This document is a quick guide to the routines available in the Perq Operating System. They are presented as the commented EXPORT declarations of all the modules which make up the operating system.

## SYSTEM

```
Program System;
{ Perq Operating System "Main Program"
  Copyright (c) 1980, Three Rivers Computer Corporation, Pittsburgh Pa

  This Module is the outer level main program for the Perq Operating System
  It includes the loader, the command interpretter, error routine entry
 points...

}

{********************} Exports {********************}

Const MainVersion = 'A';        {The Major Version Level
                                  for the Operating System}
      DebugSystemInit = False;  {Set true for breakpoints during system init.}

Var UsrCmdLine: String;         {The string typed on the keyboard to execute a
                                 command.  This is the string to look at for
                                 parameters (See module CmdParse)}
    SystemVersion: Integer;     {The Minor Version Level
                                 for the operating system}
    SystemInitialized: Boolean;{Set true when system is up.  Used to control
                                 error handling}
    UserMode: Boolean;          {Set true when a user program is executing.
 Used
                                 to control error handling}
    CmdFile: Text;              {Indirect Command file}
{ the following procedures are called by microcode, which knows the order of
  declaration.  These must be the FIRST three procedures declared in the main
  program }
Procedure SegmentFault( Seg1,Seg2: Integer ); {Called when a non resident
                                    segment is accessed}
Procedure StackOverFlow;        {Called when the user stack overflows}
Procedure RunTimeError( Err: Integer );  {Called on various execution errors
                                such as division by zero, range errors....}
{These error handling procedures are available for user programs}
Procedure UserError( Message: String );  {Prints Message on console,
                                  then a trace back.    Then it exits to the
                                   command interpretter}
Procedure UserDump( Message: String ); {Called when Control-Shift D is typed}
Procedure Load( RunFileName: String ); {Load A user program from a file named
                                 RunFileName.RUN}
                                 {SHOULD NOT BE CALLED BY A USER PROGRAM}
Procedure Command;              {Interpret the next command}
Procedure BreakPoint( Break: Integer );  {Cause a breakpoint number   Break}

{********************} Private {********************}
```

DYNAMIC

```
{$R-}
{$Q-}
module Dynamic;
{ Perq Operating System Dynamic Memory Manager
  Written by John Strait
  Copyright (c) 1980
  Three Rivers Computer Corporation,  Pittsburgh PA.


This module implements New/Dispose for Pascal
It is NOT intended to be called directly by a user program
}


exports


imports Memory from Memory;


{ ---------------------------

Dynamic Allocation - New and Dispose.

  New( S: SegmentNumber; A: integer; var P: Pointer; L: integer );
  Dispose( var P: Pointer; L: Length );

  L = length of node in words, Ø represents a length of 2**16.
      if L is odd, L+1 words are allocated or de-allocated.
  A = alignment of node in words relative to beginning of segment,
      Ø represents an alignment of 2**16.  if A is odd, A+1 is used
      as the alignment.

Free memory is linked into a circular freelist in order of address.
Each free node is at least two words long and is of the form

      record Next: integer;
             Length: integer;
             Rest: 2*Length - 2 words
             end;

Where Next*2 is the address of the next free node and Length*2 is the
number of free words.

---------------------------- }


{The proceedures have different names to distinguish them for the predeclared
 New/Dispose procedures}

 procedure NewP( S: SegmentNumber; A: integer; var P: Pointer; L: integer );
 procedure DisposeP( var P: Pointer; L: integer );
```

private

READER

```
{ L LP:}
{$R-  shut off range checks }
{$Q-}


module Reader;
{ Perq Operating System   Read portion of Pascal Stream Package

  Written by John Strait
  Copyright (c) 1980,  Three Rivers Computer Corporation, Pittsburgh PA

  This module is not intended to be called by a user program.  The
  compiler will emit calls to these routines for read(...)/readln(...)

}


exports


imports Stream from Stream;


 procedure ReadBoolean( var F: FileType; var X: boolean );
 procedure ReadCh( var F: FileType; var X: char; Field: integer );
 procedure ReadChArray( var F: FileType; var X: ChArray; Max, Len: integer );
 procedure ReadIdentifier( var F: FileType; var X: integer;
                           var IT: IdentTable; L: integer );
 procedure ReadInteger( var F: FileType; var X: integer );
 procedure ReadString( var F: FileType; var X: String; Max, Len: integer );
 procedure ReadX( var F: FileType; var X: integer; B: integer );


private
```

## WRITER

```
{ L LP:}
{$R-  shut off range checks }
{$Q-}


module Writer;
{ Perq Operating System   Write portion of Pascal Stream Package
  Written by John Strait
  Copyright (c) Three Rivers Computer Corporation, Pittsburgh PA


  This module is NOT intended to be called by a user program
  The Pascal compiler emits calls to these routines for write(..)/writeln(...)
}

exports


imports Stream from Stream;


 procedure WriteBoolean( var F: FileType; X: Boolean; Field: integer );
 procedure WriteCh( Var F: FileType; X: char; Field: integer );
 procedure WriteChArray( var F: FileType; var X: ChArray; Max, Field: integer
 );
 procedure WriteIdentifier( var F: FileType; X: integer;
                            var IT: IdentTable; L, Field: integer );
 procedure WriteInteger( var F: FileType; X: integer; Field: integer );
 procedure WriteString( var F: FileType; var X: String; Field: integer );
 procedure WriteX( var F: FileType; X, Field, B: integer );


private
```

SCREEN

```
{$R-}
Module Screeen;

{****************************************************************}
{ Perq Screen Driver                                             }
{ Written By: Miles A. Barel    July 1, 1980                     }
{             Copyright (c) 1980                                 }
{             Three Rivers Computer Corporation                  }
{             Pittsburgh, PA 15213                               }
{****************************************************************}

Exports

Type
        FontPtr = Font;
        {This is the standard, system wide definition of a font}
        Font = Packed Record     { Contains character sets }
                Height:integer; { Height of the KSet }
                Base: integer;  { distance from top of characters to base-line
 }
                Index: Array [0.. 177] of { Index into character patterns }
                    Packed Record case boolean of
                      true: (Offset: 0..767; { position of char in patterns }
                        Line: 0..63;    { Line of patterns containing char }
                        Width: integer); { Width of the character }
                      false:(Loc:integer; Widd: integer)
                    end;
                Filler: array[0..1] of integer;
                Pat: Array [0..0] of integer;  { patterns go here }
                                          { We turn off range checking to }
                                          { access patterns, hence allowing }
                                          { KSets of different sizes }
            end;
        {This module contains a temporary, simple minded window package}
        {  Windows are referred to by indexes}
        WinRange = 0..15;          { Range of Legal Window Indexes }
        WindowType = Packed Record
                      winBY, winTY, winLX, winRX,    { Limits of window area }
                      winHX, winHY, winMX, winMY,    { Limits of useable area
 }
                      winCurX, winCurY, winFunc: integer; {Location of the
                                                     insert cursor.  The
 insert
                                                     cursor is distinguished
                                                     from the hardware screen
                                                     cursor}
                      winKSet: Font;                 { Current font for this
                                                       window}
                      winCrsChr: char;               { The Character to be
 used
                                                       for the insert cursor }
                      winHasTitle, winCursorOn: boolean;
                      end;
```

```
Procedure InitScreen;          {Called by IO system to initialize this module}
Procedure SPutChr(CH:char);     { put character CH out to current position }
                                { on the screen.  Characters FF, EOLN and BS
 }
                                { have special meanings:                   }
                                {        FF - clear screen                 }
                                {        EOLN - advance to the next line   }
                                {        BS - erase previous character     }
Procedure SSetCursor(X,Y: integer);        { Set Cursor Position to X,Y }
Procedure SReadCursor(var X,Y: integer);{ Read Cursor Position }
Procedure SCurOn;                          { Enable display of Cursor }
Procedure SCurOff;                         { Disable display of Cursor }
Procedure SCurChr(C: char);                { Set cursor character }
Procedure SChrFunc(F: integer);            { Set raster-op function for SPutChr }
Procedure SSetSize(Lines: integer);        { Set Screen Size (multiples of 128) }
Procedure CreateWindow(WIndx: WinRange; { Create a new Window}
              OrgX, OrgY, Width, Height: integer; Title: string);
                                           {   at OrgX,OrgY  Height by Width
                                               if Title is non null, set the title
                                               Put the cursor at the upper left
                                               corner, set the window's font
                                               and cursor character to that of the
                                               current window, clear the window,
                                               and make it the Current window }
Procedure ChangeWindow(WIndx: WinRange); { Make WIndx window the current
 window}
Procedure GetWindowParms(var WIndx: WinRange;  {Return state of this window}
                        var OrgX, OrgY, Width, Height: integer);
Procedure ChangeTitle(Title: string);      {Set the title of the current window}
Procedure SetFont(NewFont: FontPtr);       {Set the font of the current window}
Function GetFont : FontPtr;                {Return a pointer to the current font}

var
       WinTable: array [WinRange] of WindowType;
                                           {The keeper of the window stuff}
Private
```

## CMDPARSE

```
Module CmdParse;

{*****************************************************************
{
{ CmdParse:  Simple command parsing routines.
{
{ Written by Don Scelza    April 30, 1980
{
{ Copyright (C) 1980
{ Three Rivers Computer Corporation
{ 160 North Craig Street
{ Pittsburgh, Pa. 15213
{
{*****************************************************************}


{ Date: 30-Apr-80
{ Who: Don Scelza
{ Create CmdParse
{ }

{********************} Exports {********************}

Const MaxCmds = 20;                 {The most command that will fit in the table}
Type CmdArray = Array[1..MaxCmds] Of String;  {A user program declares an
                                     array of this type to use CmdParse}

Procedure CnvUpper(Var Str:String); {Convert a string to all upper case}
Function UniqueCmdIndex(Cmd:String; Var CmdTable: CmdArray; NumCmds:Integer)
                        :Integer;    {Return the index in CmdTable for the
                                     command Cmd.  NumCmds is the number
                                     of legal commands in CmdTable
                                     If Cmd was not found then return NumCmds + 1
                                     If Cmd was not unique then return NumCmds+2}
Procedure RemDelimiters(Var Src:String; Delimiters:String;  Var
 BrkChar:String);
                                    {Remove any of the characters in Delimiters
                                     from the front of the string Src, put the
 1st
                                     character not a delimiter in BrkChar}
Procedure GetSymbol(Var Src,Symbol:String; Delimiters:String;
                        Var BrkChar:String);
                               {scan the string Src looking for any of the
                                     characters in Delimiters.  Put all the
                                     characters scanned up to, but not including
                                     the delimiter in Symbol, and put the
                                     delimiter in BrkChar}

{********************} Private {********************}
```

PMD

```
module PostMortemDump;
{  Stack Trace Back Print Out for Perq Operating System
   written by John Strait
   Coppyright (C) 198Ø
   Three Rivers Computer Corporation
   Pittsburgh, Pa

This module prints a meaningful error message on errors (or when
requested by  Control-Shift D)

}


exports

const FirstRunError =  1Ø;   {Error Codes for RunTimeError, known by microcode}
      ErrDivZero =  1Ø;      {Divison by Zero}
      ErrMulOvfl =  11;      {Overflow in Multiply}
      ErrStrIndx =  12;      {String Index out of range}
      ErrStrLong =  13;      {String too long (>255 chars)}
      ErrInxCase =  14;      {Array or Case Index out of range}
      ErrSTLATE  =  15;      {Segment non resident}
      ErrUndfQcd =  16;      {Undefined Q-Code}
      ErrUndfInt =  17;      {Undefined Interrupt}
      ErrIOSFlt  =  2Ø;      {IO Segment Fault}
      ErrMParity =  21;      {Memory Parity Error}
      LastRunError =  21;

type Fault = (SegFault, StackFault, RunFault, UserFault, DumpFault);
            {Type of Fault, declaration order must match type codes
             used in System}
     ACB = record              {Activation Control Block}
           SL: integer;      {Static Link}
           LP: integer;      {Local Pointer}
           DL: integer;      {Dynamic Link}
           GL: integer;      {Global Link}
           TL: integer;      {Top of stack Pointer}
           RS: integer;      {Return Segment number}
           RA: integer;      {Return Address (offset)}
           RR: integer       {Return Routine number}
           end;

    pInteger = Integer;

    pACB = ACB;

    StackPointer = record case integer of
                     1: (IP: pInteger);  {As a virtual address}
                     2: (AP: pACB);      {As a pointer to an ACB}
                     3: (Off: integer;   {As a segment,,offset pair}
                         Seg: integer)
                     end;
    KludgeArray = array[Ø..Ø] of integer;



 procedure PMD( FaultKind: Fault; Message: string;
                E1, E2: integer; var A: KludgeArray );
 {Cause a Post Mortem Dump.  Message printed and termination conditions
  depend on FaultKind asd well as System state.  Print the Message, possibly
  with one or two parameters.  the A array is so that PMD can read the ACB of
  the caller}


private
```

## FILESYSTEM

```
Module FileSystem;

{*********************************************************************
{
{ PERQ POS File System.
{
{ Written by: John Doe  April 17, 1980
{
{ Copyright (C) 1980
{ Three Rivers Computer Corporation
{ Pittsburgh Pa.  15213
{
{*********************************************************************}

{ Most Pascal programs will use the Stream package to access the
  File System.  These procedures are used by sophisticated programs
  which need to do their own IO and buffering.
            ======NOTE==========
  This File System is a hack, and will be replaced very soon.
  Its not bad for one week of work!!!
}
{$G+}


{********************} Exports {********************}

    Const
       BlksPerFile=129;          { Blocks in each file }
       FirstBlk=0;               { Block number of the first data block }
                                 { in a file }
       LastBlk=127;              { Block number of the last data block }
                                 { in a file. }
       FIBlk=-1;                 { Block number of the File Information Block }
       NumFiles=180;             { Max number of valid files }
       BootLength = 60 + 128;    { Size of the bootstrap area on disk--the }
                                 { first n blocks on the disk.  the microcode }
                                 { boot area is 60 blocks, the Pascal boot  }
                                 { area is 128 blocks (32K). }
       StartBlk=BootLength;      { The block number of the FIBlk of the first}
                                 { user file. }
       SysFile=-1;               { File ID of the system area on disk. }
       DirFile = 1;              { File ID of the directory file }
       FNameLeng = 31;           { Max length of a file name. }
       DirPerBlk = 16;           { Number of directory entries in a block }


   Type
      DirBlk= Record                 { Record for reading disk blocks }
           Case Integer Of           { This is a Directory entry or a Block }
           1: (
               NumBlocks: Integer;   { Number of used blocks in this file }
               NumBits: Integer;     { Number of valid bits in the last block
 }
```

```
               FName: String           { File name.  A name of '' means file
                                       { not in use }
              );
           2: (
              Buffer:Array[Ø..255] Of Integer
              );
           3: (
              DirBlk: Array[1..DirPerBlk] Of String[FNameLeng]
              )
           End;

       FileID= -1..NumFiles;

       BlkNumbers= FIBlk..LastBlk;

       PDirBlk= DirBlk;


Var FSDirPrefix:String;          {Added to all file names}


Function  FSLookUp(FileName:String; Var BlkInFile,BitsInLBlk: Integer): FileID;
   {Find the existing file FileName, return its current length in blocks
    as well as the length of the last block in bits.  Function returns
    A FileID to be passed to sebsequent calls to FSBlkRead/FSBlkWrite
    A Zero is returned if the File is not found}
Function  FSEnter(FileName:String): FileID;
   {Create a new file named FileName, return a FileID}
Procedure FSClose(UserFile:FileID; Blks,Bits:Integer);
   {Close a file that has been FSLookUp/FSEntered, write the length in
    blocks and bits}
Procedure FSBlkRead(UserFile:FileID; Block:BlkNumbers; Buff:PDirBlk);
    {Read block number Block on file UserFile into buffer Buff}
Procedure FSBlkWrite(UserFile:FileID; Block:BlkNumbers; Buff:PDirBlk);
    {Write block number Block on file UserFile from buffer Buff}
Procedure FSInit;
    {Initialize the File System, Called by System}
Procedure FSDiskInit;
    {Initialize a virgin disk.
     *******WARNING**** this will destroy all data on the Disk
     Called by FileUtility}
Const
    FSDebug = False;



{********************} Private {********************}
```

STREAM

```
{$R-  shut off range checks }
{$Q-}
{      Perq Pascal Stream Package
{      Written By John Strait
{      Copyright (C) 1980
{      Three Rivers Computer Corporation
{      Pittsburgh, Pa


    This module implements formatted IO for Pascal
    It is not intended for use directly by user programs
    Pascal file IO will call these routines for you
}


module Stream;


exports


const IdentLength = 8;          { significant characters in an identifier }


type pStreamBuffer = StreamBuffer;

     StreamBuffer = record case integer of  {Funny declration so we can access
                                             packed arrays}

      0: (W: array[0..255] of integer);         { also defines size of a
 buffer}
      1: (B1: packed array[0..0] of 0..1);
      2: (B2: packed array[0..0] of 0..3);
      3: (B3: packed array[0..0] of 0..7);
      4: (B4: packed array[0..0] of 0..15);
      5: (B5: packed array[0..0] of 0..31);
      6: (B6: packed array[0..0] of 0..63);
      7: (B7: packed array[0..0] of 0..127);
      8: (B8: packed array[0..0] of 0..255)
      end;


     ControlChar = 0..31;                   {As defined by ASCII}

     FileKind = (BlockStructured, CharacterStructured);

     FileType = { file of Thing }
      packed record
       Flag: packed record case integer of
        0: (CharReady : boolean;            {Character is in File Window}
            Eoln      : boolean;            {End of Line flag}
            Eof       : boolean;            {End of File}
            NotReset  : boolean;            {false if file if a Reset has been
```

```
                                                performed on this file (for read)}
            NotOpen    : boolean;              {false if file is Open}
            NotRewrite: boolean;               {set false if a Rewrite has been
                                                performed on this file (for
write)}
            Extrnal    : boolean;              { Not Used - will be Permanent/Temp
                                                file flag}
            Busy       : boolean;              { IO is in progress }
            Kind       : FileKind);
        1: (skip1      : Ø..3;
            ReadError : Ø..7);
        2: (skip2      : Ø..15;
            WriteError: Ø..3)
        end;
       EolCh, EofCh, EraseCh, NoiseCh: ControlChar; {self explanatory}
       OmitCh          : set of ControlChar;
       FileNum         : integer;       { POS file number }
       Index           : integer;       { current word in buffer for un-packed
                                          files, current element for packed
                                          files }
       Length          : integer;       { length of buffer in words for un-
                                          packed files, in elements for packed
                                          files }
       BlockNumber     : integer;       { next logical block number }
       Buffer          : pStreamBuffer;{ I/O buffer }
       LengthInBlocks: integer;         { file length in blocks }
       LastBlockLength:integer;         { last block length in bits }
       SizeInWords     : integer;       { element size in words, Ø means
                                          packed file }
       SizeInBits      : Ø..16;         { element size in bits for packed
                                          files }
       ElsPerWord      : Ø..16;          { elements per word for packed files }
       Element: { Thing } record case integer of  {The File window}
        1: (C: char);
        2: (W: array[Ø..Ø] of integer)
        end
       end;


     ChArray = packed array[1..1] of char; {For read/write character
 array}


     Identifier = string[IdentLength];
     IdentTable = array[Ø..1] of Identifier;



var StreamSegment: integer;     { Segment buffer for I/O buffers }



 procedure StreamInit( var F: FileType; WordSize, BitSize: integer;
                          CharFile: boolean );
      {Initialize, but do not open A Stream of File F, of size
 WordSize/BitSize}
 procedure StreamOpen( var F: FileType; var Name: string;
                          WordSize, BitSize: integer; CharFile: boolean;
                          OpenWrite: boolean );
      {Open A Stream of File F, of size WordSize/BitSize, file is/is not
               CharFile and is/is not open for writing}
 procedure StreamClose( var F: FileType );
      {Close the stream F}
 procedure GetB( var F: Filetype );
      {Get an item from the file F, put it in Element}
 procedure PutB( var F: Filetype );
      {Put a item on the file F}
 procedure GetC( var F: Filetype );
      {Get a character from Text file F with Lazy evaluation}
 procedure PutC( var F: FileType );
      {Put a character from Text file F with lazy evaluation}
 procedure PReadln( var F: Filetype );
      {Perform a Read Line function}
 procedure PWriteln( var F: Filetype );
      {Perform a Write Line function}
 procedure InitStream;
      {Initialize the stream package, called by System}
 procedure StreamError( var F: FileType; E: integer );
      {Complain about a Stream Related error}


private
```

PSTRING

```
Module PERQString;


{-----------------------------------------------------------------------------
{
{ PERQ String hacking routines.
{ Written by: Donald Scelza
{ Copyright (C) 1980
{ Three Rivers Computer Corporastion
{ 160 N. Craig Street
{ Pitsburgh, Pa.  15213
{
{ File: StringRoutines.Text
{
{ Abstract:
{    This module implements the string hacking routines for the
{    Three River PERQ Pascal.
{    The routines impolemented by this module are:
{        Adjust    Concat    Substr
{        Delete    Insert    Pos
{
{---------------------------------------------------------------------------}



{~}
{ Date: 30-Apr-80
{ Who: Don Scelza
{ Changed ConCat and SubStr to use temp strings for thier hacking.
{ This will allow a use to pass the same string as input and result
{ parameters.
{ }

{ Date: 8-Apr-80
{ Who: Don Scelza
{ Created the string hacking module.
{ }


{~}

{-------------------------------------------------------------------------
{
{ Strings in PERQ Pascal are stored a single character per byte with
{ the byte indexed by 0 being the length of the string.  When the routines
{ in this module must access the length byte they must turn off range
{ checking.
{
{-----------------------------------------------------------------------}

{********************}  Exports  {********************}
```

```
Const MaxPStringSize=255;          { Length of strings}
Type PString = String[MaxPStringSize];

Procedure Adjust(Var STR:PString;  LEN:Integer);
     {Make string STR be LEN character long}
Procedure Concat(Str1,Str2:PString;  Var RetVal:PString);
     {Concatenate two strings together to form a third string}
Procedure Substr(Source:PString;  Var RetVal:PString; Index,Size: Integer);
     {Copy a portion of string Source into string RetVal starting at char
      Index Size characters long}
Procedure Delete(Var Str:PString;  Index,Size:Integer);
     {Remoce Size characters from string Str at character Index}
Procedure Insert(Var Source,Dest:PString; Index:Integer);
     {Insert the string Source at character Index in string Dest}
Function Pos(Source,Mask:PString): Integer;
     {Return the index of string Source where the first instance of Mask
      occurs in Source}


{********************}  Private  {********************}
```

## MEMORY

```
{$R-}
{$Q-}
module Memory;
{   Memory Management Routines for Perq Operating System
{   Written By John Strait
{   Copyright (C) 1980
{   Three Rivers Computer Corp
{   Pittsburgh, PA

  This module implements segment allocation and space management
  User programs needing to create segments or change default sizes
  and/or increments Import this module.
}

exports


const {$I SegNumbers.Text}      {Segment numbers of initially loadeded segments
      MemoryInBlocks =  1000;   { amount of memory on this machine }
      MaxBlocks =  400;         { maximum number of blocks in a segment }
      Two4 =  10;
      Two8 =  400;
      Two12 =  10000;
      MaxCount =  377;          { biggest useage count on a segment }
      MaxIntSize =  377;        { real sizes are numbered 0 - 255 }
      MaxExtSize =  400;        { internal sizes are 1 - 256 }
      MaxSegment =  137;        { should be 2**16 - 1 }
      SetStkBase =  60;         { StartIO code for SetStackBase }
      SetStkLimit =  120;       { StartIO code for SetStackLimit }



type Bit4 = 0.. 17;
     Bit8 = 0.. 377;
     Bit12 = 0.. 7777;
     IntSize = 0..MaxIntSize;
     ExtSize = 1..MaxExtSize;
     Address = integer;
     MemoryPosition = (Low, High);

     SegmentNumber = integer;

     SegmentKind = (CodeSegment, DataSegment);

     FreeNode = record
      N: Address;
      L: integer
      end;

     BlockArray = array[0..0] of array[0..127] of integer;

     pBlockArray = BlockArray;
```

```
    MemoryArray = record m: array[Ø..Ø] of FreeNode end;

    pMemoryArray = MemoryArray;

    Pointer = record case integer of
     1: (P: Integer);
     2: (B: pBlockArray);
     3: (M: pMemoryArray);
     4: (Offset: Address;
         Segmen: SegmentNumber)
     end;

    SATentry = packed record { Segment Address Table }
     NotResident : boolean;             { ØØ1 }
     Moving      : boolean;             { ØØ2 }
     RecentlyUsed: boolean;             { ØØ4 }
     Sharable    : boolean;             { Ø1Ø }
     Kind        : SegmentKind;         { Ø2Ø }
     Full        : boolean;             { Ø4Ø }
     InUse       : boolean;             { 1ØØ }
     Lost        : boolean;  { *** }    { 2ØØ }
     BaseLower   : Bit8;
     BaseUpper   : Bit4;
     Size        : Bit12
     end;

    SITentry = packed record { Segment Information Table }
     Increment  : IntSize;
     Maximum    : IntSize;
     Freelist   : Address;
     RefCount   : Ø..MaxCount;
     IOCount    : Ø..MaxCount;
     LockCount  : Ø..MaxCount;
     SerialLower: integer;
     SerialUpper: integer;
     DiskAddress: integer;
     NextSeg     : SegmentNumber
     end;

    SATarray = array[Ø..Ø] of SATentry;

    SITarray = array[Ø..Ø] of SITentry;

    pSAT = SATarray;

    pSIT = SITarray;

    Edge = record
            H: SegmentNumber;  { Head }
            T: SegmentNumber   { Tail }
            end;

 procedure InitMemory;    {Initialize memory mangement system, called by
 System}
 procedure MemoryError( N: integer );  {Print a meaningful message}
 procedure DataSeg( var S: SegmentNumber ); {Make this segment a Data Segment}
 procedure ChangeSize( S: SegmentNumber; Fsize: ExtSize );
 procedure CreateSegment( var S: SegmentNumber;
                          Fsize, Fincrement, Fmaximum: ExtSize );
              { Create a new segment, set Initial, Maximum and increment
 sizes}
 procedure IncRefCount( S: SegmentNumber );   { increment reference count
                                                Segment is not destroyed until
                                                RefCount is Ø }
 procedure IncIOCount( S: SegmentNumber );    { increment IO reference count
                                                Seg cannot be destroyed,
 swapped,
                                                or moved unless IOCount is Ø }
 procedure IncLockCount( S: SegmentN
umber ); { increment lock count
                                                Segment cannot be moved or
                                                 swapped unless lock count is Ø}
 procedure DecRefCount( S: SegmentNumber );
 procedure DecIOCount( S: SegmentNumber );
 procedure DecLockCount( S: SegmentNumber );
 procedure SetIncrement( S: SegmentNumber; V: ExtSize ); { Set Size Increment }
 procedure SetMaximum( S: SegmentNumber; V: ExtSize );   { Set Maximum Size }
 procedure SetSharable( S: SegmentNumber; V: boolean );  { Set Shareable flag }
 procedure SetKind( S: SegmentNumber; V: SegmentKind );  { Set Data/Code kind }
 procedure PrintTable;                                   { Print memory table }
 procedure PrintFreelist( S: SegmentNumber );    { Print Free nodes list }
 procedure MarkMemory;                             { Marks Segments already
                                                   loaded as permanent.
                                                   Called by System to mark
                                                   System segments. }
 procedure CleanUpMemory;                                { Destroy all segments
                                                           not marked.  Called
                                                           by System after User
                                                           program exits }


var SAT: pSAT;    {The System Address Table }
    SIT: pSIT;    {The System Information Table}
    First, Free, Last, Heap: SegmentNumber;
    Hole: Edge;
    State: (Scan1, Scan2, Scan3, NotFound, Found);
    StackSegment: SegmentNumber;


private
```

IO

```
{$R-}
Module IO;

{******************************************}
{ PERQ Raw IO Drivers                      }
{ Written by: Miles A. Barel               }
{ Copyright (C) 198Ø                       }
{ Three Rivers Computer Corporation        }
{ 16Ø N. Craig Street                      }
{ Pittsburgh, PA  15213                    }
{******************************************}

{*******************************}   Exports   {*****************************}

Const
        { Device Code Assignments }
        MaxUnit = 18;           { highest legal device code }
        FakeUnits = 2;          { Number of units which don't have StartIO's }
        IOStart = Ø;            { Master Z-8Ø control }
        HardDisk = 1;
        Floppy = 3;
        Speech = 4;
        IEEE488 = 5;
        Z8ØMonitor = 6;
        Tablet = 7;
        KeyBoard = 8;
        RS232In = 9;
        RS232Out = 1Ø;
        SpPutSts = 12;          { Put/Get Status }
        SpGetSts = 13;
        SpPutCir = 14;          { Put/Get from Circular Buffer }
        SpGetCir = 15;
        { Fake Units Begin Here }
        TransKey = 16;          { Translated Keyboard }
        Screen = 17;            { Screen Display }
        Clock = 18;             { Used only for Put/Get Status }
        LastUnit = Clock;       { for unit validity checking }

Type
        Double = array[Ø..1] of integer;

        IOBufPtr = IOBuffer;
        IOBuffer = array[Ø..Ø] of integer;

        CBufPtr = CBufr;
        CBufr = packed array[Ø..Ø] of char;    { same as Memory, except for }
                                               { character buffers }
        BigStr = String[255];                  { A big String }

        UnitRng = Ø..MaxUnit;

        IOStatPtr = IOStatus;
        IOStatus = record
```

```
              HardStatus: integer;            { hardware status return }
              SoftStatus: integer;            { device independent status }
              BytesTransfered: integer
          end;

       IOHeadPtr = IOHeader;
       IOHeader = record
                     SerialNum: Double;
                     LogBlock: integer;
                     Filler: integer;
                     NextAdr: Double;
                     PrevAdr: Double
                    end;

       DevTabPtr = DeviceTable;
       DeviceTable = array [UnitRng] of packed record
              CtlPtr: IOBufPtr;    { actually pointer to ChrCntlBlk or IOCB, }
                                   { but we'll coerce later }
              BlockSize: integer;     { Ø = variable size }
                                      { 1 = character device (uses circ buf)
 }
                                      { >1 = fixed blocksize (= blocksize ) }
              IntrMask: integer;      { interrupt mask bits }
              IntrPriority: integer;  { decoded iterrupt priority (Ø..14) }
              PSCode: Ø..255;         { Special code for PutStatus }
              GSCode: Ø..255;         { Special code for GetStatus }
              Name: packed array[Ø..3] of char
          end;

Const         { RS-232 Speeds }
       RS96ØØ = 1;
       RS48ØØ = 2;
       RS24ØØ = 4;
       RS12ØØ = 8;
       RS6ØØ = 16;
       RS3ØØ = 32;
       RS15Ø = 64;
       RS11Ø = 87;

Type
       Z8ØReadings = packed record       { Z8Ø Voltage/Temp Monitor Readings }
              Ground: integer;
              Volts5: integer;
              Volts12: integer;
              Minus12: integer;
              VRef: integer;
              Net: integer;
              CRTemp: integer;
              BaseTemp: integer;
              Volts55: integer;
              Volts24: integer
          end;

       Z8ØSettings = packed record      { Z8Ø Voltage/Temp Monitor Settings }
              MinGround: integer;
```

```
                MaxGround: integer;
                Min5: integer;
                Max5: integer;
                Min12: integer;
                Max12: integer;
                MinMinus12: integer;
                MaxMinus12: integer;
                MinVRef: integer;
                MaxVRef: integer;
                MinNet: integer;
                MaxNet: integer;
                MinCRTemp: integer;
                MaxCRTemp: integer;
                MinBaseTemp: integer;
                MaxBaseTemp: integer;
                Min55: integer;
                Max55: integer;
                Min24: integer;
                Max24: integer
            end;


        DevStatusBlock = packed record
                ByteCnt: integer;       {   of status bytes }
                case UnitRng of
                    KeyBoard,
                    Tablet,
                    Clock: (DevEnable: boolean);

                    RS232In,
                    RS232Out: (RSRcvEnable: boolean;
                               RSFill: Ø..127;
                               RSSpeed: Ø..255;
                               RSParity: (NoParity, OddParity, IllegParity,
                                          EvenParity);
                               RSStopBits: (Syncr,Stop1,Stop1x5,Stop2);
                               RSXmitBits: (Send5,Send7,Send6,Send8);
                               RSRcvBits: (Rcv5,Rcv7,Rcv6,Rcv8));
                    Z8ØMonitor: (Z8ØEnable: boolean;
                                 Z8ØFill: Ø..32767;
                                 case boolean of { Get or Put; true = Get }
                                   true: (       { Get Status }
                                          GetRead: Z8ØReadings;
                                          GetLimits: Z8ØSettings
                                         );
                                   false:(       { Put Status }
                                          PutLimits: Z8ØSettings
                                         ));
                    Floppy: (case integer of { Get or Put }
                              1: (           { Get Status }
                                  FlpUnit: Ø..3;
                                  FlpHead: Ø..1;
                                  FlpNotReady: boolean;
                                  FlpEquipChk: boolean;
                                  Flp
Se   End: boolean;
                                  FlpIntrCode: Ø..3;
                                 case integer of
                                  1 {IORead, IOWrite, IOFormat}:
                                   (FlpMissAddr: boolean; { in data or header }
                                    FlpNotWritable: boolean;
                                    FlpNoData: boolean;
                                    FlpFill1: Ø..1;
                                    FlpOverrun: boolean;
                                    FlpDataError: boolean; { in data or header
 }
                                    FlpFill2: Ø..1;
                                    FlpEndCylinder: boolean;
                                    FlpDataMissAddr: boolean; { in data }
                                    FlpBadCylinder: boolean;
                                    FlpFill3: Ø..3;
                                    FlpWrongCylinder: boolean;
                                    FlpDataDataError: boolean; { in data }
                                    FlpFill4: Ø..3;
                                    FlpCylinderByte:   Ø..255;
                                    FlpHeadByte:       Ø..255;
                                    FlpSectorByte:     Ø..255;
                                    FlpSizeSectorByte: Ø..255
                                   );
                                  2 {IOSeek}:
                                   (FlpPresentCylinder: Ø..255
                                   )
                                  );
                              2: (           { Put Status }
                                  FlpDensity: Ø..255; { single = Ø,
                                                        double =  1ØØ }
                                  FlpHeads: Ø..255;   { 1 or 2 heads }
                                  FlpEnable: boolean
                                 );
                              3: (           { Byte access }
                                  FlpByte1: Ø..255;
                                  FlpByte2: Ø..255;
                                  FlpByte3: Ø..255;
                                  FlpByte4: Ø..255;
                                  FlpByte5: Ø..255;
                                  FlpByte6: Ø..255;
                                  FlpByte7: Ø..255
                                 )
                            )
                end;


Var
        DevTab: DevTabPtr;              { pointer to system device table }
        CtrlCPending: boolean;          { if one C has been typed }
        CtrlSPending: boolean;          { if S has halted screen output }

Procedure InitIO;
Function IOCRead(Unit: UnitRng; var Ch: char): integer;
                                            { read a character from a }
```

```
                                               { character device and return }
                                               { status: IOB no char available }
                                               {         IOC character returned }
Fu tion IOCWrite(Unit: UnitRng; Ch: char): integer;
                                               { write Ch to character device }
                                               { and return status:           }
                                               {       IOB buffer full        }
                                               {       IOC character sent     }
Function IOSWrite(Unit: UnitRng; Str: BigStr): integer;
                                               { write Str to character device }
                                               { and return status:           }
                                               {       IOB buffer full        }
                                               {       IOC character sent     }

Type
    IOCommands = (IOReset, IORead, IOWrite, IOSeek, IOFormat,
                  IODiagRead, IOWriteFirst, IOIdle, IOWriteEOI, IOConfigure);

Procedure UnitIO(Unit: UnitRng;                    { IO to block structured }
                 Bufr: IOBufPtr;                   { devices }
                 Command: IOCommands;
                 ByteCnt: integer;
                 LogAdr: double;
                 HdPtr: IOHeadPtr;
                 StsPtr: IOStatPtr);

Procedure IOWait(var Stats: IOStatus);             { hang until I/O completes }
Function IOBusy(var Stats: IOStatus): boolean;     { true if I/O not complete }
F cedure IOPutStatus(Unit: UnitRng; var StatBlk: DevStatusBlock);
                                               { Set status on device Unit }
Procedure IOGetStatus(Unit: UnitRng; var StatBlk: DevStatusBlock);
                                               { Reads status on device Unit }
Procedure IOBeep;                              { You guessed it, BEEP! }
Function GetOct: integer;                      { read an octal   from Keyboard }



{ tablet/cursor procedures }


Ty      CurMode = (AbsCursor, RelCursor, UserCursor, OffCursor);
        CursorPattern = array[0..63,0..3] of integer;
        CurPatPtr = CursorPattern;


Var     TabRelX, TabRelY: integer;         { tablet relative coordinates }
        TabAbsX, TabAbsY: integer;         { tablet absolute coordinates }
        TabFinger: boolean;                { finger on tablet }
        TabSwitch: boolean;                { switch pushed down }
        DefaultCursor: CurPatPtr;          { default cursor pattern }


Procedure IOSetCursorMode( M: CurMode );           { set CursorMode }
Procedure IOLoadCursor(Pat: CurPatPtr; pX, pY: integer);
                                                   { load user cursor pattern }
```

```
Procedure IOMoveCursor(crsX,crsY: integer);     { move the hardware cursor }
Procedure IOReadTablet(var tabX, tabY: integer);{ read tablet coordinates }


Var     IOInProgress: boolean;          { false when speech is active }

{*******************************}    Private    {*****************************}
```

## LINEDRAW

```
{$R-}
M[illegible]LE LineDraw;

{ A simple Imortable interface to the line drawing qcode
  Written by John Strait
  Copyright (C) 1980
  Three Rivers Computer Corporation
  Pittsburgh, PA  15213
}

EXPORTS

  Type LinePtr = LineArray;
       LineArray = Array[0..0] of integer;  { a bit map to draw lines in }
       LineStyle = (DrawLine,EraseLine,XorLine);

  Procedure Line( Style: LineStyle;
                  X1, Y1, X2, Y2: integer; Origin: LinePtr );

PR[illegible]ATE
```

MOVEMEM

```
module MoveMem;
{  'elper module for Memory,  MUST be a separate segment
   Not needed by any User Programs
   Written by John Strait
   Copyright (C) 1980
   Three Rivers Computer Corporation
   Pittsburgh, PA
}

exports


imports Memory from Memory;
imports Raster from Raster;


procedure CopySegment( SrcSeg, DstSeg: SegmentNumber; NewDstBase: integer );


pr  ate
```