PERQ
Systems
Corporation

PROGRAMMING EXAMPLES

March 1984

This manual is for use with POS Release G.5 and subsequent releases until further notice.

PERQ Systems Corporation
2600 Liberty Avenue
P. O. Box 2600
Pittsburgh, PA 15230
(412) 355-0900

Table of Contents January 15, 1984

1. Introduction.

The manual describes how to use Fonts, Cursors, RasterOp, Line, Windows, CmdParse, and PopUp menus and how to allocate large amounts of memory. The manual also defines the interface between Pascal modules and the Pascal IO subsystem and includes examples of sample applications.

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.

The last part of this document describes the low level access to the IO system. This is useful for applications that want to directly control the PERQ's peripherals.

2. Allocating Memory.

This section describes how to allocate blocks of memory. Memory on the PERQ is divided into "segments". Each segment can have up to 4096 blocks. Each block is 256 words or 512 bytes. You can allocate segments with CreateSegment from module Memory or with CreateHeap from module Dynamic. Although up to 2 megabytes (4096 blocks times 512 bytes per block) can be allocated in a segment, most of the software cannot deal with segments bigger than 256 blocks (128 K Bytes). The #only# way to address the blocks past this boundary is with RasterOp. Therefore, for segments containing code or program data, the effective limit of the size is 256 blocks.

When you create a segment with CreateSegment, the segment is given an initial size, a maximum size, and an increment. When segments created with CreateSegment become full, they automatically enlarge, by multiples of the increment size, until there is enough free memory for the allocation. Segments will not grow past their maximum size, however, and it may be the case that there is simply not enough room in memory for the segment, in which case a different exception will be raised.

When you create a segment with CreateHeap, the segment has a fixed size but when it is full, another segment of the same size is allocated and chained to the first segment. Allocation will then be done from the new segment. The fixed size for a Heap segment is specified at head creation time and must be less than or equal to 256 blocks. The segment number of the first segment allocated identifies the heap. If an allocation is attempted which is larger than the size of the segments, a single larger segment is created. CreateHeap should only be used for segments from which NEW's will be done.

You may need to allocate blocks of memory to read in Fonts and pictures from files, to create pictures off screen for RasterOp, and to handle large amounts of data. For managing large amounts of data, CreateHeap is appropriate; in all other cases, use CreateSegment. Fonts and pictures are generally stored in files on the disk. To use the fonts and pictures, read the file into memory. First, do a FSLookUp (or use one of the other lookup functions) from module FileSystem. A VAR parameter to this function is the number of blocks in the file. You can pass the number of blocks returned from the lookup to CreateSegment or CreateHeap to specify how much storage to allocate.

Create the segment 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. Or, create the segment using the procedure from Dynamic:

```
Procedure CreateHeap(
              var S: SegmentNumber;
              Size: 1..256)
```

where S is set to the number of the new segment and Size is the size of the initial segment. Note that all subsequent segments use this Size.

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". Segments used this way should be created using CreateSegment.

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. For 1/4 MByte systems, the default segment is made by

```
CreateSegment(heapSegment, 4,4,256)
```

For other systems, the default segment is made by

```
CreateHeap(heapSegment, 20)
```

The extended form, NEW(seg, alignment, 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. Segments used this way can be created using CreateSegment or CreateHeap, but CreateHeap is the prefered way.

NEW is implemented by a call to the procedure NewP in Dynamic. You can call this procedure directly 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 or CreateHeap. 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 might 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. An alternative, and easier, strategy, is to use CreateHeap with a small size for the initial segment. In this case, the memory system automatically creates a number of segments and manages their swapping.

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). Note that the MultiRead procedure works only on hard disks.

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. Remember that neither MultiRead or MakePtr can address a segment bigger than 256 blocks long.

4. RasterOp and Line.

RasterOp and Line are the chief graphics primitives of the PERQ. Each is fast. The 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 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 variables 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 up:

```
  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) for a portrait screen and (1279, 1023) for a landscape screen. 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}
     RAndNot  {dest gets dest AND invert of src}
     ROr      {dest gets dest OR src}
     RorNot   {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 for portrait monitors and 80 for landscape monitors; it is 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 alignment 4.

Line is used for drawing straight, single width lines. It comes in two forms. The first, called #Line# will draw lines on the screen or on buffers with the same width as the screen. The second form, called #SVarLine# will draw lines on any width buffer and takes the word width of the buffer the same way RasterOp does. Both of these procedures are exported by the Screen Module. Both take a source and destination x and y position, a style and a pointer to the buffer to draw in. 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. Similarly, SVarLine is defined as:

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

where #width# is the word width of the buffer described by p and must be a multiple of 4.

The Screen module exports two variables that will be useful for programs dealing with the screen. SBitWidth is the width of the screen in bits (768 for portrait screen and 1280 for landscape). SBitHeight is the height of the screen in bits (currently always 1024).

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-sized.

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. Unfortunately, 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 - 3;
width := width + 7;
orgY := orgY - 3;
height := height + 7;
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;
orgY := orgY + 2;
height := height - 4;
end;
```

Each window has an associated font that is used for writing in that window. You can change the font with SetFont. Note that when you create a window, the title line is written in the font from the current window.

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. The bit pictures are defined in buffers whose width is always 48 (PortraitWordWidth) even if the screen is a landscape monitor. 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, KSetSLen,
                     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. Note that you should not use SScreenW for the Font Word width since the word width is always fixed (at PortraitWordWidth) and SScreenW may be different on Landscape monitors.

7. Cursors.

In a PERQ system, the term "Cursor" is used in two ways. 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.

You can set the picture in the cursor. PERQ software 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, which takes a CurPatPtr and two integers to locate 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(TabletMode). Specify relTablet (IOSetModeTablet(relTablet)) as the argument to turn the tablet on. When TabletMode is relTablet, puck position can be read in absolute mode or in relative mode. #RelTablet# is misnamed. It means turn the tablet on. Do not use AbsTablet or ScrAbsTablet to turn on the tablet.

To control whether the tablet is in relative or absolute mode, use the Procedure IOSetRealRelTablet. In absolute mode, cursor position on the screen is determined by the actual (absolute) tablet coordinates of the puck; the x and y coordinates are simple linear transformations of the actual values to provide a one to one mapping of the screen into the tablet surface. If the puck is in the upper-left corner of the tablet, the cursor is in the upper-left corner of the screen. In relative mode, lifting the puck or pen from the tablet surface and then returning it does not alter cursor position on the screen. Only the movement of the puck or pen on the tablet surface causes corresponding delta-x and delta-y changes in cursor position. Typically, you specify the mode as a switch to the Login command (see the PERQ Utility Programs Manual).

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 (and RealRelTablet is false), it is overwritten almost immediately by the position of the tablet. You can still read the tablet in IndepCursor mode if it was 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. 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 three-button puck, TabBlue is always false. For a pen, the "colored" booleans are always 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; zeroes black, ones white):

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 System Overview. 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, etc.) from the other keys and allows you to distinguish CTRL/SHIFT/key from CTRL/key. Some keys raise exceptions. The only way to find out if the HELP key, CTRL/SHIFT/C, and CTRL/SHIFT/D have been hit is to catch the exception. 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 CTRL/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 CTRL/C is typed, the exception CtlC is raised (also defined in System). If not caught, nothing special is done until the second CTRL/C is hit when CtlCAbort is raised. This causes the program to exit. Note that the CTRL/C's are put into the input stream. CTRL/SHIFT/C causes a separate exception to be raised. If the user wants one CTRL/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 CTRL/C from input stream}
  CtrlCPending := false;   {so next CTRL/C won't abort program}
  goto 1;                  {top of command loop}
  end;
```

(IOKeyClear comes from IO_Others.)

Another special character to know about is CTRL/S. This character prevents any further output to the screen until a CTRL/Q is typed. If you want to disable this processing, simply set CtrlSPending to false after every character is read.

IOCRead always removes the character from the input buffer if it is there. To test if a character is ready without removing it, use IOCPresent(Keyboard).

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 := ParseCmdArgs(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 := ParseStringArgs(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;
```

10. General I/O Operation.

This section provides an overview of some low level IO calls. Subsequent sections describe how to do I/O to specific devices. Only applications that need to directly control PERQ peripherals will need the information in these sections.

The module IO_Unit contains the pascal procedures which perform IO operations.

10.1 UnitIO

The Procedure UnitIO does all IO except for single character reads and writes. UnitIO is defined as follows.

```
Procedure UnitIO( Unit : UnitRng,
                  Bufr : IOBufPtr,
                  Command : IOCommands,
                  ByteCnt : integer,
                  LogAdr : double
                  HdPtr : IOHeadePtr,
                  StsPtr : IOStatPtr );
```

The definitions for the types of the parameters are in the module IO_Unit. The parameters have the following meanings:

Unit - Tells the IO system which device it should work with. Unit must be one of:

Clock
EIODisk
Floppy
GPIB
HardDisk
PointDev
RSA
RSB (EIO board only)
Speech
Z80 (EIO board only)

Bufr - Points to the information the IO system should send to a device or to a location where the IO system should put information received from a device.

Command - Tells the IO system what it should do with respect to a device. The valid commands are:

IOConfigure - Changes or sets some device state according to the information pointed to by Bufr.

IODiagRead - Does a read of the HardDisk without checking the logical header on the disk against the logical header pointed to by HdPtr. The IO system will write the logical header from the disk to the area pointed to by HdPtr.

```
IOStatus = record
    HardStatus : integer;
    SoftStatus : integer;
    BytesTransferred : integer;
```

HardStatus - Status information provided by the device. Hardstatus is device dependent.

SoftStatus - Status information provided by the IO System. IOErrors exports the complete list of SoftStatus values. If SoftStatus is IOEIOC upon return from UnitIO, the operation was successful. Anything else indicates that an error has occured.

BytesTransferred - Number of bytes of information transferred between a device and the the IO system. Should be equal to ByteCnt upon return.

10.2 Single Character IO

10.2.1 Reads

There are two procedures which read a character from a character device. They are defined as follows.

```
function IOCRead( Unit: UnitRng; var Ch: char ): integer;
function IOCRNext( Unit: UnitRng; var Ch: char ): integer;
```

Unit must be one of:

- Keyboard
- Transkey
- GPIB
- RSA
- RSB

Ch is assigned a character value that the device sent to the IO system.

The return value will be one of

- IOEIOC - character read
- IOEBUN - Unit not a legal Unit number
- IOENCD - Unit not a character device
- IOEOVR - see below
- IOEIOB - see below

IOEOVR - All character devices have an associated character buffer. The IO system puts characters received from a device into its character buffer and removes characters from the character buffer when IOCRead or IOCRNext is called. If IOCRead/IOCRNext returns the value IOEOVR it means that the IO system lost characters sent by a device because the device's character buffer was full. The returned character is valid. The lost characters were received after the character returned in the previous call to IOCRead, but before the returned character.

IOEIOB - Only IOCRead can return this value. It means that there is no character available from the specified device. IOCRNext does not return until is has a character from the specified device, however long it may have to wait.

To determine if a device has sent a character without actually reading the character use the function IOCPresent, defined as

```
function IOCPresent( Unit: UnitRng ): boolean;
```

This function is true if the device specified is a character device and has sent a character.

10.2.2 Writes

The function IOCWrite sends a character to a character device. It is defined as:

```
Function IOCWrite( Unit: UnitRng; Ch: char ): integer;
```

Unit must be one of

- GPIB
- RSA
- RSB
- ScreenOut
- Speech

Ch is the character to write.

The return value will be one of

- IOEIOC - character sent successfully
- IOENCD - unit is not a character device
- IOEBUN - unit is not a device

10.3 Interrupts

Usually, the IO system handles all device interrupts. They are transparent to pascal modules. If pascal modules wish to trap interrupts themselves, they can tell the IO system to raise an exception when it receives an interrupt from a device. To enable/disable such exception raising use the IOSetExceptions procedure defined as

```
Procedure IOSetExceptions( Unit : UnitRng;
                           IntType : IntrType;
                           var Setting : boolean );
```

Unit - the device for which to enable/disable interrupt exception

IntType - the type of interrupt exception to enable/disable must be one of

IODataInterrupt
IOATNInterrupt

Setting

true enables the interrupt exception
false disables the interrupt exception

When IOSetException returns, Setting will be true if the interrupt exception was enabled before the call to IOSetException and false if the interrupt exception was disabled before the call to IOSetException.

The exception the IO system will raise is defined as

```
Exception DevInterrupt( Unit : UnitRng;
                        IntType : IntrType;
                        ATNCause : Integer );
```

Unit - the device sending the interrupt

IntType - the type of interrupt it sent will be one of

IODataInterrupt
IOATNInterrupt

ATNCause - the cause of an attention interrupt

The IO system raises an IODataInterrupt whenever the character buffer of a character device goes from empty to nonempty. The IO system raises an IOATNInterrupt whenever the IO system receives an attention interrupt from a device. Before raising one of these exceptions, the IO system disables attention and data available interrupts for that device. This prevents the system from raising a second exception while the first is being processed. The IO system reenables these interrupts upon returning from the exception handler or when IOClear exceptions is called.

11. Device Operation.

This section describes specific device operation at the lowest level.

11.1 HardDisk

Normally, application access to the disk is through the file system, which uses the interface described in this section. Few, if any, applications will need to call UnitIO for the hard disk.

The Following UnitIO Commands are legal.

```
     IODiagRead
     IOFormat
     IORead
     IOReset
     IOSeek
     IOWrite
     IOWriteFirst
```

Bufr - Must point to a 256 word aligned area of memory or be nil. If it is neither, the IO system will assign IOEBAE to SoftStatus and return without executing the command.

ByteCnt - Must be a nonnegative multiple of 512. If it isn't, the IO system will assign IOEBSE to SoftStatus and return without executing the command.

LogAdr - LogAdr[0] contains the Disk Address, LogAdr[1] is ignored.

HdPtr - Points to the Disk Header. The Disk Header is defined as

```
IOHeader = record
     SerialNum : double;
     LogBlock  : integer;
     Filler    : integer;
     NextAdr   : double;
     PrevAdr   : double
     end;
```

StsPtr -
BytesTransferred will be set.

The hard status for the EIO disk differs from the hard status for the hard disk. The hard status for the EIO disk is defined in DiskDefs as SMStatus. The hard status for the hard disk follows.

```
DskResult = packed record
  case boolean of
    true  : ( Result : integer );
    false : ( CntlError : ( OK,
                            AddrsErr
                            PHCRC,
                            LHSER,
                            LHLB,
```

```
                    LHCRC,
                    DaCRC,
                    Busy );
         Fill2     : boolean;
         TrackZero : boolean;
         WriteFault : boolean;
         SeekComplete : boolean;
         DriveReady : boolean)
end;
```

CntlError -

```
     OK       - operation successful
     AddrsErr - address error
     PHCRC    - physical header CRC
     LHSer    - logical serial wrong
     LHLB     - logical block wrong
     LHCRC    - logical header CRC
     DaCRC    - data CRC
     Busy     - device busy
```

Fill2 - uses up space

TrackZero - the head is at track zero

WriteFault - write failed

SeekComplete - the head is not moving

DriveReady - drive is ready

SoftStatus will be one of

```
   IOEIOC = Operation successful
   IOEILC = Command not one of those listed above
   IOEBUN = Illegal Unit number (not a device)
   IOEBSE = ByteCnt not a multiple of 512
   IOENHP = Nil HdPtr
   IOETIM = Disk operation did not complete
   IOEWRF = A Write Fault of some sort
   IOEADR = Address Error
   IOEPHC = Physical Header CRC
   IOELHS = Logical Serial Number Wrong
   IOELHB = Logical Block Number Wrong
   IOELHC = Logical Header CRC
   IOEDAC = DataCRC
   IOEDNI = Disk Busy
   IOEBAE = Bufr not aligned properly
```

11.2 Floppy

The following UnitIO commands are legal:

IOFormat - formats the specified track. LogAdr[1] is the cylinder to format. It must be within the range 0 to 76. If it isn't, the IO system will set SoftStatus to IOECOR and return without formatting the floppy. ByteCnt must be a multiple of four. If it isn't, the IO system will set SoftStatus to IOEBSE and return without formatting the floppy. The IO system will format the the specified track so that it has ByteCnt/4 sectors. (POS generally assumes that a floppy has 26 sectors to a track, thus ByteCnt should be 104.) Bufr points to at least ByteCnt bytes of information. Each four bytes of information defines a sector ID. A sector ID is defined as

```
Byte 1 - Cylinder (same as LogAdr[1])
Byte 2 - Head     (0..1)
Byte 3 - Sector   (1..ByteCnt/4)
Byte 4 - N        (0=128, 1=256)
```

The IO system does no checking of sector ID's. If byte values are out of range, that sector on the floppy cannot be used. If two sector ID's have the same Sector value, reads and writes to that Sector will randomly choose between one and the other. BytesTransferred is 0.

IORead - reads data from the floppy. ByteCnt is the number of bytes of data to read and must be a multiple of the SectorSize. If it isn't, the IO system will set SoftStatus to IOEBSE and return without reading any data. Bufr points to the memory space which will hold the data. LogAdr contains the initial cylinder and sector number. The IO system will read the data from this sector. If it needs to read more data, it will read the next sector on the cylinder. If there are no more sectors on the cylinder it will read the first sector on the next cylinder. It continues this process until it has read the necessary number of bytes. BytesTransferred will be 0.

IOReset - puts the floppy in an idle state. The heads are left at track 0. ByteCnt, Bufr, and LogAdr are ignored. BytesTransferred is 0.

IOSeek - moves the floppy's head to the specified track. LogAdr[1] is the track number.

IOWrite - writes data to the floppy. It is identical to IORead except that it writes data and that Bufr points to the data to write to the floppy.

Unit - is Floppy

Bufr - see below (Must always point to a quad word aligned memory space. If it doesn't, the IO system will set SoftStatus to IOEBAE an return without executing the command.)

Command - see below

ByteCnt - see below

LogAdr - see below

HdPtr - ignored

StsPtr -
BytesTransferred will be set

HardStatus is as follows

bit 0 - missing address mark
bit 1 - not writeable
bit 2 - no data
bit 3 - not used
bit 4 - overrun
bit 5 - data error
bit 6 - not used
bit 7 - end of cylinder

SoftStatus will be one of
IOEIOC - operation successful
IOEBUN - illegal unit number
IOIILC - illegal command
IOEBAE - bad buffer alignment
IOECOR - cylinder out of range
IOESOR - sector our of range
IOEBSE - ByteCnt not multiple of blocksize
IOEDNR - device not ready
IOEUEF - equipment fault (not your fault)
IOEOVR - floppy overrun
IOEMDA - missing header address mark
IOEDNW - device not writable
IOECMM - cylinder mismatch
IOESNF - sector not found
IOEDAC - data CRC error
IOELHC - logical header CRC error

11.3 RS232 and Speech

On the EIO board there are two RS232 channels, RSA and RSB. In addition, Speech output and PointDev input is implemented via a third RS232 Channel. On the CIO board, RSB does not exist. Below, RS232 stands for one of RSA, RSB, or Speech. Section 11.7 details the PointDev.

Single Character reads are legal for RSA,
legal for RSB,
illegal for Speech.

Single Character writes are legal for all three.

The following UnitIO commands are legal:

```
IOConfigure
IOReset
IOSense
IOWrite
IOWriteHiVol { not legal for RSB }
IOWriteRegs
```

Unit - Has the value RSA, RSB, or Speech

Bufr - See below

Command - See below

ByteCnt - See below

LogAdr - ignored

HdPtr - ignored

StsPtr -

```
     BytesTransferred see below
     HardStatus will be 0
     SoftStatus will be one of
        IOEIOC = Operation Successful
        IOEBUN = Illegal Device
        IOEILC = Illegal Command
        IOEBAE = buffer not aligned correctly for hivol write
        IOEBSE = see below
        IOERDI = illegal register number
        IOECDI = illegal baud rate
```

Interrupts - The IO system will raise IOATNInterrupt exceptions and IODataInterrupt exceptions if so enabled.

The valid commands perform as follows:

Single Character Reads - Nothing unusual

Single Character Writes - Nothing unusual

IOConfigure - For RSA and RSB this command sets the transmit and receive baud rate. ByteCount must be two. BytesTransferred will be set to zero. Bufr points to at least two bytes of information. The first byte of contains the transmit baud rate. The second byte contains the receive baud rate. The baud rate must be one of

RSEXT
RS110
RS150
RS300
RS600
RS1200
RS2400
RS4800
RS9600
RS19200 (EIO board only)

For Speech, this command sets the bit rate. ByteCount must be two. BytesTransferred will be set to zero. Bufr points to at least two bytes of information. These two bytes form an integer count. The first byte being the low order byte of the count, the second the high order byte. (For CIO boards, the second byte is ignored.) The IO system loads this count into the CTC chip. To determine the correct count to load for a desired bit rate, divide the base clock rate by the desired rate. The base clock rate of a CIO board is 2.456 Mhertz. The base clock rate of an EIO board is 4 Mhertz.

IOReset - This command halts RS232 communications and places the specified device into an idle state. Characters in the input character buffer are not affected. Characters waiting to be sent to the device are discarded. Both Bufr and ByteCnt are ignored. BytesTransferred will be set to zero.

IOSense - This command puts two bytes of status information into the memory Bufr points to. ByteCnt is ignored. BytesTransferred is set to 2. The IO system puts Read Register 1 of the SIO chip into the first byte, Read Register 2 into the second byte.

IOWrite - This command sends data out on the RS232. Bufr points to the data to send. ByteCnt is the number of bytes of data to send. BytesTransferred is set to the number of bytes actually transferred.

IOWriteEOI - This command is like IOWrite except that the last byte is sent with the a CRC. The name is rather confusing and may be changed in the future.

IOWriteHiVol - This command is like IOWrite except that the information is sent via a DMA chip.

IOWriteRegs - This command programs the SIO controller chip. ByteCnt must be even and less than 13. If it isn't, the IO system will set SoftStatus to IOEBSE and return without sending any information to the SIO controller chip. Bufr points to ByteCnt/2 pairs of bytes. The first byte of each pair must be one of

0 : Command Register
3 : Receiver Logic and Parameters
4 : Control for Tx and Rx
5 : Tx Control
6 : Sync Char 1
7 : Sync Char 2

If it isn't, the IO system will set SoftStatus to IOERDI and return without sending any information to the SIO controller chip. The Second byte of the pair is the value to write to the register. IO_Unit contains a type RS_WriteReg which gives more information about these registers. BytesTransferred will be set to ByteCnt. NOTE : Since the PointDev is implemented via the same port as Speech, changing the registers of the Speech device may affect the PointDev.

After an IOReset command, the SIO controller registers have been set to:

For Z80 reg 0 - Write to command register:

NextRegisterPointer is set to: 0
Command is set to: R_NullCommand
ResetCRC is set to: R_NullResetCRC

For Z80 reg 3 - Write to receiver logic and parms:

RSRcvEnable is set to: true
SynCharLoadInhibit is set to: false
AddressSearchMode is set to: false
RxCrcEnable is set to: false
EnterHuntPhase is set to: false
AutoEnables is set to: true
RSRcvBits is set to: RS_8

For Z80 reg 4 - Write to control for Tx and Rx:

RSParity is set to: RS_NoParity
RSStopBits is set to: RS_St1x5
SyncMode is set to: R_8BitSync
ClockRate is set to: R_X16

For Z80 reg 5 - Write to control for Tx:

TxCrcEnable is set to: false
RTS is set to: true
UseCrc16 is set to: false
TxEnable is set to: true
SendBreak is set to: false
RSXmitBits is set to: RS_Send8
DTR is set to: true

11.4 GPIB

Single Character Reads are legal.

Single Character Writes are legal.

The Following UnitIO Commands are legal.

```
IOConfigure
IOReadHiVol
IOReset
IOSense
IOWrite
IOWriteEOI
IOWriteHiVol
IOWriteRegs
IOFlush
```

Unit - has the value GPIB

Bufr - see below

Command - see below

ByteCnt - see below

LogAdr - timeout count

HdPtr - ignored

StsPtr -
```
    BytesTransferred see below
    HardStatus will be 0
    SoftStatus will be one of:
       IOEBUN = Unit is not a legal device
       IOEBSE = Bad ByteCnt, see below
       IOEILC = Illegal command
       IOEUDE = IO System error (it's not your fault)
       IOEIOC = Command Successful
       IOERDI = Illegal register number
       IOEBAE = Bufr not quad word aligned
       IOECDI = Bad configure information
       IOEDNR = Device not ready
```

Interrupts - The IO system will raise ATNInterrupt exceptions and DataInterrupt exceptions if so enabled. The GPIB will not send attention interrupts unless it is configured to do so by an IOConfigure command. Furthurmore, an IOWriteRegs command must be used to indicate which attention interrupts the GPIB should send to the IO system.

The valid commands perform as follows:

Single Character reads - IOCRead and IOCRNext will never return IOEOVR as the character buffer can never fill completely. (The IO system doesn't let the GPIB send more characters if it has no place to put them.)

Single Character writes - nothing unusual

IOConfigure - This command enables/disables Pascal's receiving interrupts other than Data In and Data Out from the GPIB contoller chip. ByteCnt must be 1. If it isn't, the IO system will set SoftStatus to IOEBSE and return without configuring the GPIB. Bufr points to at least one byte of information. This value of this byte must be 0 or 255. If not, the IO system will set SoftStatus to IOECDI and return without configuring the GPIB. If the first byte of information is 0, the GPIB controller chip will not send attention interrupts to the IO system. If it is 255, the GPIB controller chip will send attention interrupts to the IO system. BytesTransferred will be set to 0.

IOReadHiVol - This command reads data from the GPIB via a DMA channel. This provides a high transfer rate. ByteCnt is the number of bytes of information to be read and must be greater than one. If it isn't, the IO system will set SoftStatus to IOEBSE and return without reading any characters. Bufr points to an area of memory into which the IO system will put the data read. This area must be aligned on a four word boundary. If it isn't, the IO system will set SoftStatus to IOEBAE and return without reading any characters. BytesTransferred will be set to ByteCnt.

IOReset - This command puts the GPIB controller into an idle state. ByteCnt must be 0. Bufr is ignored. BytesTransferred will be set to 0.

IOSense - This command provides 10 bytes of status information. ByteCnt is ignored. Bufr must point to at least 10 bytes of memory. The following bytes are provided:

Byte 1 : Interrupt Status 0
Byte 2 : Interrupt Status 1
Byte 3 : Address Status changed
Byte 4 : Bus Status
Byte 5 : Address Switch 1
Byte 6 : Command Pass Through
Byte 7 : Address Status
Byte 8 : Bus Status
Byte 9 : Address Switch
Byte 10: Command Pass Through

The difference between bytes 1 through 6 and bytes 7 through 10 is that bytes 1 through 6 show status at most recent interrupt while bytes 7 through 10 is current as of time IOSense is issued.

BytesTransferred is set to 5.

IOWrite - This command sends data out to the GPIB. ByteCnt is the number of bytes of information to send. Bufr points the information to send. BytesTransferred will be set to the number of bytes actually sent to the GPIB. (This may differ from ByteCnt if some error occurs during transmission.)

IOWriteEOI - This command is identical to IOWrite except that EOI is set with the last byte of information.

IOWriteHiVol - This command is identical to IOWrite except that the information is sent via a DMA channel. This allows faster transmitting of information.

IOWriteRegs - This command programs the registers on the GPIB controller chip. ByteCnt must be even, otherwise the IO system will set SoftStatus to IOEBSE and return without writing any information to the GPIB controller chip. Bufr points to pairs of bytes. The first byte of each pair indicates which register to write and must be one of the following:

0 - Interrupt Mask 0
1 - Interrupt Mask 1
3 - Auxilliary Command
4 - Address Register
5 - Serial Poll
6 - Parallel Poll

If it isn't, the IO system sets SoftStatus to IOERDI and returns without sending any information to the GPIB controller chip. BytesTransferred is set to ByteCnt.

NOTE -- The registers above are in the order given in the Texas Instruments data manual for the GPIB controller Chip.

11.5 Keyboard

Single character reads are legal. Single character reads to the keyboard return the eight bit character generated by the keyboard. Some of these characters are not valid ASCII characters as they have the high order bit set. If you wish to receive valid ASCII characters only, use device Transkey. Transkey will map characters with the high order bit set to appropriate control characters.

11.6 Clock

The UnitIO commands IOConfigure and IOSense are legal.

Unit - has the value Clock

Bufr - see below

Command - see below

LogAdr - ignored

HdPtr - ignored

```
StsPtr -
     BytesTransferred - see below
     HardStatus will be 0
     SoftStatus will be one of
          IOEIOC - operation successful
          IOEILC - illegal command
          IOEBUN - illegal device
          IOEBSE - bad byte count for configure command
          IOECDI - bad configure information
```

A specific description follows:

IOConfigure - sets the clock. ByteCnt must be six. Bufr points to six bytes of information. The six bytes are as follows

```
 1 - Cycles  (50 or 60)
 2 - Year    (year mod 100)
 3 - Month   (1..12)
 4 - Day     (0..31)
 5 - Hour    (0..23)
 6 - Minute  (0..59)
```

If any of the bytes is not in the specified range, the IO system sets SoftStatus to IOECDI and returns. Values before the out of range value are set correctly. BytesTransferred is 0.

IOStatus - Provides the date and time. ByteCnt is ignored. Bufr must point to at least eight bytes of memory space. The IO system provides the following bytes of information.

```
ClockStat = packed record
       Cycles  : 0..255;
       Year    : 0..255;
       Month   : 0..255;
       Day     : 0..255;
       Hour    : 0..255;
       Minute  : 0..255;
       Second  : 0..255;
       Jiffies : 0..255
       end;
```

BytesTransferred will be 8.

11.7 PointDev

The following UnitIO commands are legal:

IOConfigure - turns the pointdev on or off. ByteCnt must be one. If not, the IO system will set SoftStatus to IOEBSE and return without changing the state of the PointDev. Bufr points to at least one byte of information. If the first byte is zero, the IO system will turn the PointDev off. If it is not zero, the IO system will turn the PointDev on. BytesTransferred will be 0.

IOSense - finds out if the PointDev is on or off. ByteCnt is ignored. Bufr points to a memory area. The IO system will assign zero to the first byte of this area if the PointDev is on. It will assign 255 to this byte if the PointDev is off. BytesTransferred is set to 1.

Unit - has the value PointDev

Bufr - see below

Command - see below

ByteCnt - see below

LogAdr - ignored

HdPtr - ignored

StsPtr -
- BytesTransferred see below
- HardStatus will be 0
- SoftStatus will be one of
  - IOEIOC - operation successful
  - IOEILC - illegal command
  - IOEBUN - illegal device
  - IOEBSE - see below
  - IOEUDE - undefined system error (not your fault)

11.8 Transkey

Single character reads are legal. Single character reads to the transkey are identical to single character reads to the keyboard except that the IO system maps characters with the high order bit set to appropriate control characters.

11.9 ScreenOut

Single Character Writes are legal. IOCWrite will always return IOEIOC.

11.10 Z80

The following UnitIO commands are legal:

IOReadHiVol
IOSense
IOWriteHiVol
IOWriteRegs

Unit - Has the value Z80

Bufr - See below

Command - See below

ByteCnt - See below

LogAdr - See below

HdPtr - ignored

StsPtr -
- BytesTransferred see below
- HardStatus will be 0
- SoftStatus will be one of
  - IOEBUN = Z80 commands not valid
  - IOEBSE = Bad block size
  - IOEBAE = buffer not quad word aligned
  - IOEILC = Illegal command
  - IOEIRD = Bad register number
  - IOEUDE = Unknown error
  - IOEIOC = Operation complete

The valid commands perform as follows:

IOReadHiVol - reads data from the Z80 memory. ByteCnt is the number of pieces of data to read and must be greater than one. LogAdr[0] contains the Z80 memory address to start reading from. Bufr points to an area of memory into which the IO system will put the data from the Z80. Bufr must be quad word aligned.

IOWriteHiVol - writes data into the Z80. ByteCnt is the number of bytes of data to write and must be greater than one. LogAdr[0] is the Z80 address to write the data to. Bufr points to the data to write and must be quad word aligned. WARNING: Changing the Z80 memory can result in the PERQ not working.

IOSense - provides two bytes of information. Bufr points to at least 2 bytes of memory. The 2 bytes of information are the major and minor version numbers of the Z80 code. See IO_Unit for a definition of the Z80 status.

IOWriteRegs - transfers control to a specified location in the Z80 memory. LogAdr[0] contains the memory location to jump to. WARNING: Jumping to random places in Z80 memory can result in the PERQ not working.