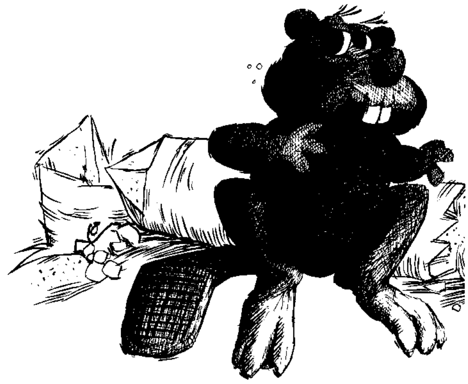
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT



Programming the Electrohome QUICKPEL Graphics Board

Darrell R. Raymond
Vilhelm Böggild
Data Structuring Group

CS-86-62

November 1986

Programming the Electrohome QUICKPEL Graphics Board†

Darrell R. Raymond
Vilhelm Böggild

Data Structuring Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

The Electrohome QUICKPEL is an IBM PC compatible graphics board designed primarily for decoding and displaying NAPLPS graphics. The board can also serve as a powerful coprocessor for custom graphics software, but its coprocessor capability has been neither adequately described nor supported. This document is a tutorial for programming the QUICKPEL, and it contains a substantial collection of facilities that simplify the writing of such programs.

November 20, 1986

† This research was funded by grant G1154 of the Natural Sciences and Engineering Research Council of Canada.

Programming the Electrohome QUICKPEL Graphics Board†

Darrell R. Raymond
Vilhelm Böggild

Data Structuring Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

1. Introduction.

The Electrohome QUICKPEL is a powerful IBM PC-compatible graphics board whose main function is the decoding of NAPLPS (North American Presentation Level Protocol) graphics, fully defined in the CSA T500 standard [1] and more commonly known as *videotex* graphics. The board can deliver a resolution of 200 x 256 pixels, and shows up to 16 of a possible 512 colours at any one time. The main functionality of the board is provided by an 8088 CPU, a ROM-based NAPLPS decoder, and a ROM-based multitasking executive known as ETEX. These and other details are described further in the QUICKPEL *User's Guide,*[2] *Programmer's Guide,*[3] and *Technical Reference Manual.*[4]

The QUICKPEL board is based on a standalone videotex decoder which has the ability to run "telesoftware" — locally-executed but remotely-accessed code which is downloaded over the communications link when requested. In the QUICKPEL board this is really the capability to run "coprocessor" software, since the board is more likely to receive its programs directly from the PC in which it is resident than from a more remote site. The ability to execute graphics-related programs without consuming the PC's own processor time is an attractive feature, since it frees the PC to do other kinds of processing during the display and modification of graphical data.

Writing programs for the QUICKPEL requires knowledge of graphics, data transfer, 8088 assembler, PC-DOS, and ETEX, as well as a strong tolerance for the host of quirks and bugs that accompanies this collection of software and hardware. We have not attempted to provide a formal description of the board or its utilities. Such a description would almost surely be erroneous in many places, but more importantly, even a correct reference manual isn't very helpful when writing the "first" program. Instead we present a series of example programs for the QUICKPEL, and try to illustrate different features and problems of programming the board as we go along. Not all of our assumptions or methods are optimal (or perhaps even completely correct), but by following them it is possible to get programs running fairly quickly. This document is not meant to

```
ts          equ      04h
ec_stop     equ      026h
tele_ram    equ      0c000h
video_ram   equ      010a0h

telesoft    segment  para public
            assume   cs : telesoft, ds : telesoft, ss : telesoft
            org      100h
begin:      jmp      start
            dw       64h dup (0h)
stack       dw       0h

start       proc     near
            sti
            mov      ax,tele_ram         ; ds = ss = telesoftware ram
            mov      ds,ax
            mov      ss,ax
            mov      sp,offset stack     ; set up this task's stack area
            mov      ax,video_ram        ; es = video ram
            mov      es,ax

            mov      si,0
            mov      di,256*200          ; set di to last pixel
            dec      di
row:        mov      cl,es:[si]          ; swap the pixels
            mov      al,es:[di]
            mov      es:byte ptr [si],al
            mov      es:byte ptr [di],cl
            dec      di
            inc      si
            cmp      di,si
            ja       row
            mov      bx,ts
            int      ec_stop

start       endp
telesoft    ends
            end      begin
```

Figure 1. *Mirror.*

replace the documentation, which should be read by every programmer with great caution and much skepticism. Instead, we want to draw your attention to some of the fine and nasty points about programming the QUICKPEL.

We assume the reader has written some assembler code, though not necessarily for the 8088. Hence we will elaborate on features of the 8088 assembler and architecture which are common knowledge to those who have programmed this CPU, but which would seem odd or cause difficulty to those who have only programmed more orthogonal architectures.

2. A simple standalone program.

First we examine a simple coprocessor program in some detail. The program shown in Figure 1 flips the video display about the central horizontal and vertical axes; thus, the pixel in the top righthand corner is moved to the lower left, the top lefthand corner is moved to the lower right, and all other pixels are moved accordingly. This program can be used to invert a NAPLPS page or any other graphics that is currently being displayed.

```
ts          equ       04h
ec_stop     equ       026h
tele_ram    equ       0c000h
video_ram   equ       010a0h

telesoft    segment   para public
            assume    cs : telesoft, ds : telesoft, ss : telesoft
```

The program begins with definition of important constants. *tele_ram* and *video_ram* identify the start of the program segment and the start of the display RAM, † respectively. The pixels of the display are accessed by reading and writing a contiguous 210 x 256 block of bytes in RAM. The lower nibble of each byte contains the value of the colour map used to display the corresponding pixel. The upper nibble is ignored on writing and is returned as zero on reading. *ec_stop* is the name of an ETEX system call which stops a task. In this case, the program uses *ec_stop* to halt itself.

8088 assembler programs exist in segments of size 64K. Since the QUICKPEL board has only 16K RAM for telesoftware programs, all programs and subroutines will fit in one segment, which we will refer to as *telesoft* for all programs in this document. The segment definition is *para* to indicate that the segment should start on a paragraph boundary, and *public* to indicate that source for the segment need not be completely contained in this file. In this case the source *is* contained within the file, but most of the programs in this document employ subroutines which are in different files. The *assume* statement is an assembler pseudo-op; it produces no code but instructs the assembler to assume that certain registers

† Video ram actually begins at 01000h, but we ignore the first ten rows of pixels since they are used for status display.

have given values, thus permitting the assembler to generate the most compact code. However, it is the programmer's responsibility to ensure that what the assembler is told to assume is in fact true. Strange and wondrous bugs will occur if the assembler assumes segment register values which the programmer did not intend.

```
        org     100h
begin:  jmp     start
        dw      64h dup (0h)
stack   dw      0h

start   proc    near
        sti
        mov     ax,tele_ram          ; ds = ss = telesoftware ram
        mov     ds,ax
        mov     ss,ax
        mov     sp,offset stack      ; set up this task's stack area
        mov     ax,video_ram         ; es = video ram
        mov     es,ax

        mov     si,0
        mov     di,256*200
        dec     di
```

The next section is initialization of registers and data storage areas. All telesoftware programs in this document begin at 100h, although other addresses are deemed possible in the manual. The *begin* label is the entry point of the program.

After definition of the entry point, execution jumps over the data definition area, which in this program consists solely of the stack. It is important that stack space be allocated, even if the program never accesses the stack. ETEX automatically uses the stack area of the currently executing task to save context during interrupts, hence tasks must always have stack space available.

The program is defined as a *near* procedure, which means that all jumps will be within the same segment. Interrupts should be enabled, since the task is started with interrupts disabled (otherwise the processor will never be able to respond to any more data from the PC). The segment registers are initialized to the values indicated in the *assume* pseudo-ops. The extra segment register **es** is set to the start of video ram; since segment registers can't be loaded directly, we must use **ax** as an intermediate. It is our convention to set **es** to video ram and **ds** to telesoftware ram in the main procedure of the programs in this document. Next we initialize the registers to be used in the main loop. The index register **si** is given the offset of the first byte or pixel in video ram, and **di** is given the offset of the last pixel in video ram. The low-order nibble of these pixels contains the colour map entry for the pixel, so by switching corresponding pixels we can invert the image.

```
row:    mov     cl,es:[si]              ; swap the pixels
        mov     al,es:[di]
        mov     es:byte ptr [si],al
        mov     es:byte ptr [di],cl
        dec     di
        inc     si
        cmp     di,si
        ja      row
```

In the main loop of the procedure, the pixels pointed to by **si** and **di** are swapped. This is done by moves in and out of the **al** and **cl** registers. Note that the **es** register is used to override the segments the assembler would normally assume (i.e., **es** for the **di** register, **ds** for the **si** register). After the swap the **di** and **si** registers are adjusted to point to pixels one step closer to the middle of the display; the process is repeated until **si** equals **di**.

```
                mov     bx,ts
                int     ec_stop

start           endp
telesoft        ends
                end     begin
```

The coprocessor program is halted by invoking *ec_stop* with the task number in the **bx** register. If the program doesn't halt itself, it may be difficult or impossible to halt it, to communicate with the other tasks, or to load a new telesoftware program. The last statements conclude the definition of the procedure. Note that the *end begin* statement is the means by which the assembler determines that the statement labelled *begin* is the entry point of the program, so it is important that this be specified as shown.

There are several important points to note about the linking and loading of this program. Executable files can be either .EXE files or .COM files, where .EXE files use multiple segments and .COM files use a single segment. See sections 3-6 to 3-22 of the *IBM Macro Assembler* manual [5] for more details about .EXE and .COM files. Though our programs will fit in one segment, the coprocessor loader requires that the program be in .EXE format. Further, we must ignore the *IBM Macro Assembler* manual's exhortation to define a segment of type *stack*, as is stated on page 13 of the *Programmer's Guide*.[3] Presumably this is because ETEX would ignore the *stack* segment that the programmer defines, preferring instead to assume that the programmer has set up **ss:sp** to provide enough stack space for all subroutine calls. Note also that the the .EXE file need not be exe2bin'd, as suggested on page 13 of the *Programmer's Guide*, nor does it need the extension .COP.

Mirror can be loaded with the LCS program, or the programmer may wish to look at Appendix 2 for source code that should help him to write his own loader.

3. Subroutines.

Programming efficiency is significantly improved by creating modules that can be used in many programs.

3.1. Blit.

A simple task that most programs perform is stopping themselves. Figure 2 shows the procedure *Stop* which halts its invoking program.†

```
ts          equ         04h
ec_stop     equ         026h

telesoft    segment     public
            assume      cs : telesoft
            public      Stop

Stop        proc        near
            mov         bx,ts
            int         ec_stop
            ret

Stop        endp
telesoft    ends
            end
```

Figure 2. *Stop.*

The main points to note about *Stop* are that the *end* statement does not refer to a label, a public entry point is defined, a single *assume* statement is used, and *ret* is used at the end of the procedure to return to the calling procedure. The definition of a public label permits other programs to use this subroutine; note that the subroutine is in segment *telesoft* and is not the only source code in this segment.

The proper use of *Stop* is seen in *Shrink*, shown in Figure 3. *Shrink* is a program that produces a 1/4 size replica of the current display in the upper right hand corner. It does this by taking every other pixel on every other row and moving it to the appropriate position in the upper right hand corner of the display.

---

† More accurately, *Stop* will halt the task numbered ts. This will be task number for every program in this tutorial. Programmers should consult the *Programmer's Guide* if they feel brave enough to write multiple-task applications.

```
video_ram   equ      010a0h
tele_ram    equ      0c000h

telesoft    segment  para
            assume   cs : telesoft, ds : telesoft, es : telesoft, ss : telesoft
            extrn    Stop : near

            org      100h
begin:      jmp      start
            dw       64h dup (0h)
stack       dw       0h

start       proc     near
            sti                          ; IRQs must be on
            mov      ax,tele_ram         ; ds = ss = telesoftware ram
            mov      ds,ax
            mov      ss,ax
            mov      sp,offset stack     ; set up this task's stack area
            mov      ax,video_ram        ; es = video memory
            mov      es,ax
            mov      ds,ax               ; ds = video memory

            mov      di,256*200          ; counter initialised
            dec      di
            mov      si,di
            dec      si
            mov      cx,128
            mov      bx,200

            std
 again:     movsb                        ; move a byte
            dec      si                  ; skip a pixel
            dec      cx                  ; if not done half one row, continue
            jnz      again
            sub      di,128
            mov      cx,128
            sub      si,256
            dec      bx                  ; skip a pixel
            dec      bx
            jnz      again

done:       call     Stop                ; stop myself
start       endp
telesoft    ends
            end      begin
```

Figure 3. *Shrink.*

```
        std
again:  movsb             ; move a byte
        dec      si       ; skip a pixel
        dec      cx       ; if not done half one row, continue
        jnz      again
        sub      di,128
        mov      cx,128
        sub      si,256
        dec      bx       ; skip a row
        dec      bx
        jnz      again
```

The key part of *Shrink* is the above loop. **di** points to the current pixel of the shrunken display; **si** points to the current pixel of the display being shrunk. **cx** keeps track of whether a row is finished, while **bx** keeps track of whether the whole of video ram has been shrunk.

es and **ds** had been set to point to the start of video ram before this loop (see Figure 3), while **di** and **si** are offsets to the end of video ram. *movsb* automatically moves the byte pointed to by **ds:si** to **es:di** and decrements both **si** and **di** (*std* specifies decrement; if we had used *cld* then **si** and **di** would have been incremented). We *dec* **si** to skip a pixel and then continue. The rest of the statements in the loop merely check for the end of a row and move to every other row.

3.2. Blit.

A more complex subroutine is *Blit*, which copies one rectangular section of video memory to another. *Blit* requires as parameters the coordinates of the top left corner of the source rectangle, the coordinates of the top left corner of the destination rectangle, and the height and width of the rectangle. The coordinates of the top left corner of the source and destination rectangles are passed in the **bx** and **ax** registers, respectively. The **dx** register contains the height and width of the rectangle with the height in **dh** and the width in **dl**. The coordinates are passed as 16-bit quantities with the y coordinate in the high order byte and the x coordinate in the low order byte. This arrangement is possible because one byte contains just enough discrimination for the 256 pixels along the horizontal axis of the video display. Incrementing the high order byte corresponds to incrementing a row in video memory. The code for *Blit* is shown in Figure 4.

Blit saves the registers it modifies, as subroutines should generally do (unless there are significant costs in doing so). In order to copy data from one section of video ram to another, both the **ds** and the **es** must point to video ram. *Blit* adjusts **ds** to point to video ram.

The main reason for adjusting **ds** is so that a *rep movsb* instruction can be employed. *rep movsb* is a very fast means for moving blocks of data, but requires that the source segment be given by **ds**, the source

```
video_ram   equ       010a0h

telesoft    segment   public
            assume    cs : telesoft
            public    Blit

Blit        proc      near
            push      cx
            push      si
            push      di
            push      ds

            mov       cx,video_ram    ; make data segment point
            mov       ds,cx           ; to video memory

onerow:     mov       si,bx           ; source address
            mov       di,ax           ; destination address
            mov       ch,0
            mov       cl,dl           ; number of columns

            rep       movsb           ; copy one row

            dec       ah              ; go down one row in both
            dec       bh              ; source and destination
            dec       dh              ; decrement row counter
            cmp       dh,0
            jne       onerow          ; do another row

            pop       ds
            pop       di
            pop       si
            pop       cx
            ret

Blit        endp
telesoft    ends
            end
```

Figure 4. *Blit.*

index by **si**, the destination segment by **es**, the destination index by **di**, and the size of the block by **cx**. If these five registers are properly initialized (as in *Blit*), *rep movsb* will automatically move a byte from **ds:si** to **es:di**, increment **si** and **di**,† and decrement **cx**, repeating this until **cx** is zero. It is **very** important to keep track of the **ds** and **es** registers, and

† **di** and **si** are incremented if the direction flag is cleared by *cld*; auto-decrement is used if the the direction flag is set with *std*.

to know which assembler instructions use them by default, either alone or with the **si** and **di** registers. Ignorance of these architectural peculiarities can be the source of inexplicable behaviour.

After initialization, each row is copied and the registers are re-set for the next row. After all the rows have been copied to the new location, the registers which were altered during the execution of the routine are restored to their original values.

3.3. Scroll.

Scroll scrolls rows of pixels in a specified rectangle or window on the screen. Scrolling is achieved by moving each row to the one above it; the top row is moved to a temporary location and then moved to the bottom before termination. As in *Blit*, the coordinates of the top left hand corner of the window to scroll are passed in the **bx** register, and the height and width of the window are passed in the **dx** register. *Scroll* is shown in Figure 5.

Scroll first saves the working registers and then stores the passed parameters in local variables. The **es** and **ds** registers must be exchanged so that the top line of the window can be copied and saved to a temporary location in telesoftware ram. Alternatively, the top line of the window could have been saved in a part of video memory which is not visible on the display device.

After the first line has been moved, the main loop moves each line to cover the one above it. This requires resetting the **es** register to point to video ram, and setting **di** and **si** for each row. After all the lines have been moved up, the temporarily-stored top line is copied to the bottom row, requiring **ds** to be set to telesoftware ram again. Finally, the original values of the registers are restored.

Blit and *Scroll* are used in the program in Figure 6 to copy a section of video ram to another area and start both the source and destination windows scrolling continuously. This is not a terribly useful program, but it illustrates again how to incorporate subroutines with a main program in the QUICKPEL environment.

```
video_ram       equ       010a0h
tele_ram        equ       0c000h

telesoft        segment   public
                assume    cs : telesoft
                public    Scroll
window_width    db        ?
window_height   db        ?
xpos            db        ?
ypos            db        ?
xout            db        ?
yout            db        ?
count           db        ?
temp_line       db        256 dup (?)

Scroll          proc      near
                push      cx                    ; save the working registers
                push      si
                push      di
                push      ds

                mov       window_width,dl       ; save the top line
                mov       window_height,dh
                mov       ypos,bh               ; set the variables
                mov       xpos,bl
                mov       yout,bh
                mov       xout,bl
                mov       dx,video_ram          ; make data segment point
                mov       ds,dx                 ; to video memory
                mov       dx,cs                 ; es needs to point to data
                mov       es,dx

                mov       si,bx
                lea       di,temp_line
                mov       ch,0
                mov       cl,window_width
                rep       movsb                 ; move to the temp line...
                mov       count,1               ; set row counter
                dec       ypos
                mov       dx,video_ram
                mov       es,dx                 ; set es to same as ds

loop:           mov       dh,yout
                mov       dl,xout
                mov       di,dx
                mov       dh,ypos
                mov       dl,xpos
                mov       si,dx
```

```
            mov     ch,0
            mov     cl,window_width
            rep     movsb                   ; move a row
            inc     count                   ; if not last row, do another
            mov     al,window_height
            cmp     count,al
            je      last_row
            dec     ypos
            dec     yout
            jmp     loop

last_row:   mov     dh,ypos
            mov     dl,xpos
            mov     di,dx
            lea     si,temp_line
            mov     ch,0
            mov     cl,window_width
            mov     dx,tele_ram             ; reset ds
            mov     ds,dx
            rep     movsb                   ; move from the temp line...

            pop     ds                      ; restore the working registers
            pop     di
            pop     si
            pop     cx
            ret

Scroll      endp
telesoft    ends
```

Figure 5. *Scroll.*

```
video_ram   equ       010a0h
tele_ram    equ       0c000h
ht_width    equ       03232h
source      equ       05032h              ; coords for source window
dest        equ       0a032h              ; coords for destination window

telesoft    segment   para public
            assume    cs : telesoft, ds : telesoft, es : telesoft, ss : telesoft
            extrn     Blit    : near
            extrn     Scroll  : near
            org       100h
start:      jmp       begin
            db        64h  dup (0h)
stack       dw        9090h

begin:      sti                           ; IRQs must be on
            mov       ax,tele_ram         ; ds = ss = telesoftware ram
            mov       ss,ax
            mov       sp,offset stack     ; set up this task's stack area.
            mov       ds,ax
            mov       ax,video_ram        ; es = video ram
            mov       es,ax

            mov       bx,source           ; copy one window
            mov       dx,ht_width
            mov       ax,dest
            call      Blit

loop:       mov       bx,source           ; scroll windows forever
            mov       dx,ht_width
            call      Scroll

            mov       bx,dest
            mov       dx,ht_width
            call      Scroll

            jmp       loop

telesoft    ends
            end       start
```

Figure 6. Program using *Blit* and *Scroll*.

4. Communications.

The next type of coprocessor program we examine is one which transfers data between the QUICKPEL and the PC. The QUICKPEL board contains an 8255 communications chip which presents four ports to the PC: a two-way serial port, a one-way keyboard port, a read-only status port, and a write-only control port. Most of the communications with the board is done through the two-way port; the keyboard port is reserved for slow data such as might be transmitted by a typist. In Section 5 we present a program which uses the keyboard port. The control port can be written on to enable or disable interrupts, and the status port can be read to determine the status of the board. More information on these ports can be found on pages 8-11 and page 30 of the *Technical Reference Manual.*[4]

```
session_in   equ       01h
spare        equ       0ah
ec_switch    equ       024h

telesoft     segment   public
             assume    cs : telesoft
             public    Init_Rx

Init_Rx      proc      near
             push      ax
             push      bx
             mov       ax,spare
             mov       bx,session_in
             int       ec_switch        ; redirect Rx port
             pop       bx
             pop       ax
             ret

Init_Rx      endp
telesoft     ends
             end
```

Figure 7. *Init_Rx.*

These physical ports are mapped to logical ports by ETEX. In order to communicate, a coprocessor program must first logically switch the required port so that its input or output is appropriately directed. If the port is not switched, data that is intended for the coprocessor program will be sent elsewhere (probably to the PLPS task). For example, in order to read data from the serial I/O port, the routine *Init_Rx* in Figure 7 must first be invoked.

```
ec_recv      equ       022h
ec_wtrecv    equ       023h
spare        equ       0ah

telesoft     segment   public
             assume    cs : telesoft
             public    Rx

Rx           proc      near
             push      bx
             push      dx
loop:        mov       bx,spare
             int       ec_recv        ; get a byte from Rx process
             cmp       dh,1
             je        done
             mov       bx,0400h       ; can't? then wait
             int       ec_wtrecv
             jmp       loop
done:        pop       dx
             pop       bx
             ret

Rx           endp
telesoft     ends
             end
```

Figure 8. *Rx*.

Data transmitted from the PC to the serial I/O port is accepted by the *Session_in* task, which writes the data on the logical port *session_in*; by means of the *ec_switch* call the data will be switched to the port *spare*. The port numbers are given on page 10 of the *Technical Reference Manual.*[4]

If the port has been properly switched, a data byte can be accepted by invoking *Rx* as shown in Figure 8. *Rx* invokes *ec_recv*, looking for a byte. If no byte is found, *Rx* invokes *ec_wtrecv* to wait for one. This is repeated until a byte is found; the returned byte is passed back in **al**.

Transmitting data from the QUICKPEL to the PC occurs in much the same fashion, except that it is not necessary to switch the logical port for transmission. Hence bytes can be sent by simply calling *Tx* as shown in Figure 9.

The *Rx* procedure will work for most simple ASCII data, but will not receive raw binary data. This is because certain bytes are interpreted by the *Session_in* task as requests to do flow control, start telesoftware loading, etc. In order to pass raw binary data to a telesoftware program we must encode any bytes which could be intercepted by *Session_in*; this is done by *BinRx* as seen in Figure 10.

```
ec_send       equ       020h
ec_wtsend     equ       021h
session_out   equ       05h

telesoft      segment   public
              assume    cs : telesoft
              public    Tx

Tx            proc      near
              push      bx
              push      dx
send:         mov       bx,session_out
              int       ec_send          ; try to send to Tx process
              cmp       dh,1
              je        done
              mov       bx,0020h         ; can't? then wait
              int       ec_wtsend
              jmp       send
done:         pop       dx
              pop       bx
              ret

Tx            endp
telesoft      ends
              end
```

Figure 9. *Tx*.

Special bytes interpreted by *Session_in* have a value less than $20; hence the core of our binary transparency transmission is is to flag all such bytes by sending first an SOH ($01), then adding $20 to the byte and sending it. *BinRx* looks for an SOH, and then subtracts $20 from the subsequent byte; otherwise it simply passes the byte on.

The "correct" (and suggested by Electrohome) flag character to employ is Data Link Escape ($10). However, in our QUICKPEL boards this is not completely safe, and the odd escaped data byte still appears to be interpreted by *Session_in*, even when $20 has been added. We have experienced no problems with using SOH as the flag byte.

Two useful programs which employ these communications procedures are *Load* and *UnLoad*, which transfer pixels between video ram and files on the PC. *UnLoad* takes a "snapshot" of the contents of video ram and sends it to a program on the PC, which can store this information in a file. The "snapshot" can be redisplayed by sending it to *Load*.

There are 51200 pixels in the display area of video ram (excluding the status line). Sending each of these individually would require more than a minute of data transfer time, plus a significant amount of storage space on the PC's hard disk. In order to make *Load* and *UnLoad* more efficient,

```
ec_recv      equ       022h
ec_wtrecv    equ       023h
spare        equ       0ah

telesoft     segment   public
             assume    cs : telesoft
             public    BinRx
             extrn     Rx : near

BinRx        proc      near
             call      Rx           ; get a byte
             cmp       al,01h       ; if not SOH, done
             jne       done
             call      Rx           ; else get the real data byte
             sub       al,20h
done:        ret

BinRx        endp
telesoft     ends
             end
```

Figure 10. *BinRx*.

```
telesoft     segment   public
             assume    cs : telesoft
             public    BlockTx
             extrn     Tx : near

BlockTx      proc      near
             or        al,0f0h      ; don't send a zero byte
             mov       ah,0         ; send a block
             call      Tx
             mov       al,cl
             mov       ah,0
             call      Tx
done:        ret

BlockTx      endp
telesoft     ends
             end
```

Figure 11. *BlockTx*.

```
ts          equ       04h
ec_resume   equ       02ch

telesoft    segment   public
            assume    cs : telesoft
            public    Synch
            extrn

Synch       proc      near
            call      Init_Rx      ; set up for synchronization
            mov       bx,ts
            int       ec_resume    ; start myself
            mov       al,0
            call      Tx           ; synchronize : send value
            call      Rx           ; get value
            ret

Synch       endp
telesoft    ends
            end
```

Figure 12. *Synch*.

we use run-length encoding to reduce the number of bytes to be sent. Run-length encoding reduces data transmission by sending only two bytes to describe a block of consecutive pixels with the same value. A pair of bytes will completely describe the block; the first byte gives the value of the pixel, and the second byte gives the number of pixels in the block. Since a byte cannot contain a value larger than 255, large blocks must be sent as multiples of 255.

UnLoad is shown in Figure 13; it invokes the new procedures *TxBlock* and *Synch*, which are shown in Figures 11 and 12, respectively. *Synch* is designed to overcome a bug in the QUICKPEL which causes an indeterminate delay between the startup of a task and the time when it is able to receive data (this bug is documented on page 21 of the *Technical Reference Manual*). *Synch* merely exchanges a zero byte with the PC program in order to initialize communications. *Synch* invokes *Init_Rx*, which frees us from having to do this in the main procedure.

UnLoad's main activity is to look for blocks of consecutive pixels of the same value. When it has identified such a block, or when the maximum block size of 255 has been reached, it calls *TxBlock* with the value of the pixel in **al** and the number of bytes in the block in **cl**. *TxBlock* then sends these two bytes to the PC program waiting to receive the data. At the end of video ram a single zero is sent to indicate the end of the data. Note that *TxBlock* masks the upper nibble of the value block so that a zero byte is not sent by accident (i.e., when the value of the pixel is zero).

```
video_ram   equ       010a0h
tele_ram    equ       0c000h

telesoft    segment   para public
            assume    cs : telesoft, ds : telesoft, es : telesoft, ss : telesoft
            extrn     Tx : near, BlockTx : near, Synch : near, Stop : near
            org       100h
begin:      jmp       start
            db        64 dup (0h)
stack       dw        0h

start       proc      near
            sti                             ; IRQs must be on
            mov       ax,tele_ram           ; ss = telesoftware ram
            mov       ds,ax                 ; ds = telesoftware ram
            mov       ss,ax
            mov       sp,offset stack       ; set up this task's stack area
            mov       ax,video_ram          ; es = video memory
            mov       es,ax
            call      Synch                 ; synchronize with PC program

            mov       di,256*200            ; counter initialised
            dec       di
            mov       al, es:byte ptr [di]
            mov       cx,0

loop:       dec       di
            mov       ah, es:byte ptr [di]
            inc       cx
            cmp       di,0h
            jz        done                  ; done all video ram?
            cmp       cx,255                ; maximum block size?
            jl        next                  ; if not, keep going
            call      BlockTx               ; otherwise send a block
            mov       al,es : byte ptr [di]
            mov       cx,0
            jmp       loop

next:       cmp       ah,al                 ; different pixel?
            je        loop                  ; if not, keep going
            call      BlockTx               ; otherwise send a block
            mov       al, es:byte ptr [di]
            mov       cx,0
            jmp       loop

done:       call      BlockTx
            mov       cx,1
            mov       al, es:byte ptr [di]
```

```
            call      BlockTx
            mov       al,0h                  ; indicate end of video ram
            call      Tx
            call      Stop                   ; stop myself

start       endp
telesoft    ends
            end       begin
```

Figure 13. *UnLoad.*

Load accepts run-length encoded data from a PC program and sets the appropriate nibbles in video ram. *Load* is shown in Figure 14.

Depending on the complexity of the graphics to be displayed, *Load* can require less than half the time of execution of the NAPLPS code; generally the storage space required for the display is comparable to the space required for NAPLPS.

```
video_ram   equ       010a0h
tele_ram    equ       0c000h

telesoft    segment   para public
            assume    cs : telesoft, ds : telesoft, es : telesoft, ss : telesoft
            extrn     Synch : near, Stop : near, BinRx : near
            org       100h

begin:      jmp       start
            dw        64 dup (0h)
stack       dw        0h

start       proc      near
            sti                         ; IRQs must be on
            mov       ax,tele_ram       ; ds = ss = telesoftware ram
            mov       ds,ax
            mov       ss,ax
            mov       sp,offset stack   ; set up this task's stack area
            mov       ax,video_ram      ; es = video memory
            mov       es,ax
            call      Synch             ; synchronize with PC program
            mov       di,256*200        ; counter initialised
            de        di

            std
again:      call      BinRx             ; get a byte
            cmp       al,0              ; null indicates eof
            je        done
            mov       bl,al             ; first byte is value
            call      BinRx
            mov       cl,al             ; second byte is size
            mov       al,bl
            rep       stosb
            jmp       again

done:       call      Stop              ; stop myself

start       endp
telesoft    ends
end         begin
```

Figure 14. *Load.*

5. A cursor managing program.

As the last application in this tutorial we present *Cursor*, a program used to manipulate a custom cursor. A custom cursor is desirable for several reasons, including the slow speed of the cursors supplied by NAPLPS, the inability to change the standard cursor's shape or colour, and the fact that NAPLPS code affects the position of the cursor and thus complicates cursor control.

Cursor uses the keyboard port of the QUICKPEL, which is a one-way port (PC to QUICKPEL). The advantage of using this port is that it is normally ignored by the PLPS task, and hence cursor input is automatically disambiguated from other input. The disadvantage of using this port is that it doesn't have flow control, and hence it possible to lose bytes. This has not been a problem for us because of the low data rate common to a mouse-driven cursor.

```
kybd_in      equ      03h
ts_kybd_in   equ      08h
ec_switch    equ      024h

telesoft     segment  public
             assume   cs : telesoft
             public   Init_Ky

Init_Ky      proc     near
             mov      ax,ts_kybd_in
             mov      bx,kybd_in
             int      ec_switch        ; redirect Ky port
             ret

Init_Ky      endp
telesoft     ends
             end
```

Figure 15. *Init_Ky*.

In order to use the keyboard port we must observe two conditions. First, it is necessary to turn off local echo so that bytes sent to the board are not displayed on the status line. If local echo is used then other NAPLPS conditions become important, such as the definition of protected fields. Second, certain byte sequences are interpreted as local function requests, and hence the PC program must be careful not to send these sequences inadvertently. A list of the sequences which invoke local functions is given on page 3 of the *Technical Reference*; bytes in the range $0-$7f should not be interpreted as local function requests, so we will confine our program to these bytes.

The keyboard port must be switched from the PLPS task so that the coprocessor program receives the keyboard port data (as was done for *Rx* by *Init_Rx*). Keyboard port initialization is performed by *Init_Ky* in Figure 15.

After the port has been switched, *Ky* can be invoked. *Ky* is a straightforward modification of *Rx* and is seen in Figure 16.

```
ec_recv     equu      022h
ec_wtrecv   equ       023h
ts_kybd     equ       08h

telesoft    segment   public
            assume    cs : telesoft
            public    Ky

Ky          proc      near
            push      bx
            push      dx
loop:       mov       bx,ts_kybd
            int       ec_recv          ; get a byte from kybd
            cmp       dh,1
            je        done
            mov       bx,0100h         ; can't? then wait
            int       ec_wtrecv
            jmp       loop
done:       pop       dx
            pop       bx
            ret

Ky          endp
telesoft    ends
            end
```

Figure 16. *Ky*.

Our main routine *Cursor* is shown in Figure 17. *Cursor* repeatedly calls *Ky* to obtain a byte and then passes the byte to *Test_in* (shown in Figure 18) to determine the new position of the cursor.

Bytes sent to *Cursor* (and passed to *Test_in*) indicate the direction of movement in the upper nibble and the displacement in the lower nibble. The PC program controlling the cursor must also send a *cursor off* command before any NAPLPS code is sent to the board; this ensures that the cursor is invisible when drawing operations are being executed. After all the NAPLPS code has been sent, *Test_in* invokes *ec_resume* on the PLPS task to allow it to finish executing any pending NAPLPS code; finally, the cursor is redrawn in its current position. The external variable *xor_count*

```
video_ram   equ        10a0h
tele_ram    equ        0c000h

telesoft    segment    para public
            assume     cs : telesoft, ds : telesoft, es : telesoft, ss : telesoft
            extrn      Ky : near, Init_Ky : near, Test_in : near

            org        100h
start:      jmp        begin
            db         64h               dup (0h)
stack       dw         9090h

begin:      sti                          ; IRQs must be on
            mov        ax,tele_ram       ; ds = ss = telesoftware ram
            mov        ss,ax
            mov        sp,offset stack   ; set up this tasks stack area.
            mov        ds,ax
            mov        ax,video_ram      ; es = video ram
            mov        es,ax             ; NOTE: ignore the status lines

            call       Init_Ky

get_next:   call       Ky                ; pass input bytes to Test_in
            call       Test_in
            jmp        get_next

telesoft    ends
            end        start
```

Figure 17. *Cursor.*

is used to keep track of the number of times to XOR the cursor (twice to move it, once to either hide or display it). The external variables *xout* and *yout* contain the new position of the cursor, and are accessed by the subroutine *Draw_cur*.

The subroutine *Draw_cur* produces an XOR cursor at the position indicated by *xout* and *yout*. It does this by XORing the pixels that define the cursor; when the cursor is moved, the pixels are restored to their original value by XORing again. The cursor is defined as a contiguous set of bytes 240 bytes in the shape of a 12 x 20 rectangle. In Figure 19 the cursor shape is a hand with a pointing finger; because the background is defined as zero, only the hand will show on the display (XOR of a pixel with 0 produces no change). The pointing finger lies on the middle of the screen at (125,100). If the public variable *xor_count* is set to 3, then the cursor is moved to the new position indicated by xout and yout. If *xor_count* is set to 2, then the cursor is XORed once (i.e., toggled) in place.

```
north       equ       80h
south       equ       40h
east        equ       20h
west        equ       10h
cur_on      equ       70h
cur_off     equ       90h
ec_resume   equ       02ch
naplps      equ       08h

telesoft    segment   public
            assume    cs : telesoft, ds : nothing, es : nothing, ss : nothing
            extrn     Draw_cur : near, xout : byte, yout : byte, xor_count : byte
            public    Test_in
            dir       db              0

Test_in     proc      near
            mov       dir,al
            and       dir,11110000b   ; upper nibble indicates direction
            and       al,00001111b    ; lower nibble indicates displacement

            cmp       dir,north
            jne       test_south
            add       yout,al
            jmp       draw

test_south: cmp       dir,south
            jne       test_east
            sub       yout,al
            jmp       draw

test_east:  cmp       dir,east
            jne       test_west
            add       xout,al
            jmp       draw

test_west:  cmp       dir,west
            jne       test_on
            sub       xout,al
            jmp       draw

test_on:    cmp       dir,cur_on
            jne       test_off
            mov       bx,naplps
            int       ec_resume       ; allow PLPS to finish
            mov       xor_count,2
            call      Draw_cur
            jmp       return
```

```
test_off:   cmp     dir,cur_off
            jne     return
            mov     xor_count,2
            call    Draw_cur
            jmp     return

draw:       mov     xor_count,3      ; move the cursor
            call    Draw_cur

return:     ret
Test_in     endp
telesoft    ends
            end
```

Figure 18. *Test_in*.

```
video_ram       equ       10a0h
cursor_width    equ       20
cursor_height   equ       12

telesoft        segment   public
                assume    cs : telesoft, ds : nothing, es : nothing, ss : nothing
                public    xor_count, xout, yout, Draw_cur

cursor          db        00,00,00,00,00,00,00,15,15,15,15,15,00,00,00,00,00,00,00,00
                db        00,00,00,00,00,00,15,15,15,15,15,15,15,00,00,00,00,00,00,00
                db        00,00,00,00,00,15,15,15,15,15,00,00,00,00,00,00,00,00,00,00
                db        00,00,00,00,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15
                db        15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15
                db        15,15,15,15,15,15,15,15,15,15,15,15,15,15,00,00,00,00,00,00
                db        15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,00,00,00,00
                db        15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,00,00,00,00
                db        15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,00,00,00,00,00
                db        15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,00,00,00,00,00
                db        15,15,15,15,15,15,15,15,15,15,15,15,00,00,00,00,00,00,00,00
                db        00,00,00,00,00,15,15,15,15,15,15,15,00,00,00,00,00,00,00,00
count           db        0
xor_count       db        ?
xout            db        124
yout            db        87
xpos            db        124
ypos            db        87

Draw_cur        proc      near
repeat:         dec       xor_count              ; outer loop. Do two XOR's
                cmp       xor_count,0
                je        return
                cmp       xor_count,2
                je        first
                mov       al,xout                ; ah = y coord
                mov       ah,yout                ; al = x coord
                mov       ypos,ah
                mov       xpos,al                ; reset xpos,ypos
                jmp       xor_it
first:          mov       al,xpos
                mov       ah,ypos

xor_it:         mov       si,offset count-1
                mov       count,0h               ; set row counter

loop:           mov       dh,ah
                mov       dl,al
                mov       di,dx
```

```
             mov    cx,cursor_width
once_more:   mov    bh,ds:[si]                ; XOR the pixels
             xor    es:byte ptr [di],bh       ; for the cursor
             inc    di
             dec    si
             dec    cx
             jne    once_more

             inc    count                     ; if not last row, do another
             cmp    count,cursor_height
             je     repeat
             inc    ah
             jmp    loop

return:      ret
Draw_cur     endp
telesoft     ends
             end
```

Figure 19 *Draw_cur*.

6. Acknowledgements.

We would like to thank Frank Tompa for providing the hardware and permitting us to play with the QUICKPEL when we really should have been doing something else. Steve Williams implemented coprocessor software which helped us test several of our programs and proved quite useful in its own right. Finally, thanks to Chris Howlett of Co-Triple who revealed the magic byte $c4, which makes everything possible.

7. Appendix 1.

This section contains a short list of "gotchas" to watch out for when programming the QUICKPEL. If you write a program and it doesn't seem to work for some unfathomable reason, you might try thinking about some of the problems in this list.

- *the NAPLPS what-are-you-waiting-for gotcha* — since NAPLPS code is treated as a stream of data, the PLPS task will not immediately display the result of an instruction. Instead, it waits until it gets the next instruction before completing execution of the previous sequence, as described on page 26 of the CSA T500 standard.[1] Hence if you send the code to draw a circle you will not see a circle until at least one more byte has been sent. One way to do this is to set the drawing point to its current position after each sequence of instructions.

- *the missing-bytes gotcha* — this can occur in at least two forms. First, rapid data transfer to the serial port without paying attention to the XON/XOFF protocol will often result in overflow of the 2K buffer and a consequent loss of bytes. Secondly, rapid data transfer to the keyboard port is just not recommended, since there is no flow control to this port and we don't know how big the buffer is.

- *the segment-register gotcha* — some assembler programs are so simple that there is no good reason for them not to work — except that the segment registers don't contain the values you thought they did. If the program starts space-walking, and especially if weird tartan patterns are showing up on the display, it's likely that the segment registers **es** or **ds** have been set to some unexpected value.

- *the assume* gotcha — this is closely related to the *segment-register* gotcha. You were careful to avoid telling the assembler to *assume* anything. Surprise, it *assume*s things by default anyway. You must specifically indicate *assume* **xs** *: nothing* for segment register **x** if you want to be sure it isn't used.

- *the .EXE gotcha* — coprocessor programs must be linked together to produce .EXE files, not .COM files. See 3-6 to 3-21 of the *IBM Macro Assembler* [5] manual for details about the distinction between .EXE and .COM files. Ignore the documentation on page 10 of the *Programmer's Guide*[3] which tells you to exe2bin the .EXE file, and just load it directly with starting address at 0100h.

8. Appendix 2.

This appendix contains a set of programs and utilities that make up a simple driver package for the QUICKPEL. The programs are written in PORT, a language very similar to C. Translation to C or your favorite applications program should be simple.

The basic component of the driver is *Send*, which writes a single byte to the QUICKPEL's main I/O port. *Send* tests to ensure that the QUICKPEL is ready for another byte, and then searches for an XOFF which signals if the QUICKPEL's 2K input buffer was full. *Send* uses the PORT primitives *IO_out* and *IO_in*, which invoke 8088 assembler *out* and *in*.

```
\
\  send a byte
\
(ch : unsigned[8])
{
repeat
 if (IO_in($8382) > 127)
   break;
repeat
 if (IO_in($8380) != $13)
   break;
IO_out($8380, ch);
}
```

Figure 20. *Send.*

It is often necessary to reset the QUICKPEL to an initial state. This can be done by *Reset* as seen in Figure 21. *Reset* first sends the magic byte \$c4, which causes the right magic to be invoked. Next, the board itself is given a cold start command, which reinitializes the telesoftware and other internal tasks. Interrupts are disabled, and finally the NAPLPS initialization is invoked.

In order to load raw binary data (such as coprocessor programs) we must send it in binary transparency mode. As described earlier, all bytes greater than or equal to \$20 can be sent unchanged; bytes less than \$20 must first be flagged, and then be added to \$20 before being sent. The function *Send_Bin_Trans* in Figure 22 uses DLE as a flag. Why did we use SOH as a flag character earlier, and DLE now? We find it necessary to use two binary transparency protocols, one for loading coprocessor programs and one for transmitting data to these programs. *Send_Bin_Trans* is used for loading coprocessor programs and uses DLE because the coprocessor loader on the board expects DLE as the flag character. A similar function (which we call *BinSend*) is used to communicate with programs that invoke *BinRx*; this function is identical to *Send_Bin_Trans* except that it uses SOH (\$01) as the flag byte.

Though the QUICKPEL is supplied with a program for loading coprocessor software, it may be desirable or necessary to write your own loader because of differences in the operating system or the need to incorporate the loader in another program. Figure 23 contains the source for *Load_Telesoft*, our loader; its associated functions are found in Figures 24 through 28. *Load_Telesoft* takes a .EXE file as produced by the linker, sends its starting address and size, strips off the first 512 bytes and then sends the program to the QUICKPEL. The program is followed by a command to start the telesoftware.

Other driver routines can be written by following the pattern shown in the loader's associated functions. Most driver utilities are merely three or four *Sends* of the appropriate bytes.

```
\
\ Load coprocessor program
\
import(Data_types, IO_characters, IO_descriptor, IO_modes)
()
file : &char
fp   : &IO_descriptor
size : unsigned
{
Obtain_command_line();
file = Next_arg();
if (fp = Open(file, READ, UNSPECIFIED_TYPE, 0) == 0)
  {
  Printf("File %s doesn't exist.*n", [file]);
  Flush();
  return;
  }
size = Size_Of_File(fp);
Eight_Bit_Mode();
Telesoft_Address(0);
Telesoft_Size(size - 512);
Send_Telesoft(fp, size);
Close(fp);
Telesoft_Start($100);
}
```

Figure 23. *Load_Telesoft.*

```
\
\  send starting address of coprocessor software
\
( start : unsigned[16] )
{
Send($1B);
Send($26);
Send($3A);
Send_Bin_Trans(start >> 8);
Send_Bin_Trans(start & $00ff);
}
```

Figure 24. *Telesoft_Address.*

```
\
\  start coprocessor software
\
( start : unsigned )
{
Send($1B);
Send($26);
Send($3F);
Send_Bin_Trans(start >> 8);
Send_Bin_Trans(start & $00ff);
}
```

Figure 25. *Telesoft_Start.*

```
\
\  send size of coprocessor software
\
( size : unsigned )
{
Send($1B);
Send($26);
Send($3D);
Send_Bin_Trans(size >> 8);
Send_Bin_Trans(size & $00ff);
}
```

Figure 26. *Telesoft_Size.*

```
\
\  set eight bit mode
\
()
{
Send($1B);
Send($23);
Send($32);
}
```

Figure 27. *Eight_Bit_Mode.*

```
\
\     send telesoftware program
\
import(IO_descriptor, Seek_origins)
( fp   : &IO_descriptor
  size : unsigned  )
  i    : unsigned
{
Seek(fp, 512, BEGINNING_OF_FILE);
Select_input(fp);
for (i=512; i<size; ++i)
  Send_Bin_Trans(Get());
}
```

Figure 28. *Send_Telesoft.*

9. References

1. Videotex/Teletext Presentation Level Protocol Syntax, T500-1983, Canadian Standards Association, Rexdale, Ontario (October 3, 1983).
2. *Quickpel User's Guide,* Electrohome Ltd., Kitchener, Ontario (June 1984).
3. *Quickpel Programmer's Guide,* Electrohome Ltd., Kitchener, Ontario (June 1984).
4. *Quickpel Technical Reference Manual,* Electrohome Ltd., Kitchener, Ontario (June 1984).
5. *Macro Assembler Version 2.00,* International Business Machines Corp. (August 1984).