Programming Examples

Brad A. Myers

This manual describes how to use Fonts, Cursors, RasterOp, Line, Windows, CmdParse, PopUp menus, and how to allocate large amounts of memory. The manual includes examples of sample applications.

Three Rivers Computer Corporation
720 Gross Street
Pittsburgh, PA 15224
(412) 621-6250

The information in this document is subject to change without notice and should not be construed as a commitment by Three Rivers Computer Corporation. The Company assumes no responsibility for any errors that may appear in this document.

Three Rivers Computer Corporation will make every effort to keep customers apprised of all documentation changes as quickly as possible. The Reader's Comments card is distributed with this document to request users' critical evaluation to assist us in preparing future documentation.

1. Introduction.

This manual describes how to use the PERQ operating system to perform some interesting operations. Examples are given of the ways we have found to be successful in performing these operations. Although there are obviously many ways to perform these operations, the ones given here are successful.

2. Allocating Memory.

This section describes how to use the Memory module to allocate blocks of memory needed for reading in Fonts and pictures from files, for creating pictures off screen for RasterOp, and for handling large amounts of data.

Fonts and pictures are generally stored in files on the disk. To use the fonts and pictures, they must first be read into memory. First, do a FSLookUp (or one of the other lookup functions) from FileSystem. A VAR parameter to this function is the number of blocks in the file. This number can be passed to the memory manager to tell it how much storage to allocate. Memory on the PERQ is divided into "segments". Each segment can have up to 256 blocks. Each block is 256 words or 512 bytes. When a segment is created, it is given an initial size, a maximum size, and an increment by which to increase the current size when that is not enough. A segment is created by using the procedure from memory:

```
Procedure CreateSegment(
            var Seg: Integer;
            initialSize,              {in blocks}
            sizeIncrement,            {in blocks}
            maximumSize: integer);    {in blocks}
```

where seg is assigned the segment number that has been created. There are two ways to use a segment once created. The first is simply to create it with a fixed size and use the entire segment at once. For example, when reading an entire file into memory. Use MakePtr(seg, offset, TypeOfPointer) to create a pointer of type TypeOfPointer in that segment at word offset "offset". The second way to allocate out of a segment is to use the standard Pascal NEW. NEW has been extended to have two forms. The standard form, NEW(p), allocates the pointer out of the default segment. The extended form, NEW(seg, allignment, p), allocates the storage out of the specified segment. Some buffers need to be specially aligned. For example, RasterOp buffers need to be on a multiple of 4. Do not use 0 for the alignment. For DISPOSE, only the pointer should be specified. NEW is implemented by a call to the procedure NewP in Dynamic. The user can call this procedure directly if he wants to specify the size of storage to allocate. NewP is defined as

```
Procedure NewP(seg: integer;
               allignment: integer;
               var p: MMPointer;
               size: integer);
```

The segment number of 0 is always defined to be the default segment for NewP and NEW. All other segment numbers should come from a prior CreateSegment. To calculate the size of a record or array, WordSize is a useful intrinsic. It returns the size of any PASCAL variable or type and can be used in constant or variable expressions. The user must remember the size used with NewP since DisposeP takes the size as a parameter.

```
Procedure DisposeP(var p: MMPointer; size: integer);
```

The size MUST be the same size used with NewP. One way to insure this is to store the size as a field in a record. As an example of NewP, we make a variable length array of strings:

```
Type
  s25 = String[25];
  NameDesc = RECORD
               numCommands: integer;
               recSize: integer;
               commands: array[1..1] of s25; {vbl length array}
             END;
  pNameDesc = ^NameDesc;
```

To allocate a pNameDesc with NUM names in the segment seg, the following would be done:

```
    var p: MMPointer;
        size: integer;
        names: pNameDesc;
    begin
    size := 2*WordSize(integer) + { for the 2 integers  }
            NUM*WordSize(s25);    { the variable part }
    NewP(seg, 1, p.p, size);
    names := RECAST(p.p, pNameDesc);
    names^.recSize := size;
    names^.numCommands := NUM;
{$R-}  {turn range checking off to assign names}
    for i := 1 to NUM do
       names^.commands[i] := '<some string>';
{$R=}  {return range checking to the previous state}
    end;
```

Since Dynamic uses special places in the segment to store the free list information used by NEW, it is bad practice to mix NEW and MakePtr on the same segment.

When a program requires a large amount of data, consider the swapping characteristics of the operating system. Since POS swaps an entire segment at once, a big segment will take much longer to read in and write out. Also, there may simply not be enough memory to hold the large segment and all other necessary data. Therefore, the user should divide the data into separate segments, each of which is about 10 blocks large. For example, this is what the editor does to hold the piece table.

3. Reading in Large Files.

There are a number of ways to read in a font or a picture from the disk. The fastest and most straightforward way is to use MultiRead. This is a special procedure that uses the micro-code's ability to read multiple blocks at once. The read, therefore, occurs at the maximum possible speed (the actual speed depends on how contiguous the blocks are on the disk).

To use multi-read on a file called FileName do the following:

```
var fid: FileID;  {imported from FileSystem}
    blocks, bits: integer;
    seg: Integer;
begin
fid := FSLookUp(FileName, blocks, bits);
if fid = 0 then {file not found}
else begin
     CreateSegment(seg, blocks, 1, blocks);  {allocate}
     MultiRead(fid, MakePtr(seg, 0, pDirBlk), 0, blocks);
     end;
end;
```

MultiRead takes a fileID, a pointer to the start of the block of memory, the first block to read of the file to read, and the number of blocks. The above code reads in the entire file.

If you do not wish to import MultiRead, you can read in each block of the file using FSBlkRead. Replace the MultiRead call above with the following

```
for i := 0 to blocks - 1 do
   FSBlkRead(fid, i, MakePtr(seg, i*256, pDirBlk));
```

The MakePtr creates a pointer to the i-th block (the i*256-th word) of the segment.

WARNING: The multi-block read exported by FileAccess does not work.

4. RasterOp and Line.

RasterOp and Line are the chief graphics primitives of the PERQ. Each is fast. TThe primitives allow drawing of rectangles and lines, respectively. RasterOp is described in the PERQ Pascal Extensions manual and Line is exported by the Screen module.

Use RasterOp to clear a rectangle (either white or black); transfer a picture from one place to another; or combine two pictures. Use Line to draw a single width line on the screen at any orientation.

RasterOp is a general utility. It can be used on buffers that are not on the screen. Therefore, it takes parameters that describe the dimensions of the buffer. For the Screen, the two constants SScreenW and SScreenP are exported by the Screen module. As a first example, we will clear an area of the screen 100 bits wide, 200 bits tall, starting at position (300, 400):

```
RasterOp(RXor, 100, 200, 300, 400, SScreenW, SScreenP,
                         300, 400, SScreenW, SScreenP);
```

We do this by Xoring the area with itself. Similarly, to clear an area to black, use the function RXNor. The function names are exported by the module Raster. To move a rectange from one area of the screen to another, simply use a different source and destination position. Remember that the destination is specified first.

To move a rectangle one bit down:

```
RasterOp(RRpl, 100, 200, 300, 400, SScreenW, SScreenP,
                         300, 399, SScreenW, SScreenP);
```

The position (0,0) is in the upper left corner; the lower right corner is (767, 1023). RasterOp does not validate the widths or positions so be careful. Be especially careful to avoid negative widths and heights since these are taken as large positive numbers. The available RasterOp functions are:

```
RRpl   {dest get src}
RNot   {dest get invert of src}
RAnd   {dest gets dest AND src}
RNAnd  {dest gets dest AND invert of src}
ROr    {dest gets dest OR src}
RNor   {dest gets dest OR invert of src}
RXor   {dest gets dest XOR src}
RXNor  {dest gets dest XOR invert of src}
```

RasterOp can also move a picture from or to an off-screen buffer. Suppose a picture is 543 bits wide and 632 bits high. The buffers used by RasterOp must be a multiple of 4 words in width. Therefore, allocate a buffer that is 36 words (=576 bits) wide and 632 bits high. This is 22752 words. Since segments can only be allocated on block boundaries, round up to 22784 words or 89 blocks and create a segment of this size and a RasterPtr to its start:

```
CreateSegment(seg, 89, 1, 89);
p := MakePtr(seg, 0, RasterPtr);
```

Now we might read a file into this buffer as described in Section 3. Next, we want to transfer the picture onto the screen, say at position (10, 100). We use

```
RasterOp(RRpl, 543, 632, 10, 100, SScreenW, SScreenP,
                          0,  0,   36, p);
```

The destination (given first) is (10, 100) on the screen, but the source is now the buffer. The bit width to transfer is 543 (the second argument), but the word width of the buffer is 36. (SScreenW is 48, the number of words across the screen). p is the pointer to the buffer. A picture can be transfered from the screen into a buffer, or between buffers in a similar manner.

If you want to allocate a buffer using NEW or NewP for RasterOping to or from, be sure to make the allignment 4.

Line is used for drawing straight, single width lines on the screen. It takes a source and destination x and y position, a style and a pointer to the buffer to draw in. Currently, it can only draw lines in buffers that have width 48 (e.g. the screen). Line is defined as:

```
Line(style: LineStyle; x1, y1, x2, y2: integer;  p: RasterPtr);
```

where the style is DrawLine, XOrLine or EraseLine. Use SScreenP for p.

5. Windows.

POS currently supports multiple, overlapping windows. However, POS does not know when two windows overlap. Thus all windows are "transparent" in that anything written to a covered window will "show through" any windows that are on top. Even with this restriction, windows are useful for a number of applications. For example, if multiple things are going on and the user wants to separate the input and output of each. The Screen package handles scrolling of the text inside windows automatically. Therefore separate windows scroll separately (if they do not overlap). This is useful, for example, in a graphics package where there are commands typed in a small window with the rest of the area used for the graphics (an example is the CursDesign program from the User Library).

The user must maintain the allocation of windows; the user tells the screen package where each window is and is expected to remember the number for each window. Window zero is reserved for the system and its size should not be changed. Use CreateWindow to create a new window. The parameters passed are for the outside of the window. There are two bits of border, then a hair line, then two more bits on each side. On the top there may be a title line which is a band of black with white letters in it. Once a window is created, it cannot be moved or re-signed.

Creating a new window automatically changes output to go to the new window. Given a set of windows, you can change amongst them by using the ChangeWindow command. The procedure GetWindowParms returns parameters of the current window. Unfortuneately, you must do transformations on the numbers returned to get the inside and outside areas of windows:

```
GetWindowParms(var windx: WinRange; orgX, orgY, width, height:
integer;
                    var hasTitle: boolean);
```

windx is the current window number and hasTitle tells whether there is a title line. Calculate the outside of the window as follows:

```
begin
orgX := orgX - 2;
width := width + 5;
orgY := orgY - 2;
height := height + 5;
if hasTitle then
   begin
   orgY := orgY - 15;
   height := height + 15;
   end;
end;
```

Calculate the inside of the window as follows:

```
    begin
    orgX := orgX + 2;
    width := width - 4;
    if hasTitle then
       begin
       orgY := orgY + 2;
       height := height - 4;
       end;
    end;
```

One thing to note: the title line for a window is written in the font that was in effect before the window was created. This font will also be remembered as the font to use the next time the window that was in use before the CreateWindow is used again.

6. Fonts.

The definition of fonts is given in the Screen module. Fonts currently can be variable width, but there is no kerning (the font must fit within the character block). A font starts with some global information: the height of the font in bits and the offset of the baseLine. Next is an array, which for each character has the position and width of that character in the font. A width of zero means the character is not defined. After this array are the actual bit pictures for the characters which are defined. Fonts can be created by using the FontEd program from the User Library available from the Sales department.

To use a font, it must first be loaded into memory. See the section on reading files above. The Screen package allows you to change the font to one you have defined. First, you should define a new window so that you don't change the font for the default system. Now simply call the function SetFont passing it a pointer to the top of the segment into which you read the font. If you wish to RasterOp a character (ch) using font FontP onto the screen by hand (at position (xPos, yPos)), use the following form (copied from SPutChr in Screen):

```
var Trik: Record Case Boolean of
                  true: (F: FontPtr);
                  false: (seg, ofst: integer);
                end;
begin
with FontP^.Index[ord(ch)] do
  if width > 0 then
      begin
      Trik.f := FontP;
      RasterOp(RRpl, width, FontP^.height, xPos,
                    yPos-FontP^.Base, SScreenW, SScreenP,
                    Offset, Line*FontP^.height, SScreenW,
                    MakePtr(Trik.seg, Trik.Ofst+#404, FontPtr));
      end;
 end;
```

The #404 is the size of the introductory part of a font. Trik is used to create a pointer to the actual bit pattern part of a font.

7. Cursors.

Unfortuneately, the term "Cursor" is used in two ways in PERQ-land. First, it is the position where the next character will be placed on the screen. This "cursor" is usually signified by an underline "_". The second "cursor" is the arrow or other picture that usually follows the pen or puck on the tablet. This section discusses the latter form.

The picture in the cursor can be set by the user. POS currently uses a number of different pictures. The default arrow cursor, the "scroll" and "do-it" cursors for PopUp menus, the hand that moves down the side of the screen, and the Busy Bee are all examples of cursors. The program CursDesign from the User Library can be used to create cursors. Once a picture has been created, it can be read into Memory from the file (see above) and then copied into the Cursor. Each cursor is 56 bits wide and 64 bits tall which comes to 4 words wide and 64 bits tall or exactly one block. Therefore a file with one cursor in it can be read in directly into the cursor buffer. The definition of the cursor and all utility procedures for manipulating it are in IO_Others.

```
var curs: CurPatPtr;
begin
New(0,4,curs);
Fid := FSLookup(CursorFile, blks, bits);
FSBlkRead(fid, 0, RECAST(curs, pDirBlk));
end;
```

Note that the cursor buffer must be quad-word aligned (since a RasterOp is done from it by the system). To set a cursor, use the function IOLoadCursor. It takes a CurPatPtr and two integers to tell it the x and y offsets in the cursor from where the cursor is positioned. Thus, for a "bull's eye" cursor where the center is the interesting point, the offsets would be the offsets from the top left of the center. For a right pointing arrow, the offsets would describe the point of the arrow. The user then does not need to compensate when reading the cursor position. IO_Others exports the cursor DefaultCursor which is the upper-left pointing arrow.

The cursor can be used in a number of ways. If you want the cursor to follow the tablet and then read the tablet coordinates, use the cursor mode TrackCursor.

```
IOCursorMode(TrackCursor);
```

Be sure to turn the tablet on using IOSetModeTablet(relCursor). If you want to explicitly set the position of the cursor, use cursor mode IndepCursor. To set the cursor position, use the function IOSetCursorPos(x,y); Note that if you set the cursor position in Track mode, it will be overwritten almost immediately by the position of the tablet. You can still read the tablet in IndepCursor mode if it has been turned on; the tablet position is simply not used to set the cursor position.

To read the tablet position, use the function IOReadTablet. It returns the last x and y position read from the tablet. If the pen or puck is away from the tablet, it may be an old point. It is not possible to tell if the tablet is sensing the pen or puck. The buttons can be read using the variables TabSwitch, TabYellow, TabBlue, TabWhite, and TabGreen. TabSwitch tells if any button was pressed. For a puck, the other booleans tell which button it was. For a pen, the "colored" booleans will always be false. These booleans are true while the button is held down. The user is required to wait for a press-let up event:

```
repeat until tabswitch;
while tabswitch do;
{ read tablet position, or whatever }
```

The Cursor functions determine how the cursor interacts with the picture on the screen under the cursor. The cursor function also determines the background color. The even functions have zeroes in memory represented as white and ones as black (this is the default: white background with black characters). Odd functions have zeroes represented as black and ones as white. The functions are as follows (inverted means screen interpretation as just described):

CTWhite:	Screen picture is not shown, only cursor.
CTCursorOnly:	Same as CTWhite only inverted.
CTBlackHole:	This function doesn't work.
CTInvBlackHole:	This function doesn't work either.
CTNormal:	Ones in the cursor are black, zeros allow screen to show through.
CTInvert:	Same as CTNormal only inverted.
CTCursCompl:	Ones in the cursor are XORed with screen, zeros allow screen to show through.
CTInvCursCompl:	Same as CTCursCompl only inverted.

8. Reading Characters from the Keyboard.

The normal PASCAL character Read waits for an entire line to be typed before returning any characters. This allows editing of the line (backspace, etc.) as described in the PERQ Introductory Manual. If you want to get the characters exactly when they are hit, you must call IOCRead in IO_Unit. The normal form for this call is

```
If IOCRead(TransKey, c) = IOEIOC then { c is a valid character }
```

where IOEIOC is a constant defined in the module IOErrors and c is a character variable. If IOCRead returns some value other than IOEIOC, then no character has been hit. "Transkey" tells IO that you want the standard ASCII interpretation of the character. If you use "KeyBoard" instead, you will get the actual 8 bits returned by the keyboard. This code allows you to distinguish the special keys (INS, DEL, HELP, etc.) from the other keys and allows you to distinguish CONTROL-SHIFT-key from CONTROL-key. You will have to experiment to get the code for the desired key. There is no way to tell when a key has been let up.

IOCRead does not write out the character typed. If you want it printed, you should use Write(c). If you want to print all the special symbols in the font file (there is a picture associated with every control character), you can set the high bit of the character. This prevents the Screen package from interpreting the character as its special meaning if any. Thus, you could print the picture for RETURN by using

```
Write(chr(LOr(RETURN, #200)));
```

IOCRead also does not turn on the input marker ("_") which shows the user that he is supposed to type something. Do a SCurOn (from Screen) before requesting input and an SCurOff when done to make the underline prompt appear.

The HELP key and ^C are handled specially by the IO system. If the HELP key is hit, an exception is raised. If you do not handle this exception (called HelpKey, exported by System), "/HELP<CR>" will be put into the input stream as if typed. If you do handle this exception, you can put chr(7) into the input stream: the code for HELP. When ^C is typed, the exception CtlC is raised (also defined in System). If not caught, nothing special is done until the second ^C is hit when CtlCAbort is raised. This causes the program to exit. Note that the ^C's are put into the input stream. ^SHIFT-C causes a separate exception to be raised. If the user wants one ^C to do something special in a program (for example, abort type-out and go to top level as in FLOPPY), put the following Handler at the top level:

```
Handler CtlC;
   begin
   WriteLn('^c');
   IOKeyClear;                {remove the ^C from input stream}
```

```
        CtrlCPending := false;    {so next ^C won't abort program}
        goto 1;                   {top of command loop}
        end;
```

(IOKeyClear comes from IO_Others.) Another special character to know about is ^S. This character prevents any further output to the screen until a ^Q is typed. If you want to disable this processing, simply set CtrlSPending to false after every character is read.

9. CmdParse and PopCmdParse.

CmdParse and PopCmdParse export a number of procedures that help read and parse strings of commands and arguments. Procedures exist for handling command files (which may be nested), for parsing a string containing inputs, outputs and switches into its components, and for getting a command index from a string or a PopUp menu.

The modules CmdParse and PopCmdParse document how each of the procedures work. This section provides an example of how to use the parsing procedures in CmdParse.

```
var ins, outs: pArgRec;
    switches: pSwitchRec;
    switchAr: CmdArray;
    err: String;
    ok, leave: boolean;
    c: Char;
    s: CString;
    isSwitch: boolean;
    i: integer;
  begin

  <assign all switches to SwitchAr>

  c := NextString(s, isSwitch);  {remove "<utility>"}
  if (c<>' ') and (c<>CCR) then
      StdError(ErIllCharAfter, '<utility>', true);
  ok := ParseCmdLine(ins, outs, switches, err);
  repeat
     if not ok then StdError(ErAnyError, err, true);
     while switches <> NIL do {handle all the switches}
        begin
        ConvUpper(switches^.switch);
        i := UniqueCmdIndex(switches^.switch,
                 switchAr, NumSwitches);
        case i of
             1 : <handle switch # 1>
             2 : <handle switch # 2, etc.>
             otherwise: StdError(ErBadSwitch,
                              switches^.switch, true);
             end;
        switches := switches^.next;
        end;
     if (outs^.name <> '') or (outs^.next <> NIL) then
         StdError(ErNoOutFile, '<utility>', true);
     if ins^.next <> NIL then
         StdError(ErOneInput, '<utility>', true);
     if ins^.name = '' then
        begin
        Write('<Prompt for argument>: ');
        ReadLn(s);
        ok := ParseStringLine(s, ins, outs, switches, err);
        leave := false;
        end
```

```
    else begin
         leave := true;
         if not RemoveQuotes(ins^.name) then
             StdError(ErBadQuote, '', true);
         FSRemoveDots(ins^.name);

         <handle the argument>

         end;
until leave;
end;
```