0
240 DATA 0,0,0,0,0,12,0,0,0,48
250 DATA 0,0,0,0,48,0,0,0,48,0
260 DATA 0,0,120,0,0,0,120,0,0,0
270 DATA 252,0,0,0,252,0,0,0,120,0
280 DATA 0,0,120,0,0,0,48,0,0,0
290 DATA 48,0,0,0,48,48,0,0,48,48
300 DATA 0,0,120,120,0,0,120,120,0,0
310 DATA 252,252,0,0,252,252,0,0,120,120
320 DATA 0,0,120,120,0,0,48,48,0,0
```

```
330 DATA 1,0,0,0
```

Once you've run this little program to generate a 104 byte sprite called diamond_spr in ramdisc ram1_, enter the command CURSPRLOAD 'ram1_diamond_spr' to set the sprite as SBASIC's new cursor.

There are two other handy extensions for turning the sprite cursor off or on, and not just for BASIC. By giving either the job name, or job number and job tag (you can see these by listing the running job with the JOBS command).

```
CURSPRON "jobname"
```
or
```
CURSPRON jobnum,jobtag
```

turn on the sprite cursor, while

```
CURSPROFF "jobname"
```
or
```
CURSPROFF jobnum,jobtag
```

turn off the sprite cursor. To allow a job such as SBASIC to turn off its own cursor sprite, just issue the command CURSPROFF -1.

Using these commands you can replace the flashing cursor for many programs as long as you know their details, for example, to replace the Xchange cursor: CURSPRON "Xchange"

The sprite may include transparent pixels, that's how I was able to cut out the corners to create the diamond shape, so you could (theoretically at least) create a rounded cursor, or a hollow box, a cross, a line or even just a dot, probably even an invisible 6x10 cursor although I didn't dare try that one!

If you change the cursor for a pointer driven program, it doesn't replace the standard pointer arrow, just the flashing cursor you get when asked to INPUT something, for example. So if you were to try to change the cursor for Launchpad or such a program, it would continue to use its standard pointer arrow for most things, only using the new cursor sprite when asking you to enter something from the keyboard.

Q. When I used a QL,it was easy enough to save a copy of the screen with the command sbytes filename,131072,32768. On modern systems like QPC2 the screen can be bigger and more colourful. How can I work out how to save one of these bigger screens – I'm told they can be longer and stored in a different place in memory.

A. As you have realised, saving a screen picture is a bit more complex on SMSQ/E systems since the screen size can vary, and its address in memory can change too. Sorry everyone, this will be a bit of a long answer!

The good news is that SBASIC has extensions which can help greatly to simplify saving screen pictures.

In order to save a screen, we need to know

- Its base address in memory
- How many pixels across
- How many pixels down
- The length of each line in bytes

To find where the screen starts in memory, we use the SCR_BASE function:

```
LET base_address = SCR_BASE
```

To find how many bytes between the start of one line and the start of the next, SBASIC provides the SCR_LLEN function (standing for Screen Line Length).

```
LET line_length = SCR_LLEN
```

The two functions SCR_XLIM and SCR_YLIM tell us the screen x limit and screen y limits. Used without a channel number, or with the channel number of the lowest open channel (the primary channel), they tell us how many pixels across and down the screen respectively.

```
LET across = SCR_XLIM
LET down = SCR_YLIM
```

So, adding the information from these functions to an SBYTES command, we can now save a screen in any mode like this, using this simple two line program from the SMSQ/E manual:

```
ssz = SCR_LLEN * SCR_YLIM : REM screen size
SBYTES filename$, SCR_BASE, ssz : REM save the screen
```

You could easily combine them into one line, like this:

```
SBYTES filename$, SCR_BASE, SCR_LLEN*SCR_YLIM
```

And to reload the screen later needs just one simple command, as long as you know that the file is the right size for the screen mode and resolution you are using:

```
LBYTES filename$,SCR_BASE
```

As ever, though, there is a minor fly in the ointment. The Aurora card. Some of its display modes surprisingly do not use the full line length! The aurora is always geared up to have a line length wide enough to hold 1024 pixel wide screens in 4 colour mode, a fixed line length of 256 bytes per line. Saving and reloading will probably work OK as long as you are loading the screen back on the same computer. If you used the above program line to save the screen, it would seem to work, but it would add the unused memory space to the right of the screen, so you might get a bigger screen file than you expected and it might either have a blank or random section to the right of the picture.

If you wish to save a screen comprising of just the actual visible line length, things become a little more complex, but not impossible by any means. What you have to do is to calculate how wide each line should be, and save that amount from each Aurora video line in memory, line by line, using the PEEK$ function to fetch the relevant part of the video of each line. To calculate the actual length of the visible line, we have to know how many bits or bytes correspond to each pixel and save the right number of bytes for each line.

In MODE 4 and MODE 8, we divide the number of pixels across by 4 to get the number of bytes. In theory we have to make sure this rounds up to an even number of bytes. In practice, it always does for a full screen.

MODE 4 and MODE 8: 8 pixels per two bytes.

```
LET visible_length = SCR_XLIM DIV 4
```

256 colour (8-bit) screen mode 16: One byte per pixel
```
LET visible_length = SCR_XLIM
```

16 bit colour modes 32 or 33 (QPC2, QXL, SMSQmulator, Q40, Q60): 2 bytes per pixel
```
LET visible_length = 2*SCR_XLIM
```

To decide which of those to use, we can use the DISP_TYPE function to tell us the screen display mode:

```
LET display_mode = DISP_TYPE
SELect ON display_mode
  =0,4,8 : LET visible_length = SCR_XLIM DIV 4
  =16    : LET visible_length = SCR_XLIM
  =32,33 : LET visible_length = 2*SCR_XLIM
END SELect
```

If new display modes are introduced in the future, it is simple enough to add to this once you know how many bytes per pixel. For example, a “true colour” 24-bit mode (MODE 64) would use 3 or 4 bytes per pixel (most probably a long word per pixel – 4 bytes).

So now we know how much of each Aurora video line to save, we can create a program which will step through the video memory a line at a time, saving just the part of each line needed:

```
OPEN_NEW #3,filename$
REMark set visible_length as above
...
LET video_address = SCR_BASE
FOR y = 0 TO SCR_YLIM-1
  PRINT #3,PEEK$(video_address,visible_length);
  LET video_address=video_address+SCR_LLEN
END FOR y
CLOSE #3
```

Please note that the line starting with PRINT #3 has a semi-colon at the end. This is to prevent it adding an unwanted linefeed character after the PEEK$ function.

Things start getting more complex when we need to read this back, as we need to read a line at a time from the file and poke it into the screen in the right place. Fiddly, but not impossible:

```
OPEN_IN #3,filename$
REMark set visible line length as above
...
LET video_address = SCR_BASE
FOR y = 0 TO SCR_YLIM-1
  LET lne$ = "" : REM fetch copy of current line
  FOR x = 1 TO visible_length : lne$ = lne$&INKEY$(#3)
  REMark place the line into the screen memory
  POKE$ video_address,lne$
  REMark move pointer to start of next line
  LET video_address = video_address + SCR_LLEN
END FOR y
CLOSE #3
```

This routine will be quite slow for large screens, as it reads the graphics data back byte by byte, line by line. It may be possible to speed it up by using a command such as INPUT$ from some toolkits (I think Turbo Toolkit and DJToolkit have such functions) which read a given number of bytes from a channel at a time, letting you replace the two lines LET lne$="" and FOR x=1 TO… with something like:

```
LET lne$=INPUT$(#3,visible_length)
```

When I tried this on my computer, the speedup was quite drastic compared to using INKEY$.

One word of warning: in the higher colour and resolution modes, saving screens can result in very large files. I was using 16-bit colour mode in 1024x768 pixel resolution and it created a graphics file 1,572,864 bytes long. Suddenly I realised why floppy disks can go out of fashion!

If you want to work out in advance what size a file will be, the basic calculations are as above. Work out how many bytes across a line for the mode you are using, multiply it by the depth of the screen in pixels and you have a length (in bytes) of the resultant screen.

A final word on this, for those who may want their software to also work on non-SMSQ/E systems where these extensions are not present. On the overwhelming majority of QDOS systems (with the exception of the

uQLx and Atari emulator cards) the screens will always be 32,768 bytes long, starting at address 131072 since standard unpatched QDOS cannot provide for larger displays or more colours. The question then is how to protect your program from errors caused by non-existent extensions. This is actually easier than you might think: check the ROM version and use IF statements to take different actions. Here's a simplified routine, which may not work on the Aurora for the reasons described above:

```
v$ = VER$
REMark default values for QDOS
base=131072
size=32768
IF v$ = 'HBA1' THEN
  REMark aha, running on SBASIC
  base=SCR_BASE
  size=SCR_YLIM*SCR_LLEN
END IF
SBYTES filename$,base,size
```

Another way is to make use of freely available extensions in toolkist which work on both QDOS and SMSQ/E. One example is the Display-Code toolkit I wrote a few years ago, published in QL Today at the time and now available to download as freeware from my website's Toolkits Page at http://www.dilwyn.me.uk/tk/index.html .

Quanta members can also use a set of extensions called Screen Parameters, written by Bruno Coativy, available on Library Disk UT01. This handy little package provides extensions to tell you the screen address, size, line length and so on.

Q. OK, I think I get that. Now, how can I cut and paste parts of the picture from one place on the screen to another?

A. Now it becomes really complex! If you are familiar with pointer environment programming software, it is best to use existing routines within the programming tools to do this, as the pointer environment provides routines to save and restore areas of the screen. Not only does this prevent you having to write unnecessary code, it provides the most future-proof code basis to ensure that the program you wrote

does not run into problems in the future when system designers bring in new video modes, for example.

Sometimes, though, your needs cannot be fully met by existing software, and anyway, we are all tinkerers, it is our right to explore, write code, peek and poke here and there – that's what being a QL user is all about!

To offer a solution to this, I am going to print a slightly modified listing based on a routine from Tobias Fröschle, which he published on QL Forum. It involves working out the address of where the top left of the block to be copied starts, how many bytes wide it is, how many lines deep and from that how many bytes needed to store those lines.

To keep things as simple as possible, this routine does not try to cater for the Aurora special case described above, and it requires a command such as MOVE_MEMORY to move chunks of memory about. Such commands are commonly available in many toolkits, probably the best known being the Turbo Toolkit.

In modes 4 and 8, because there are multiple pixels per byte, it is easier to write these routines if you make sure that the block to be saved is a multiple of 8 pixels wide, and that the block starts on a word boundary (a multiple of 2 bytes or 8 pixels) across the screen. Otherwise you get into the realms of copying part bytes and shifting and rotating bits and pixels and that is simply beyond the scope of this article!

This version of the routine builds up an area save header similar to those used by the pointer environment, for three reasons:

- To be compatible with pointer environment area save files
- To allow the copied area to be saved to a file as a _pic file, one of the most common QL graphics file formats
- To allow the copied area to be printed with the SDUMP command, by supplying the address value returned from this function to the SDUMP command

To call the routine, call the ScrSave function with four parameters representing the x and y co-ordinates of the top left of the part to be saved, plus the width and height (in pixels) of the area to be saved in

memory or copied. In QL modes 4 and 8, the x and w values (origin across and width) **MUST** be multiples of 8 pixels wide.

```
start_address = ScrSave(x_origin,y_origin, wide, high)
```

The routines have no error trapping as they stand, to keep them fairly short for publication, I'm sure readers will have fun adding this.

```
1000 DEFine FuNction ScrSave (x, y, w, h)
1010   LOCal llen, scrBase, bpp, pixlen, blen, maddress
1020   LOCal bstart, mneed, srcPtr, dstPtr, lineNumber
1030   :
1040   REMark set defaults for QDOS in case no SMSQ/E
1050   scrBase = 131072 : llen = 128 : scrMode = 0
1060   bpp = (128 * 8 / 512) / 8 : REMark Bytes per pixel QL
modes (0.25)
1070   IF VER$ = "HBA" THEN
1080     REMark use SMSQ/E values
1090     scrBase = SCR_BASE
1100     llen = SCR_LLEN
1110     scrMode = DISP_TYPE
1120     SELect ON scrMode
1130       =0,4,8 : bpp = (llen * 8 / SCR_XLIM) / 8 : REMark QL
modes
1140       =16    : bpp = 1 : REMark 8-bit colour mode
1150       =32,33 : bpp = 2 : REMark 16-bit colour mode
1160     END SELect
1170   END IF
1180   :
1190   REMark length of one window scanline in bytes, rounded up
1200   REMark in QL colour modes where there is more than one
1210   REMark pixel per byte
1220   IF scrMode < 16 THEN
1230     blen = bpp * 8 * INT ((w + 7) / 8)
1240   ELSE
1250     blen = bpp * w : REMark higher colour modes
1260   END IF
1270   :
1280   REMark start address within screen
1290   bstart = scrBase + (y * llen) + (x * bpp)
1300   :
1310   REMark memory needed +5 words for area save header
1320   mneed = blen * y + 10
1330   maddress = ALCHP(mneed)
1340   :
1350   REMark set up area save header before graphics
1360   POKE_W maddress, HEX('4afc'): REMark header flag 19196
1370   POKE_W maddress + 2, w      : REMark remember width
1380   POKE_W maddress + 4, h      : REMark remember height
```

```
1390   POKE_W maddress + 6,blen    : REMark remember line
increment
1400   POKE maddress + 8,DISP_TYPE : REMark remember mode number
1410   POKE maddress + 9,0         : REMark zero the spare byte
1420   :
1430   dstPtr = maddress + 10 : REMark start just after header
1440   srcPtr = bstart        : REMark start point of first line
in screen
1450   FOR lineNumber = 1 TO h
1460     MOVE_MEMORY srcPtr TO dstPtr, blen
1470     srcPtr = srcPtr + llen : REMark next line down screen
1480     dstPtr = dstPtr + blen : REMark next line store area
1490   END FOR lineNumber
1500   RETurn maddress
1510 END DEFine ScrSave
1520 :
```

It is possible to adapt the routine to allow the copied area of the screen to be saved to a file if you wish, a copy-to-file routine. The easiest way to do this is to make the 'mneed' variable global, by removing it from the LOCal list in line 1020. That way, all you need to do is to use an SBYTES command to save the graphic to a file using the start address and length values:

```
SBYTES filename$, start_address, mneed
```

This saves a standard QL _pic file. It can be reloaded later by allocating the same amount of memory as the length of the file, and will then be the same as a copied area from the routine above.

```
start_address = ALCHP(FLEN(\filename$))
LBYTES filename$, start_address
```

Slight digression: let me formally document the _pic file format here. It's basically a block of graphics, with a 10 byte header preamble in the file storing a flag to identify the type of flag, the idth and height of the picture (in pixels), the line increment of the graphics in the file (bytes from start of one line to the next) and the mode number, plus a single spare unused byte. The offset and length below are in bytes from the start of the file:

Offset	Length	Function
0	2	Flag of $4AFC (decimal 19196)
2	2	Width, in pixels
4	2	Height, in pixels

6	2	Line increment, in bytes
8	1	Screen mode number
9	1	Spare, unused byte.
10		start of graphics, in same format as screen mode it was saved from.

Having copied the relevant part of the screen, you'll need a matching routine to paste it back somewhere else on the screen. Call this with the address returned by the routine above as the first parameter, to tell it where to paste FROM, then the x and y co-ordinate of the point on the screen the pasted image should appear at. Note that for QL modes 4 and 8 the x co-ordinate should be a multiple of 8 pixels across, and the co-ordinates cover the whole screen, starting from top left, down to the bottom right. Graphics should be pasted back to the same screen mode as that they were saved from, otherwise it will either look very odd, or possibly even crash the computer!

```
1530 DEFine PROCedure ScrPaste (srcAddr,x,y)
1540   REMark paste the part screen stored at scrAddr
1550   REMark to the co-ordinates x,y (top left)
1560   REMark in mode 4 or 8, x must be multiple of 8
1570   LOCal scrBase, llen, blen, bpp
1580   LOCal srcPtr,dstPtr,lineNumber
1590   :
1600   REMark set defaults for QDOS in case no SMSQ/E
1610   scrBase = 131072 : llen = 128
1620   bpp = (128 * 8 / 512) / 8 : REMark Bytes per pixel for
mode 4
1630   IF VER$ = "HBA" THEN
1640     REMark use SMSQ/E values
1650     scrBase = SCR_BASE
1660     llen = SCR_LLEN
1670     scrMode = DISP_TYPE
1680     SELect ON scrMode
1690       =0,4,8 : bpp = (llen * 8 / SCR_XLIM) / 8 : REMark QL
modes
1700       =16    : bpp = 1 : REMark 8-bit colour mode
1710       =32,33 : bpp = 2 : REMark 16-bit colour mode
1720     END SELect
1730   END IF
1740   :
1750   REMark length of a scan line
1760   blen = PEEK_W(srcAddr+6)
1770   :
1780   REMark where are we pasting from?
1790   srcPtr = srcAddr+10
1800   :
```

```
1810   REMark where are we pasting to?
1820   dstPtr = scrBase + (y * llen) + (x * bpp)
1830   :
1840   FOR lineNumber = 1 TO PEEK_W(srcAddr+4)
1850     MOVE_MEMORY srcPtr TO dstPtr, blen
1860     srcPtr = srcPtr + blen
1870     dstPtr = dstPtr + llen
1880   END FOR lineNumber
1890   :
1900   REMark to release heap memory automatically after paste,
1910   REMark just remove the REMark from the next line
1920   REMark RECHP srcAddr : srcAddr = 0
1930 END DEFine ScrPaste
```

As it stands, the ScrPaste procedure does not release the memory used to hold the copy in memory/ Line 1920 shows how to do this automatically after each paste – just remove the REMark statement before the RECHP srcAddr:srcAddr=0 statements.

Alternatively, if you think you might want to make multiple pastes of the same image, leave that as it stands and elsewhere in your program put the RECHP srcAddr:srcAddr=0 statements to release the heap memory once you have finished pasting.

OK, so now you know how to load and save screen and _pic files, plus how to copy and paste graphics, off you go and write a graphics program, please!