THE WINDOW MANAGER

December 7, 1984

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

Accent is a trademark of Carnegie-Mellon University.

Accent and many of its subsystems and support programs were originally developed by the CMU Computer Science Department as part of its Spice Project.

The major part of the design and most of the implementation of the window manager was done by Brad Myers of PERQ Systems Corporation. The design grew out of his discussions with many people, including Gene Ball, the window manager designers at International Computers Limited, and various interested parties at CMU and PERQ Systems Corporation. Amy Butler and Dave Golub of PERQ Systems were instrumental in completing the window manager's implementation.

The information in this document is subject to change without notice and should not be construed as a commitment by PERQ Systems Corporation. The company assumes no responsibility for any errors that may appear in this document. PERQ Systems Corporation will make every effort to keep customers apprised of all documentation changes as quickly as possible. The Reader's Comments card is distributed with this document to request users' critical evaluation to assist us in preparing future documentation.

PERQ, PERQ2, LINQ, and Qnix are trademarks of PERQ Systems Corporation.

Table of Contents

1. Theory

The Accent window manager supports the Covered Window Paradigm in which the screen area is divided into various rectangular areas, called windows, which may be overlapping. The graphics and text in one window do not affect the graphics and text in other windows. The window manager supports graphic operations (which includes text) to windows even if the window is partially covered by other windows, although only the uncovered portions of the graphics will actually be displayed on the screen.

The window manager also supports icons, which are small pictures that represent windows. The icons in the window manager provide useful information about the state of the process running in each window. An icon exists for each window, which allows the icons to be used for controlling windows and getting information about the processes running in them whether they are on or off the screen. All of the icons are collected together in one window so that they can be manipulated as a group.

The window manager will support either the portrait or landscape monitor, as well as all of the tablets that can be used with the PERQ workstation. It currently will not handle the color monitor.

This document describes how programmers can incorporate the window manager into application programs. A separate document, "User's Guide to the Window Manager" in the *Accent User's Manual*, contains an introduction to the Accent window manager and its basic features from an end-user's viewpoint.

This document is organized as follows:

Chapter 2 gives details on how to use the window manager in a program;

Chapter 3 contains the type definitions for the window manager;

Chapters 4 through 13 contain the routine definitions;

Chapter 14 contains information on key translation definitions, creating a text file for a key translation table, and the use of such tables in application programs.

Chapter 15 lists errors and exceptions;

and Chapter 16 is a sample program.

The window manager's name is Sapphire, so you will find the words "Sapphire," "Saph," and "Sapph" throughout the window manager program.

If you incorporated the window manager in Accent Release S5 into your programs, please note the changes to the window manager that are listed in the release notes accompanying Release S6. Changes to Keytran since Release S5 are listed in Chapter 14.

2. Use

2.1. Getting Started

To assist you in incorporating the window manager into your own program, Chapter 16 lists a sample application program. In creating your own program, remember to do the following:

1. In order to use the window manager, the application program must import Sapph from SapphUser. This will import other necessary modules, including the interface to Matchmaker, the Accent remote procedure call generator.

2. If handling emergency messages (discussed in Chapter 9), import SaphEmrServer from SaphEmrServer and SaphEmrExceptions from SaphEmrExceptions.

3. The file SapphFileDefs contains font and cursor definitions you may wish to use. A second file, SapphLoadFile, contains routines to load, use, and alter cursor files, font files,, and picture files.

4. Finally, the information needed for manipulating keytranslation tables is in the two additional files KeyTran and KeyTranDefs.

All of these modules are available from the Pascal library located in the LibPascal directory.

2.2. Message Passing

The passing of messages in Accent in general is explained in "Theory of Operations" in this manual. Message passing is transparent to the user. For the most part, it will also be transparent to you as you write an application program. You should merely import modules containing the procedure headers and call the procedures; the message passing will be handled by Matchmaker, the Accent remote procedure call generator. For further information see the document "Matchmaker: The Accent Remote Procedure Call Language" in the *Accent Languages Manual*.

The only instance in which you will need to deal with message passing is if you wish to use emergency messages (explained in Section 2.5 and Chapter 9). Because Matchmaker doesn't currently support emergency messages, you will receive a message from the window manager. You will have to extract the information needed to handle a resized window or exposed window exception.

2.3. Data Types

The window manager supports a variety of data types, and there are separate procedures for dealing with each type. The data types that describe rectangular areas are windows, viewports, and regions. There are other data types for cursors, fonts, etc.

For many of the routines of the window manager, there are special values for parameters. The predefined constants are all defined in SapphDefs.pas. For example, NULLViewport and NULLWindow are special values for viewports and windows. DONTCARE, UNCHANGED, and BOTTOM are special values for integers that can be used in various places as appropriate. Chapter 3 explains what these values do in the procedures that take them.

2.3.1. Windows vs. Viewports

When an application wants an area in which to do some graphics, it has three choices: it can use the window provided by its shell, it can create a new window, or it can create a new viewport. This section will discuss the differences between windows and viewports and will describe how to create each.

A window is a rectangular area. It may be on or off the screen. Windows can have borders, title lines, and icons, and they can be manipulated by the user. The window package simply uses VP (viewport) graphics primitives to implement its graphics. The handling of the mouse user interface is done entirely by the window manager. Thus, most applications that want an area will use windows. Each window is composed of one or more viewports. A viewport is the unit of graphical display in the window manager. Viewports will typically only be used when an existing window is to be sub-divided into various sections. In a two viewport window one viewport is used to display the title line and borders of the Window. The second viewport, created just inside the first viewport, is passed to the user to be used for all graphics in the window. When the application wants to do graphics in a window, it uses the viewport returned by the CreateWindow call (described below). It is the viewport layer that knows how to do graphics to partially covered areas.

All graphics operations are clipped to the boundaries of a viewport. If an operation were to extend out of the viewport's borders, it would be simply clipped (cut off at the border). Thus, for example, it is impossible for an application to draw over the borders of a window since it is provided with only the inner viewport that does not include the border areas.

A viewport can be divided into a number of regions. Regions do not clip graphics operations and are only used for tracking and

interpreting keyboard actions. An application can define a number of regions and associate different cursors with different regions, for example. The window manager would then change the cursor picture when the puck was moved from one region to another. This saves processor cycles since the application program does not need to wait in a loop constantly checking the cursor to see if the picture needs to be changed.

Windows are the listeners. Pressing in a window makes the window the listener, at which time the window's border changes to the listener border.

Both windows and viewports are hierarchical. A window or a viewport can have sub-windows or sub-viewports that are constrained to lie entirely within their parents' borders. Just as all windows and viewports are clipped to the boundaries of the screen, all sub-viewports are clipped to the boundaries of their parent viewports (similarly with windows). If an application creates a window as a child of another window, the user will be allowed to move it around inside the other window, but it will not be possible to move it outside the parent window. However, a window that is created as a child of the special type FULLWINDOW can be moved anywhere on the screen.

2.3.1.1. Windows

The procedure CreateWindow is used to create windows. Once a window is created, it can be modified to have a different position and rank. The rank of a window or viewport is its ordering in the "Z" direction (away from you) and determines how covered a window is. The lower the rank, the closer to the viewer (and the less covered) a window is. In addition to creating and modifying a window, there are also operations to delete them, change the title, and control the pictures presented in the icon.

2.3.1.2. Viewports

A viewport can be created, modified or deleted, or graphics can be done in it. (Do not modify or delete the viewport returned by the creation of a window.) Each window or viewport has exactly one owner which is the process that calls the create operation. When the owner process dies, the windows and viewports owned by it are automatically deallocated. The owner can pass the windows and viewports to other processes in messages if it is careful to designate that they are implemented as Ports. If the owner wants some other process to be the owner, it can simply pass the ownership rights with the port in a message.

2.3.2. Regions, Cursors, and Tracking

The viewport layer handles all mouse tracking. Most applications that use cursors will need more than one (the window manager itself uses over 20, for example), so it is efficient to store them together. Therefore, the cursors in the window manager all come in groups called CursorSets. (In this document "cursor" will be used to mean the picture that usually follows the mouse on the screen.) The first block of a CursorSet describes the group and has the x and y offsets for all cursors in the set. The individual cursors are addressed by number.

A CursorSet can be read from a file using the procedure LoadVPCursors which is located in the user library module SapphLoadFile. CursorSets can be most easily created using the CursDesign program. They have a special format (which is different from all previous cursor formats and is described in the user library module SapphFileDefs). The first block of a CursorSet describes the CursorSet as a whole. CursorSets typically have the extension ".SCursor". Cursors can be loaded from the disk using the procedure LoadVPCursors. This procedure returns NullPort if the file cannot be found. Cursor sets may be created from any 64x64 bitmap data. A bitmap of a

single cursor or bitmaps of the entire set can be extracted from a cursor set. An individual cursor may be altered, either by changing the bitmap or by changing the offset. Accent's default cursors can be changed by editing the file Winman.SCursor, using the CursDesign program.

2.3.2.1. Regions

A viewport can be divided into a number of "regions" each with its own cursor. Regions are different from viewports in that they do not clip graphics. Therefore, regions can be arbitrarily overlaid on a viewport. Regions are defined by number. The first two are special regions, VPREGION for the entire inside of a viewport, and OUTREGION for the entire outside of a viewport. When created, viewports are automatically given these two regions. Applications can define their own regions by pushing them with the procedure PushRegion which takes the viewport, a region number and the coordinates of the region in the viewport. The application program is responsible for maintaining the region numbers and remembering which is which. A region number already in use can be pushed and the newest one will take precedence. VPREGION and OUTREGION can also be pushed, but for these the coordinates are ignored. DeleteRegion removes the most recently added region of the specified number. If there is only one region by that number, then the region number becomes illegal. It is a bad idea to delete the last VPRegion or OUTRegion. A region's coordinates may be changed using ModifyRegion.

2.3.2.2. Cursors

Once a region has been defined, a number of different properties for the region can be set. The procedure SetRegionCursor allows the cursor for a region to be defined. (See the complete description in Chapter 10.)

2.3.2.3. Tracking

The way the cursor tracks while the region is active can be controlled using the procedure SetRegionParms. The tracking parameters allow for relative or absolute tracking, for the cursor to be restricted to stay inside some box, and for the cursor to be restricted to a particular grid.

Using different regions with different cursors, gridding, etc., should allow most applications to simply set up a set of regions and then read the cursor position only when there is an interesting event. Hopefully, few applications will need to turn tracking off and wait in a loop (handling tracking themselves) since this infringes on the user-interface of the window manager and will be much less efficient.

2.4. Graphics Primitives

All graphic operations are clipped to the boundaries of the viewport in which they are done. In addition, only the visible (uncovered) portions of the operations will be done on the screen. If the viewport has memory, the covered portions will be updated in off-screen memory; otherwise, the operation will simply not be done. If some portion of the operation requires information that is not available (for example, RasterOp-ing from a covered portion of the source), the application will be notified of the sections that need to be regenerated in the same manner as if the portions had become uncovered. The special protected rasterOp primitives provided by the Kernel will be used to increase efficiency. If an operation is done to a window that is not covered, these fast primitives will be used. If the window is covered, the Kernel primitive will fail and the window manager routine will have to be called since only the window manager knows how to do these operations when parts of the viewports are clipped. This optimization, however, is entirely invisible to most applications since it is added at the interface to Matchmaker.

The window manager has no notion of a current position or current color, so all operations take a full coordinate specification. In addition, the window manager does not implement a character cursor; this has to be handled by the application. The window manager currently does not support any colors.

2.4.1. Moving the Contents of a Viewport

VPROP and all other graphics primitives are optimized for the case where the viewport is not covered. In this case, the operation is done directly by the Kernel without process swapping or message passing. If the viewport is covered, however, the operations cannot be handled by the Kernel so a message is passed to the window manager where it will be performed. The window manager call for VPRop is ViewRop. Similarly, for the other graphics primitives named VP---, the window manager call is View---. If the View--- procedure is called directly, this will be perfectly correct; it will just be slower when the viewport is not covered.

2.4.2. Using Fonts

Fonts in the window manager are represented as Viewports which can be easily deallocated when desired by using DestroyViewport. Two routines return the viewport that represents a font. The first, LoadFontFile, comes from the module SapphLoadFile in the user library and takes a font file name. The second, LoadFontData, is a window manager call that takes data in the form of type 'FontMap.' The special font viewport that is returned can then be used in the routines that draw text in a viewport. Both routines return NULLViewport if the font name is not found. A third procedure, SetFontChar, allows you to replace one character at a time in an existing font.

2.4.3. Writing Text

The three text writing procedures (VPString, VPChar, and VPChArray) come in two forms. The first form (using the name VP--- given above) returns information about the text drawn. For example, VPString returns the number of characters that were printed and the pixel after the last character. The second form of the procedures (using the name VPPut---, as in VPPutString) does not return any information. This will be faster since no return message needs to be generated. This also means, however, that error messages cannot be reported. If there is an error in a procedure of the second type, then the text is simply not displayed.

Currently there is a restriction on the use of VPString and VPChArray. The height of a font that VPString and VPChArray will handle is set at 45 pixels; therefore these procedures will return an error if they are used with a taller font. VPChar, however, will work with any size font.

2.4.4. Transferring Contents to and from Virtual Memory

There are other graphics primitives for transferring the contents of a viewport to and from virtual memory (for reading and specifying a viewport's contents from a program). They are PutViewportBit, GetViewportBit, GetViewportRectangle, and PutViewportRectangle.

2.5. Emergency Messages

There are two emergency messages that may be generated for any viewport. One of these is sent to the application when the viewport is modified. This may happen, for example, if the viewport is part of a window that is explicitly modified by the user via the window manager's user interface. This emergency message may be converted into a Pascal exception using the interface to Matchmaker, defined in SaphEmr.Defs.

If a portion of a viewport is uncovered for any reason, that portion will have to be regenerated. The window may have become less covered either by the removal of some other window or by the window being brought to the top. The window may have been moved from partially off screen to more on screen or it may have been brought from fully off screen. The window may have been grown from a smaller size to a bigger size. Finally, a RasterOp may have been done where a part of the source that was covered was placed in an exposed portion of the destination. If the viewport has memory, then the picture can simply be recalled from the backup memory. Most viewports will not have memory, however, so some program will have to be notified that the viewport needs to be refreshed. Even if the viewport has memory, it may still get this exception if the Viewport is grown to a bigger size since the picture for the new area is not available. The emergency message may be converted into an exception in the same manner as the message for viewport size changed. One of the parameters of this message is a variable length array of rectangles. This includes all of the rectangles that need to be refreshed for the viewport. The application can either update only the rectangles in the list, or it can simply refresh the entire viewport. The rectangle list is allocated using ValidateMemory so the application should be sure to deallocate the memory using InValidateMemory as follows (where ra is the rectangle array):

```
gr := InValidateMemory(KernelPort, recast(ra, VirtualAddress),
        wordsize(RectArray)*2);
```

In order to get around current restrictions in Matchmaker, the interface files were hand edited. The SERVER side (SaphEmrServer.pas) is used by the application program. It exports a procedure (SaphEmrServer) which can be used on the incoming emergency message to convert it into one of the two exceptions (which currently are defined in SaphEmrExceptions.pas). The USER side, SaphEmrUser.Pas, is used at the window manager end to actually generate the

emergency messages (note that this is backward).

2.6. Listener

The listener is the window or process that is currently getting keyboard typing. The window that is the listener is marked with a special border.

In order to be the listener, a window must first be enabled. Enabling a window will allow the user to point at the window to make it the listener. This is accomplished with EnableWinListener. If you plan to translate any events received while the window is listener, you must also have a valid key translation table. You can load your own table using routines from the user library module Keytran.

The user may make an enabled window the listener at any time by using the window manager user interface commands, such as pressing a mouse button while the cursor is in the desired window.

**Caution--Typically, application programs should never set the listener since users will choose which window should be the listener, using the window manager. However, in the event the application does wish to change the listener window, it should call MakeWinListener, which makes the window the listener. The danger in doing this is that, due to synchronization, it will never be possible to make sure the application program is not overriding an explicit change-listener command given by the user or vice versa.

2.7. Key Translation

The purpose of key translation is to translate keyboard events and mouse presses into commands that can be executed by the application, using the routines described in Chapter 14 of this document. Translation tables are required and can be created, added to, changed, or deleted using these routines.

Refer to Chapter 12 on input for information on untranslated events, and to Chapter 14 for information on using the key translation facility.

3. Type Definitions

This chapter contains the type definitions used with the Window Manager. The following chapters contain the routine definitions, organized according to their function.

The definitions throughout this document are given in Pascal. If you are programming in the C language, please refer also to the document "C System Interfaces" in the *Accent Languages Manual*. If you are programming in the Lisp language, see the document "Lisp Interaction with the Accent Operating System" in the *Accent Lisp Manual*. When FORTRAN becomes available under Accent, the definitions will be the same as in the C language.

3.1. Module SapphDefs

MODULE SAPPHDEFS (in file SapphDefs.pas in the Pascal library)

This module contains exported type definitions for the Sapphire Window Manager. This includes the exports for both the Window manager layer and the Viewport layer. This type module is imported by both the server and the user side of the interface.

Types:

Window — A rectangular area that is controllable by the user and which has an icon and an (optional) border and title line. A window is composed of two viewports; the outer one kept private by the window manager used to hold the title and border, and an inner one passed to the application to fill with graphics or text.

Viewport — A rectangular display area which may be on the screen. The

coordinate system of a viewport is fixed with 0,0 at the upper left and is linear with the pixels on the screen. All graphic operations are clipped to a viewport's boundaries and viewports are the unit of refreshability on the screen. A viewport may have "children" sub-viewports. Graphic operations to sub-viewports are clipped to the boundaries of the parent viewport.

Rectangle — Defined by its upper left corner and a width and height. The upper left corner is usually relative to some viewport.

Font — A standard character set defined by the Perq standard format. Fonts are represented externally as viewports.

Cursor — (1) the picture that follows the mouse on the screen.

(2) any symbolic picture which may at some time be connected to the mouse. A Cursor is 56 bits wide and 64 bits tall. Cursors are only accessible to the users in CursorSets.

CursorSet — A collection of cursors. There may be one or many cursors in a set. On the disk, files containing CursorSets usually have the extension ".SCursor". Each cursor has associated with it the origin or offset for the point. CursorSets can be conveniently be created using the latest version of CursDesign.

Region — A subset of a Viewport. Regions are used for cursor tracking and key translation.

EventRec — An untranslated event generated directly by the user by typing on the keyboard or pressing a mouse button, consisting of a viewport, a region, a keystate code, and a pointer position.

```
------------------------------------------------------------------------}

{$Version V1.4 for Accent}
{-------------------------------------------------------------------------
{\\\\\\\\\\\\\\\} EXPORTS {///////////////}


imports AccentType from AccentType;

{********************  VIEWPORTS ********************}

{------ Exported Constants ------}
const

  VPREGION = 1;
  OUTREGION = 0;

  UNCHANGED = -32001;
  OFFSCREEN = -32002;
```

```
  DONTCARE = -32004;
  BOTTOM = 32000;

  NULLViewPort = NullPort;

  MaxNumRectangles = 256 div 4;
                     {number that can fit in a page}

  SysFontName   = 'Fixl3.Kst';
  SysFontHeight = 13;
  SysFontWidth  = 9;

{------ EXPORTED types ------}

Type
  VPStr255 = string[255];

  Viewport = Port;

  CursorSet = Port;

  Rectangle = record
               lx, ty, w, h: integer;
              end;

    {used to control the cursor}
  CursorFunction = (cfScreenOff, cfBroken, cfOR,
                    cfXOR, cfCursorOff);

    {used in calls to GetKeyEvent to control waiting}
  KeyHowWait = (KeyWaitDiffPos, KeyDontWait, KeyWaitEvent);

    {used in ViewLine}
  LineFunct = (DrawLine, EraseLine, XORLine);

    {used in ViewColorRect}
  RectColorFunct = (RectBlack, RectWhite, RectInvert);

    {used in ViewRop, ViewStrArray, etc.}
  RopFunct =
   (RRpl,      { Destination gets source                       }
    RNot,      { Destination gets NOT source                   }
    RAnd,      { Destination gets Destination AND source       }
    RAndNot,   { Destination gets Destination AND (NOT source) }
    ROr,       { Destination gets Destination OR source        }
    ROrNot,    { Destination gets Destination OR (NOT source)  }
    RXor,      { Destination gets Destination XOR source       }
    RXNor);    { Destination gets Destination XOR (NOT source) }

Type
  VPIntegerArray = Array[0..0] of Integer;
  pVPIntegerArray = ^VPIntegerArray;

    {used in ViewCharArray}
  VPCharArray = Packed Array[0..1] of Char;
  pVPCharArray = ^VPCharArray;
```

```
    {used in exception for viewport becoming exposed}
  RectArray = Array[1..MaxNumRectangles] of Rectangle;
  pRectArray = ^RectArray;  { array of 64 rectangles }
                            { (one page's worth) }

  VPPortArray = Record
                  num: Integer;
                  ar: Array[0..0] of Port;
                end;
  pVPPortArray = ^VPPortArray;


{********************  KEY DEFINITIONS ************************}

Const

  { User Input events for the 3RCC Perq }
  { 0 — 127     — down key transition w/o CNTRL,
                    mapped to ASCII }
  { 128 — 255   — down key transition with CNTRL,
                    ASCII for non-control }

  { special key tran codes.  Will set special bit.
  { used for mouse button transitions and other special ones }

  cdYellowDown = 256;       { or pen button }
  cdWhiteDown = 257;
  cdBlueDown = 258;
  cdGreenDown = 259;
  cdYellowUp = 260;
  cdWhiteUp = 261;
  cdBlueUp = 262;
  cdGreenUp = 263;

  cdRegionExit = 264;       { generated when cursor leaves
                              a region }
  cdTimeout = 265;          { event generated for timeouts }
  cdPosResponse = 266;      { response to Position request
                              if same position}
  cdDiffPosResponse = 267; { response to Position request
                              if diff position}
  cdNoEvent = 268;          { response to Position request
                              if not listener}
  cdListener = 269;         { generated when new viewport is
                              listener}

  MaxCode = 127 + 16;       { up to 16 specials, currently
                              have 14}


  { transition as stored in untranslated Event }
  { ** WARNING ** the fields must overlay correctly!! }

type
```

```
  KeyState = packed record case integer of
      1: (code:     0..#177;  { key character coding }
          cntrlon:  boolean;  { true for keyboard chars,
                              if cntrl key was down}
          special:  boolean;  { true for transitions not
                              generated by keyboard, i.e.
                              buttons or regionExit, etc. }
          EscCl:    0..#7;    { current escape class -- added
                              in KeyTran. Always zero from
                              Sapphire. }
          mbuttons: 0..#17);  { state of mouse buttons after
                              event }

      2: (fullCode: 0..#777;  { code and special and control }
          escClAndMButtons: 0..#177);  { the rest }
      3: (ks: integer);
    end;

{     EventRec:
       an untranslated event generated directly by the user by
       typing on the keyboard or pressing a puck button,
       consisting of a keystate code, a region, a viewport,
       and a pointer position.}

  EventRec = record {one untranslated event-not packed since
                     goes into a message}
               vp     : Viewport;
               x,y    : Integer;  { relative to VP }
               region : Integer;
               code   : KeyState;
             End;

{************************** WINDOWS ***************************}

Const BorderOverhead     = 5;
      TitleOverhead      = SysFontHeight + 6;

      LandScapeBitWidth  = 1280;
      PortraitBitWidth   = 768;
      LandScapeBitHeight = 1024;
      PortraitBitHeight  = 1024;

      TitStrLength       = LandScapeBitWidth div SysFontWidth;

      NullWindow         = NullPort;
      ASKUSER            = -32005;
      MaxCoord           = 16000;  {largest legal window coord}
      MinCoord           = -16000; {smallest legal window coord}
      MaxSize            = 16000;  {largest legal window size}

{--- Icons ---}
      NumProgressBars    = 2;
      ProgressInTitle    = 1;  {the UtilProgress nest level that
                                is in the title line}
      IconWidth          = 64;
      IconHeight         = 64;
```

```
    ProgStrLength      = (IconWidth - 2*BorderOverhead)
                              div SysFontWidth;

{--- Exported Types ---}

Type Window = Port;

    TitStr = String[TitStrLength];
    ProgStr = String[ProgStrLength];

    pWinNameArray = ^WinNameArray;
    WinNameArray = Array[0..0] of ProgStr;  {vbl length array}
```

3.2. Module SapphFileDefs

MODULE SAPPHFILEDEFS (in file SapphFileDefs.pas in the Pascal library)

This module contains the type definitions for Viewports, available to Applications. Extracted from VPDefs.

```
------------------------------------------------------------------------}

{$Version V4.5 for Accent}


{\\\\\\\\\\\\\\\\\\\\} EXPORTS {////////////////////////}


imports AccentType from AccentType;
imports SapphDefs from SapphDefs;

{***************** Font Definitions ********************}

Const FontWordWidth = 48;       {word width for all fonts}
      FontHeadOverhead = #404; {number of words in the
                                  header for fonts}

Type  FontMap = ^FontMapRec; {font format on disk and in memory}

      FontMapRec = packed record {header portion of a font}
        Height:     integer;      {Height of the KSet }
        Base:       integer;      {distance from top of
                                   characters to base-line}

        { array indexed by ASCII of character descriptors}
        Index: array [0..#177] of packed
          record
            Offset: 0..767;     { position of character
                                  in patterns }
            Line:   0..63;      { Line of patterns containing
                                  char }
            Width:  integer;    { Width of the character }
```

```
        end;

      Filler: array[0..1] of integer;

      Pat: Array [0..0] of integer;  { patterns go here }
      end {FontMapRec};

{********************  CURSOR DEFINITIONS **********************}

Pattern =  ^PatternMap;
PatternMap = array [0..63 {y}, 0..3 {x}] of integer;
                                        { 64 x 64 bit map }

pCursorArrayRec = ^CursorArrayRec;
CursorArrayRec = record  { Multiple cursors as a group read
                           from a file }
                 firstBlock: Packed Record
                              numCursors: Integer; { 1..127 }
                              filler: Integer;
                              offsets: Array[0..126] of
                                        record
                                          x: Integer;
                                          y: Integer;
                                        End;
                            End;
                  cursors: array[0..0] of PatternMap;
                                       {vble length array}
                end;

type OffsetPairArray = array[0..126] of
                       record
                         x: integer;
                         y: integer;
                       end;
     pPatternMapArray = ^PatternMapArray;
     PatternMapArray = array[0..0] of PatternMap;
```

3.3. Module KeyTranDefs

MODULE KEYTRANDEFS (in file KeyTranDefs.pas in the Pascal library):

This module contains the definitions for the Keytran file passed between Accent programs.

```
const
      KeyTran_Error_Base = 4600;

      ErElementNotFound   = KeyTran_Error_Base + 1;
      ErIllegalKeyVersion = KeyTran_Error_Base + 2;
      ErNoFreeSpaces      = KeyTran_Error_Base + 3;
      ErHashZero          = KeyTran_Error_Base + 4;
      ErNotValidTable     = KeyTran_Error_Base + 5;
```

```
{------ Exported Constants for key translations ------}

  WILDREGION = 31;  {no matter what region}

  cChCmd = 0;  {special reserved command numbers}
  cNoCmd = 1;


  KeyTranVersion = 4;     {Version of the format for the
                             Keytran file}

  { MAXMAP & WILDREGION must be packable in 14 bits }
  MAXMAP = 511;
  MAPNIL = 0;

type

  KeyKind = (Standard, Control, Mouse, NonStandard);

   {keykind tells how complex the event must be before a lookup
    is done in the table.  Mouse means mouse or control.}

  EscKind = (CodeNormal,    { return code & change Escape class
                              to 0 }
             CodeSame,      { return code & leave Escape class
                              unchanged }
             EscReturn,     { change escape class to 'chval',
                              return cmd }
             EscNoop        { change escape class to 'chval',
                              return NoCmd }
            );

  { Types of key translation:
  {  1) Standard--  no explicit table entry exists,
  {                 char code & <CNTRL> used to construct 'ch',
  {                 cmd = cChCmd
  {                 mouse transitions, etc. -> cmd = cNoCmd,
  {                 EscClass becomes 0
  {  2) CodeNormal--find table entry that matches key event,
  {                 escty=CodeNormal
  {                 return cmd, ch  pair from that entry,
  {                 EscClass becomes 0
  {  3) CodeSame--  returns cmd,ch from matching entry,
  {                 escty=CodeSame,
  {                 EscClass doesn't change
  {  4) EscReturn-- matching entry has escty=EscReturn,
  {                 return cmd,ch  and
  {                 EscClass becomes ord(ch)
  {  5) EscNoop--   entry with escty=EscNoop,
  {                 EscClass becomes ord(ch),
  {                 return cNoCmd
  {  6) GetArg--    entry with escty=EscNoop,
  {                 EscClass := ord(ch) (probably 0),
  {                 CmdState := cmd, returns cNoCmd
  {                 next key,  returns 'ch' from that entry,
  {                 'cmd' from GetArg
```

```
  {----------------------}

KeyMap = packed   {entries in the table}
    record
      hashkey:  KeyState;   { transition that this entry
                              describes }
      cmd:  0..255;         { cmd code that it returns }
      chval: char;          { character that it returns;
                              will be 0..7 if
                              escTy = EscReturn or EscNoop }
      escty: EscKind;       { type of entry }
      next:  0..MAXMAP;     { 0 means NIL }
      region: 0..WILDREGION; { region for this entry; used
                               with hashKey and current region
                               to determine if success }
    end;

KeyTransTab = packed     {the actual key translation table }
    record
      Version : 0..255;  {format of the key translation table}
      CmdState:  0..255;
      RawKeyBoard: boolean; { don't translate anything; command
                              comes out cChCmd for all keyboard;
                              for specials, command is 2,
                              ch is special-256 }
      EscState:  0..7;      { current Escape sequence state }
      NumMaps:   0..MAXMAP; { total maps in this keytran table }
      HashMask:  integer;
      Len:       integer;   { size of total table in bytes }

      Class: packed array [0..MaxCode] of KeyKind;
                                        { 2 bits per event }

      Map: packed array [1..1] of KeyMap; {vble length array}
    end;


  pKeyTab = ^KeyTransTab;

{     KeyEvent:
        a translated event generated by the user by typing
        on the keyboard or pressing a puck button, consisting
        of a command, char, region, viewport, and a pointer
        position.}
KeyEvent = record {not packed since goes into a message}
               vp     : viewport;
               X,Y    : integer;
               region : Integer;
               Cmd    : 0..255;
               Ch     : char;
             end;
```

4. Getting a version number

```
Function Sapph_Version(win:window):string;
```

Abstract:

Returns the version number and name of Sapphire as a string such as "V1.0".

Parameter:

win Any window (including SapphPort).

5. Setting the process control port

```
Procedure SetPMPort(ServPort: Window;
                    ProcCtlPort: Port);
```

Abstract:

Sets the process control port for Sapphire.

Parameters:

ServPort any window

ProcCtlPort port to send process control messages on

Errors:

Raises UserError if ProcCtlPort already set.

Note:

This procedure is used only by the system startup routines.

6. Window and Viewport Routines

6.1. Creating a window

```
Function CreateWindow(pw: Window;
                      fixedPosition: Boolean;
                      var leftx, topy: Integer;
                      fixedSize: boolean;
                      var width, height: Integer;
                      hasTitle, hasborder: boolean;
                      title: TitStr;
                      var progName: progStr;
                      hasIcon: boolean;
                      var vp: Viewport): Window;
```

Abstract:

Creates a new window on the screen. It will be in front of all other windows (of the same parent window). Note: If ASKUSER and aborted by the user, no window will be created and NullWindow is returned.

Parameters:

pw — The window that will be the parent of the new window. This can be FullWindow (or SapphPort, which is equivalent) to make the new window a subwindow of the entire screen.

fixedPosition — If true, the user is not allowed to move this window after it has been created.

leftX — The left X relative to the parent window of the new window. This will be the OUTER leftX of the window. If leftX is ASKUSER, then the user is requested for the window position. If ASKUSER, then set to the actual window leftX; not set if ASKUSER aborts with no window created.

topY — The top Y relative to the parent window of the new window. This will be the OUTER topY of the window. If topY is ASKUSER, then the user is requested for the window position. If ASKUSER, then set to the actual window topY; not set if ASKUSER aborts with no window created.

fixedSize — If true, the user is not allowed to change the size of this window

after it has been created.

width — The width of the OUTSIDE of the new window. The inner width may be less if there is a title or border. If width is ASKUSER, then the user is requested for the window width. If ASKUSER, then set to the actual window width; not set if ASKUSER aborts with no window created.

height — The height of the OUTSIDE of the new window. The inner height may be less if there is a title or border. If height is ASKUSER, then the user is requested for the window height. If ASKUSER, then set to the actual window height or not set if ASKUSER aborts with no window created.

hasTitle — True if the window should have a title area. This is a black area into which the title string may be written. If there is a title, the outer height will be the inner height + TITLEOVERHEAD.

hasBorder — True if the window should have a border area. This area is used to hold the hair line around the entire window and is the area where the window is shown to be the Listener or not. If there is no border, the Listener will not be shown in the window. If there is a border, it takes up BORDEROVERHEAD on each side (including the top).

title — The initial title for the window. It is always displayed in the system font. It is clipped if it won't fit in the title area.

progName — The initial string to show in the icon to name this window. If the name is not unique, the final characters are changed to a number to make it unique. The name actually used is returned.

hasIcon — If true, then the window has an icon. If false, then the window does not have an icon.

vp — Set to the viewport that corresponds to the inside of the window or NullViewport if ASKUSER aborts with no window created.

Returns:

The window created or NullWindow if ASKUSER aborts.

Notes:

1. FixedPosition and FixedSize are independent. For example, a window for an application implementing a terminal emulator may want to be exactly 80 characters across and 24 down, but not care where the window is placed. It

would set FixedSize to true and fixedPosition to false.

2. The coordinate system begins at 0,0 at the upper left corner of the inside of the parent window. The coordinate system is one-to-one and linear with the pixels on the screen. Negative numbers are legal; the window will simply be clipped at the top and/or left edge of the parent window. The legal coordinate range is (-16000,-16000) .. (16000,16000), and the maximum legal width and height is also 16000.

CAUTION - If you are creating and deleting a large number of windows, use the Kernel routine DeAllocatePort, not the window manager routine DeleteWindow, to remove the windows. DeleteWindow does not remove the ports for the deleted windows and the window manager may run out of ports.

6.2. Deleting a window

```
Procedure DeleteWindow(w: Window);
```

Abstract:

Deletes a window, all its subwindows, and all of the window's viewports. Do not use the window after calling this procedure.

Parameters:

w The window to delete.

CAUTION - If you are creating and deleting a large number of windows, use the Kernel routine DeAllocatePort, not the window manager routine DeleteWindow, to remove the windows. DeleteWindow does not remove the ports for the deleted windows and the window manager may run out of ports.

6.3. Modifying a window

```
Procedure ModifyWindow(w: Window; newleftx, newtopy,
    newouterwidth, newouterheight, newRank: Integer);
```

Abstract:

Changes the size, position and rank of a window. Any of these may be left unchanged by supplying UNCHANGED as the parameter. If the window is the IconWindow, then the icons are compacted and the window is redisplayed. If any of the parameters are ASKUSER, then the user is required to supply the missing information. If ASKUSER aborts, the window is unchanged. A special value for the rank is BOTTOM which means it is covered by all windows. (CreateWindow always creates the window with Rank = 1.)

Parameters:

w — The window whose parameters are to be modified.

newLeftx — The new outer left x of the window.

newtopy — The new outer top y of the window.

newouterwidth — The new outer width of the window.

newouterheight — The new outer height of the window.

newRank — The new rank of the window. Rank 1 means that the window is not covered, rank 2 means that the window is covered by one window, etc.

Errors:

If not allowed to change size or position and try to.

6.4. Moving a window offscreen

```
Procedure RemoveWindow(win: Window);
```

Abstract:

Moves the window to a special place off screen so that it is outside the refresh loop. The opposite of this procedure is RestoreWindow. This is nothing like DestroyWindow.

Parameters:

win The window to be removed from the screen.

6.5. Replacing a window onscreen

```
Procedure RestoreWindow(win: Window);
```

Abstract:

Gets back a window to its original place on the screen if it has been sent away using RemoveWindow. If hasn't been removed then no effect.

Parameters:

win The window to be restored to the screen.

6.6. Identifying a window

```
Procedure IdentifyWindow(w: Window);
```

Abstract:

Shows the relationship between a window and its icon briefly by video-inverting the pictures for each and drawing lines from one to the other.

Parameters:

w The window to display the relationship for.

6.7. Making a viewport

```
Function MakeViewport(pvp: Viewport; x,y,w,h, rank: Integer;
        memory, courteous, transparent: Boolean): Viewport;
```

Abstract:

Create a new viewport. MakeViewport is similar to CreateWindow in that it takes a parent. To create a viewport inside a window, use the viewport returned as the inside of the window. MakeViewport takes the initial coordinates inside its parent (ASKUSER is not allowed for viewports) and the initial rank.

Parameters:

pvp The parent viewport. The new viewport will be clipped inside of the parent viewport. To make this viewport global inside the screen, use FullViewport.

x,y The upper left corner of the new viewport with respect to pvp.

May also be OFFSCREEN in which case the viewport is offscreen (if either is OFFSCREEN then both are OFFSCREEN). x,y may be negative if the new viewport is to extend off the parent to the left or top.

w, h — The width and height of the new viewport. These must be >= 1. May extend outside the parent viewport.

rank — The rank of the new viewport with respect to other children of pvp. 1 means that the new viewport covers all others and BOTTOM (or any very large number) means that all other children cover the new viewport.

memory — Whether the viewport will have off-screen memory to back up any parts of the picture that are covered. If false, then the client is in charge of refreshing the contents of the viewport when it becomes uncovered. The client will be notified of this by a special emergency message (which can be converted into an exception in Pascal). Having memory is fairly expensive, however, since physical memory must be allocated for it. To create an offscreen picture buffer, simply use OFFSCREEN for x and y and make memory be true.

courteous — Whether the viewport saves the bit map underneath it. Currently this is used for popup menus and other pop-up viewports only under SAPPHIRE's control; do not set this parameter to true. (Currently when a courteous viewport is created, the screen area underneath is saved, but it is not updated if the picture changes. Therefore when the courteous viewport goes away, the picture put back may be incorrect.)

transparent — Whether this viewport covers viewports it is on top of. Most viewports will not be transparent but in some cases it may be convenient to be able to see graphics done to viewports behind another viewport. If a viewport is transparent then updates to viewports underneath it will show through. A possible application for this would be to cover a viewport or a set of viewports with a transparent viewport so that graphics could be done to the entire set. All of the graphics operations of a transparent viewport and the viewports underneath it are inseparably mixed together.

Returns:

The viewport created.

6.8. Destroying a viewport

```
Procedure DestroyViewport(vp: Viewport);
```

Abstract:

Deallocates a viewport and removes it from the screen. Any further use of the viewport will be an error. Also destroys all of the subviewports of this viewport.

Parameters:

vp The viewport to destroy.

CAUTION: DO NOT USE THIS TO DESTROY THE INNER VIEWPORT OF A WINDOW. USE DELETEWINDOW INSTEAD.

6.9. Getting a viewport's rank

```
Function GetVPRank(vp: Viewport): Integer;
```

Abstract:

Returns the rank of vp with respect to its parent. Higher ranks are covered by lower ranks. Rank = 1 is the top most (least covered). Offscreen viewports are not counted in the rank calculation.

Parameters:

vp The viewport.

Returns:

vp's rank If parent = NIL then returns 1. If vp not found under its parent, then returns last rank plus 1 (this should never happen).

6.10. Getting a viewport's state

```
Procedure ViewportState(vp: Viewport; var curlx, curty,
                curwidth, curheight, curRank: Integer;
                var memory, courteous, transparent: boolean);
```

Abstract:

Returns a description of vp.

Parameters:

vp The viewport information is desired for.

curLx, curTy
: Set to the upper left corner of this viewport in its parent's coordinate system.

curWidth, curHeight
: Set to the width and height of the viewport.

curRank
: Set to the rank of the viewport with respect to its brothers under their parent.

memory
: Set to true if the viewport has memory else false.

courteous
: Set to true if the viewport is courteous else false.

transparent
: Set to true if the viewport is transparent else false.

6.11. Modifying a viewport

```
Procedure ModifyVP(vp: Viewport; newlx, newty,
                   newwidth, newheight, newrank: Integer;
                   wantVpChEx: boolean);
```

Abstract:

Changes the position, size, and/or rank of a viewport.

Parameters:

vp
: The viewport to modify.

newLx, newTy
: The new upper left corner of this viewport with respect to its parent. UNCHANGED means that the corner does not change.

newWidth, newHeight
: The new width and height of the viewport or UNCHANGED if no change.

newRank
: The new rank of the viewport or UNCHANGED.

wantVPChEx
: If true and the change exception is enabled, then raises an exception after the viewport is modified. If false, then doesn't raise an exception even if enabled.

6.12. Getting a fullscreen viewport

```
Function GetFullViewport(vp: Viewport): Viewport;
```

Abstract:

Returns the viewport for the full screen.

*** NOT YET IMPLEMENTED ***

Parameters:

vp Any valid viewport.

Returns:

The viewport for the full screen. DO NOT MODIFY this viewport.

6.13. Reserving the screen

```
Procedure ReserveScreen(vp: Viewport; reserve: Boolean);
```

Abstract:

Pretends that vp is not covered. Operations are still clipped to be inside vp, but are not affected by other viewports.

WARNING: No permanent screen modifications should be done while the screen is reserved, only temporary ones (like window hair-lines). CURRENTLY, OTHER VIEWPORTS ARE NOT DISABLED so they can do graphics even if the screen is reserved.

Parameters:

vp The viewport to reserve the screen with respect to.

reserve If true then reserves the screen. If false, then releases the screen. It is OK to call with reserve false if the screen is not reserved. If vp dies, the screen is automatically un-reserved.

Errors:

If reserve is true but screen is already reserved for some viewport.

6.14. Getting screen parameters

```
Procedure GetScreenParameters(win: Window;
                var width, height: Integer);
```

Abstract:

Returns the width and height (in pixels) of the current screen.

Parameters:

win Any valid window.

width, height
Return the width and height of the screen.

6.15. Setting the title of a window

```
Procedure SetWindowTitle(win: Window; title: TitStr);
```

Abstract:

Sets the title of win to be new string. If window to be displayed has no title line, then this is a no-op. If the title is too long, then it is clipped.

Parameters:

win The window to set the title of.

title The new title.

6.16. Getting the fullscreen window

```
Function GetFullWindow(win: Window): Window;
```

Abstract:

Returns the full window. DO NOT MODIFY this window in any way.

Parameters:

win Any valid window.

Returns:

The window that is the full screen.

6.17. Changing the program name of a window

```
Procedure SetWindowName(win: Window; var progName: progStr);
```

Abstract:

Changes the ProgName for the window. Displays the new progname in the icon.

Parameters:

win — The window to change the progname for.

ProgName — The new ProgName for the window. If it conflicts (is the same as) any other ProgNames already existing then is changed to be unique by changing the last letters to be numbers.

6.18. Getting the complete state of a window

```
Procedure FullWindowState(w: Window;
                var leftx, topy, outerwidth,
                outerHeight, rank: Integer;
                var hasBorder, hasTitle,
                isListener: boolean;
                var name: ProgStr; var title: TitStr);
```

Abstract:

Returns the state of the window w itself.

Parameters:

w — The window whose description is desired.

leftx, topy — Set to the upper left corner of this window with respect to this window's parent.

outerWidth, outerHeight — Set to the width and height of the outside of the window.

hasBorder, hasTitle — Set to whether the window has a title or border.

isListener — Set to whether the window is currently the listener.

name — Set to the current name of the window.

title — Set to the current title string for the window.

6.19. Showing window progress

```
Procedure SetWindowProgress(win: Window; nestLevel: Integer;
                value, max: Long);
```

Abstract:

Shows progress in icon if appropriate. If max is 0 then random progress is done. If value >= Max then the progress bar is removed. Progress shown is (value +100 / max) percent.

Parameters:

win	The window to display the progress for.
nestLevel	Which utilProgress bar to show. It must be one or two.
value	The current value.
max	The maximum value.

Errors:

If nestLevel > NumProgressBars or < 1.

6.20. Creating an array of current window names

```
Procedure GetWinNames(win: Window; var names: pWinNameArray;
                var names_Cnt: long; var curListenIndex: Integer);
```

Abstract:

Creates an array of names of all the current windows.

Parameters:

win	Any valid window.
names	Storage is allocated and filled with all the names of the windows.
curListenIndex	Set to the index in the names array of the name corresponding with the current listener. If no listener, then will be set to -1.

6.21. Getting the window for a name

```
Function WinForName(win: Window; name: ProgStr): Window;
```

Abstract:

Returns the window for a name.

Parameters:

win — Any valid window.

name — The name of the window to get.

Returns:

The window referred to by name or NullWindow if name is not a legal window.

6.22. Getting a window's inner viewport

```
Procedure WindowViewport(w: Window; var vp: Viewport;
                         var vpWidth, vpHeight: Integer);
```

Abstract:

Returns the viewport for the insides of the window along with its width and height. This is the same viewport as is returned by CreateWindow as the var VP field.

Parameters:

w — The window whose viewport is desired.

vp — Set to the viewport for the insides of the window.

vpWidth, vpHeight — Set to the width and height of the viewport.

6.23. Defining a full size window

```
Procedure DefineFullSize(w, exceptW: Window);
```

Abstract:

Changes the meaning of "full size window" by adding a new window that should be ignored if a window is made full size. This procedure is incremental; every time it is called, the window is added to the list of ones to ignore. This list can be reset to no windows by passing in the NullWindow.

Each time a window is made full size, the meaning of full size is recalculated so if an "excepted" window is changed size, the meaning of full will automatically change.

Parameters:

w Any valid window.

exceptW The window that is to be added to the list of those to be excepted. If this is the NullWindow, then the FullWindow list is reset to be empty.

Errors:

Raises UserError if more than 11 windows are excepted.

6.24. Expanding a window to full screen

```
Procedure ExpandWindow(win: Window);
```

Abstract:

Expands the specified window to be full Screen. If this window is already full Screen and its size has not been modified, then this is a no-op. If the window was full screen and its size was modified, then this remembers the current size as the one to go back to. This is the opposite of ShrinkWindow. The meaning of full window is defined by DefineFullSize.

Parameters:

w A window that is to be expanded.

6.25. Returning a window to original size

```
Procedure ShrinkWindow(win: Window);
```

Abstract:

Shrinks the window back to its original size (opposite of expandWindow). If the window has not been expanded, then this is a no-op.

Parameters:

w A window that is to be shrunk.

6.26. Getting the window for a viewport

```
Function WinForViewPort(win: Window;vp: Viewport;
                        var isouter: boolean): Window;
```

Abstract:

Returns the Window for the specified viewport.

Parameters:

win	Any valid window.
vp	The viewport that the window is desired for. This can either be an inner or outer viewport.
outer	Set to true if vp is the outer viewport for win.

Returns:

The window for the specified viewport or NullWindow if none.

7. Icon Routines

7.1. Changing a window's error flag

```
Procedure SetWindowError(win: Window; error: boolean);
```

Abstract:

Changes the error flag for the window. Displays the picture for error in the icon if true, otherwise erases the picture.

Parameters:

win	The window to change the error flag for.
error	True to display picture. False to erase picture.

7.2. Changing a window's request flag

```
Procedure SetWindowRequest(win: Window; requesting: boolean);
```

Abstract:

Changes the requesting flag for the window. Displays the picture for requesting in the icon if true, otherwise erases the picture.

Parameters:

win	The window to change the requesting flag for.
requesting	True to display picture. False to erase picture.

7.3. Changing a window's attention flag

```
Procedure SetWindowAttention(win: Window; attn: boolean);
```

Abstract:

Changes the attention flag for the window. Displays the picture for attention in the icon if true, otherwise erases the picture.

Parameters:

win	The window to change the attention flag for.
attn	True to display picture. False to erase picture.

7.4. Compacting icons

```
Procedure CompactIcons(w: Window);
```

Abstract:

Compacts and redisplays the icons, removing any empty spaces.

Parameters:

w	A window that is ignored.

7.5. Allowing icon updates

```
Procedure IconAutoUpdate(w: Window; allowed: boolean);
```

Abstract:

Specifies that the icon for the window should or should not be automatically updated by the window manager. If not, then the icon will not have UtilProgress, WinGone, or the name displayed in it. This has the side effect that it sets Error, Requesting, and Attn to false and redisplays the icon which will therefore have an empty insides. The application is free to update the inside of the icon using the icon viewport. This procedure must be called with allowed = false before getting the viewport for the icon window.

Parameters:

w	The window whose icon should not be updated.
allowed	True to have the window manager automatically update the icon. False to do updates to icon window yourself and be allowed to get icon's viewport.

7.6. Getting an icon's viewport

```
Procedure GetIconViewport(w: Window; var iconvp: Viewport;
                          var width, height: Integer);
```

Abstract:

Returns a viewport for the insides of the icon for the window specified. This can be used by the application to show its own state in the icon. The application should be prepared to redisplay the icon if the viewport becomes uncovered or is moved. Call IconAutoUpdate(w, false) first.

Parameters:

w The window whose icon viewport is desired.

iconVP Set to the viewport or NullViewport if the window has no icon.

width, height
Set to the width and height of the icon vp.

Errors:

UserError('Update Allowed on window') if IconAutoUpdate(w, false) not called first.

Notes:

The only way to successfully destroy this viewport is to use the DeAllocIconVP routine described below.

7.7. Eliminating the icon window viewport

```
Procedure DeAllocIconVP(w: Window);
```

Abstract:

Eliminates the VP for the icon for this window. This erases the icon and redraws it. Does not allow IconAutoUpdate.

Parameters:

w The window whose icon viewport is to be destroyed.

7.8. Getting the icon window

```
Function GetIconWindow(win: Window): Window;
```

Abstract:

Returns the window that holds all the icons. Do not do graphics inside this window.

Parameters:

win Any valid window.

Returns:

The window that contains the icons.

8. Graphics Primitive Routines

8.1. Replacing a destination with a source

```
Procedure VPROP(destvp: Viewport; funct: RopFunct;
                dx, dy, width, height: Integer;
                srcVP: Viewport; sx, sy: Integer);
```

Abstract:

Does a rasterOp from src to destination using the covered windows. For setting a rectangle to white or black or inverting a rectangle, call VPColorRect instead of VPROP. Only the displayed portions on the screen are updated. If the dest VP has memory then the covered portions are updated in the offscreen memory. May raise the Exposed exception if portions in destination are not available in source. VPROP tries using the Kernel protected graphics functions first, and then, if that fails, calls SAPPHIRE's ViewRop.

Parameters:

destVP — The destination viewport. May be same as srcVP.

funct — The rasterOp function.

dx, dy — Coordinates of the upper left corner of the rectangle in the destination viewport.

width, height — The width and height of the rectangle to rasterOp.

srcVP — The source viewport. May be same as destVP.

sx, sy — Coordinates of the upper left corner of the rectangle in the source viewport.

Notes:

VPRop is the basic RasterOp primitive. It can be used to move the contents of one viewport around or to move the contents of one viewport to another. Like RasterOp in POS (the previous operating system for the PERQ), it takes the destination first, followed by the source. The function is one of the eight RasterOp functions: RRpl, RNot, ROr, RNor, RAnd, RNand, RXor, or

RXNor. The coordinate system of viewports is from (0,0) at the upper left corner of each viewport. Note that the order of the arguments is (x,y,w,h) which is different from POS's RasterOp.

8.2. Setting, clearing, or inverting a rectangle's bits

```
Procedure VPColorRect(vp: Viewport; funct: RectColorFunct;
                      x, y, width, height: Integer);
```

Abstract:

Operates on one rectangle to set, clear or invert all its bits. This is more efficient than ViewRop for these operations. Only the displayed portions on the screen are updated. If the viewport has memory then the covered portions are updated in the offscreen memory. Never generates exposed exception. Tries using the Kernel protected graphics functions first, and then, if that fails, calls SAPPHIRE's ViewColorRect.

Parameters:

vp — The viewport to modify.

funct — The operation to do: RectWhite, RectBlack, or RectInvert.

x, y — The upper left corner of the rectangle in vp's coordinate system.

width, height — The width and height of the rectangle to do.

8.3. Scrolling a viewport

```
Procedure VPScroll(destvp: Viewport; x, y, width, height,
          XAmt, YAmt: Integer);
```

Abstract:

Scrolls a portion of a viewport up, down, left, or right and erases the part that is left. Only the displayed portions on the screen are updated. If the viewport has memory then the covered portions are updated in the offscreen memory. Tries using the Kernel protected graphics functions first, and then, if that fails, calls SAPPHIRE's ViewScroll.

Parameters:

destvp — The viewport to modify.

x, y — Upper left corner of rectangle's old position with respect to

destVP.

width, height
: Width and height of the area to move.

xamt
: Number of bits to move the area horizontally. Negative numbers to move to the left, positive to move to the right.

yamt
: Number of bits to move the area vertically. Negative numbers to move up, positive numbers to move down.

8.4. Drawing a line in a viewport

```
Procedure VPLine(destvp: Viewport; funct: LineFunct;
                 x1,y1,x2,y2: Integer);
```

Abstract:

Draws a line in the viewport clipped to the displayed portions. Only the displayed portions on the screen are updated. If the viewport has memory then the covered portions are updated in the offscreen memory. Tries using the Kernel protected graphics functions first, and then, if that fails, calls SAPPHIRE's ViewLine. The line can be drawn as white, black or XOR.

Parameters:

destVP
: The viewport to draw the line in.

funct
: How to draw the line: DrawLine, EraseLine or XorLine.

x1, y1
: One end of the line. Coordinates are in destVp's coordinate space with 0,0 at the upper left.

x2, y2
: The other end of the line. Both end points are drawn.

Bugs:

The line may "bend" across viewport boundaries.

8.5. Displaying a string in a viewport with returns

```
Procedure VPString(destvp, fontVP: Viewport; funct: RopFunct;
                   var dx, dy: Integer;
                   var str: VPStr255; firstCh: Integer;
                   var lastch: Integer);
```

Abstract:

Displays a string in a viewport. As much of the string as will fit across is displayed (the procedure does not handle wrap around) and the amount that

was displayed is returned. Only the displayed portions on the screen are updated. If the viewport has memory then the covered portions are updated in the offscreen memory. Tries using the Kernel protected graphics functions first, and then, if that fails, calls SAPPHIRE's ViewString.

Parameters:

destVp	The viewport to put the string in.
fontVP	A viewport that is a font (returned from LoadFont). If NULLViewport, then SysFontVP used.
funct	The rasterOp function to use when displaying the string.
dx,dy	The starting location for the origin of the first character. Set to be the origin of the next character to be displayed after the characters actually written.
str	The string to display.
firstCh	The first character of the string to display. If DONTCARE, then 1 is used (first character of the string)
lastch	The last character of the string to display. If DONTCARE, then length(str) is used (the entire string is displayed). Set to the actual last character displayed. This may not be as many characters as was desired because the edge of the viewport was reached.

Errors:

If fontVP is not a font.

If fontHeight is too big for the special buffer.

Notes:

The procedure VPPutString does the same things as VPString, but it does not return any information and will not notify the application if fontVP is illegal or if there are other errors. VPChArray is the same as VPString except it takes a variable length array of characters instead of a string. VPChar can be used to display just one character. Again, there are corresponding procedures, VPPutChArray and VPPutChar, that do not return information.

Currently there is a restriction on the use of VPString and VPChArray. The height of a font that VPString and VPChArray will handle is set at 45 pixels;

therefore these procedures will return an error if they are used with a taller font. VPChar, however, will work with any size font.

8.6. Displaying a character array in a viewport with returns

```
Procedure VPChArray(destvp, fontVP: Viewport; funct: RopFunct;
                    var dx, dy: Integer;
                    chars: pVPCharArray;
                    arsize: long;
                    firstCh: Integer; var lastch: Integer);
```

Abstract:

Displays a character array in a viewport. Like VPString except that the characters come from a packed array of characters. Like VPPutChArray except it has return values.

Parameters:

destVp	The viewport to put the characters in.
fontVP	A viewport that is a font (returned from LoadFont). If NULLViewport, then SysFontVP used.
funct	The rasterOp function to use when displaying the characters.
dx,dy	The starting location for the origin of the first character. Set to be the origin of the next character to be displayed, after the characters are actually written.
chars	Pointer to a packed array of characters that contain the characters to be displayed. The array is deallocated.
arsize	Number of characters in the array.
firstCh	The first character to display. If DONTCARE, then zero is used (first character of the string).
lastch	The last character of the string to display. Cannot be DONTCARE. Set to the actual last character displayed. This may not be as many characters as was desired because the edge of the viewport was reached.

Errors:

If fontVP is not a font.

If fontHeight is too big for the special buffer.

Notes:

Currently there is a restriction on the use of VPString and VPChArray. The height of a font that VPString and VPChArray will handle is set at 45 pixels; therefore these procedures will return an error if they are used with a taller font. VPChar, however, will work with any size font.

8.7. Displaying one character in a viewport with returns

```
Procedure VPChar(destvp, fontVP: Viewport; funct: RopFunct;
                 var dx, dy: Integer; ch: Char);
```

Abstract:

Displays a single character in a viewport. Unlike the other text display routines, this one will not notify the user if the edge of the viewport has been reached. Only the displayed portions on the screen are updated. If the viewport has memory then the covered portions are updated in the offscreen memory. Tries using the Kernel protected graphics functions first, and then, if that fails, calls SAPPHIRE's ViewChar. VPPutChar is similar to this procedure but it does not have the return value.

Parameters:

destVp	The viewport to put the character in.
fontVP	A viewport that is a font (returned from LoadFont). If NULLViewport, then SysFontVP used.
funct	The rasterOp function to use when displaying the character.
dx,dy	The location for the origin of the character. Set to the origin of the next character to be displayed.
ch	The character to show.

Errors:

If fontVP is not a font.

8.8. Displaying a string in a viewport without returns

```
Procedure VPPutString
          (destvp, fontVP: Viewport; funct: RopFunct;
           dx, dy: Integer;
           var str: VPStr255; firstCh, lastch: Integer);
```

Abstract:

Same as VPString except no return values. Displays a string in a viewport. As much of the string as will fit across is displayed and the amount that was displayed is returned. Only the displayed portions on the screen are updated. If the viewport has memory then the covered portions are updated in the offscreen memory. Tries using the Kernel protected graphics functions first, and then, if that fails, calls SAPPHIRE's ViewString.

Parameters:

destVp	The viewport to put the string in.
fontVP	A viewport that is a font (returned from LoadFont). If NULLViewport, then SysFontVP used.
funct	The rasterOp function to use when displaying the string.
dx,dy	The starting location for the origin of the first character
str	The string to display.
firstCh	The first character of the string to display. If DONTCARE, then 1 is used (first character of the string)
lastch	The last character of the string to display. If DONTCARE, then length(str) is used (the entire string is displayed).

Errors:

NO NOTIFICATION IS GIVEN IF fontVP is not a font or fontHeight is too big for the special buffer; the operation simply doesn't happen.

8.9. Displaying a character array in a viewport without returns

```
Procedure VPPutChArray(destvp, fontVP: Viewport; funct: RopFunct;
                       dx, dy: Integer;
                       chars: pVPCharArray;
                       arsize: long;
                       firstCh, lastch: Integer);
```

Abstract:

Same as VPChArray except no return values. Displays a portion of a character array in a viewport. Like VPPutString except that the characters come from a packed array of characters. Only the displayed portions on the screen are updated. If the viewport has memory then the covered portions are updated in the offscreen memory. Tries using the Kernel protected graphics functions first, and then, if that fails, calls SAPPHIRE's ViewPutChArray.

Parameters:

destVp	The viewport to put the characters in.
fontVP	A viewport that is a font (returned from LoadFont). If NULLViewport, then SysFontVP used.
funct	The rasterOp function to use when displaying the characters.
dx,dy	The starting location for the origin of the first character.
chars	Pointer to a packed array of characters that contain the characters to be displayed. The array is deallocated.
arSize	Number of characters in the array. Needed for message passing.
firstCh	The first character to display. If DONTCARE, then zero is used.
lastch	The last character of the string to display. Cannot be DONTCARE.

Errors:

NO NOTIFICATION IS GIVEN IF fontVP is not a font or fontHeight is too big for the special buffer; the operation simply doesn't happen.

8.10. Displaying one character in a viewport without returns

```
Procedure VPPutChar(destvp, fontVP: Viewport; funct: RopFunct;
                    dx, dy: Integer; ch: Char);
```

Abstract:

Same as VPChar except no return values. Displays a single character in a viewport. Unlike the other text display routines, this one will not notify the user if the edge of the viewport has been reached. Only the displayed portions on the screen are updated. If the viewport has memory then the covered portions are updated in the offscreen memory. Tries using the Kernel protected graphics functions first, and then, if that fails, calls SAPPHIRE's ViewPutChar.

Parameters:

destVp	The viewport to put the character in.
fontVP	A viewport that is a font (returned from LoadFont). If NULLViewport, then SysFontVP used.
funct	The rasterOp function to use when displaying the character.
dx, dy	The location for the origin of the character
ch	The character to show.

Errors:

NO NOTIFICATION IS GIVEN IF fontVP is not a font; the operation simply doesn't happen.

8.11. Calculating screen coordinates of a point

```
Procedure VPtoScreenCoords(vp: Viewport; x, y: Integer;
                           var scrX, scrY: Integer);
```

Abstract:

Calculates the screen coordinates of the point in the viewport specified. Does not clip the point to the screen (the resulting scrX, scrY may be off screen). This is the inverse of ScreenToVPCoords.

Parameters:

vp	The viewport the point is with respect to.

x,y The coordinates of the point with respect to vp.

scrX, scrY Set to the corresponding point with respect to the screen.

8.12. Calculating viewport coordinates of a point

```
Procedure ScreenToVPCoords(vp: Viewport; scrX, scrY: Integer;
                           var x, y: Integer);
```

Abstract:

Calculates the viewport coordinates of the point in the screen. Does not clip the point to the viewport (the resulting x, y may be outside of the viewport). This is the inverse of VPtoScreenCoords.

Parameters:

vp The viewport the point is inside of.

scrX, scrY The point with respect to the screen.

x,y Set to the corresponding point in vp.

8.13. Loading font data

```
Function LoadFontData(vp     : Viewport;
                      pFont  : pVPIntegerArray;
                      arSize : long): Viewport;
```

Abstract:

Takes the raw font data and creates a viewport for it.

Parameters:

vp The parent viewport for the font. If the parent viewport is destroyed, the font viewport will also be destroyed.

pFont The raw font information passed as type pVPIntegerArray for Matchmaker but is actually of type FONTMAP, a pointer to a FontMapRec which is the font format on disk and in memory. In this routine, pFont is immediately recast as type FontMap and the information is used accordingly.

arSize The size of the font data map in words, needed by MatchMaker. The scan line word width for all fonts is the constant FontWordWidth which is currently 48. Take this into account when determining arSize.

Returns:

The font viewport for the data or NULLViewport if not successful.

Errors:

If cannot allocate memory for font;
If pfont is nil;
If font data is the wrong size.

8.14. Replacing one character in a font

```
Procedure SetFontChar(FontVP      : Viewport;
                      OneCharData : pVPIntegerArray;
                      arSize      : long;
                      WordsAcross : integer;
                      ch          : char;
                      CharWidth   : integer);
```

Abstract:

Replaces one character in a given font with the new data.

Parameters:

fontvp The existing font to modify.

onechardata The bit map.

arsize The size of OneCharData in words, needed by Matchmaker.

wordsacross The number of words in one scan line of the array.
NOTE: It must be a multiple of 4.

ch Becomes the index 0..#177 ASCII into the font array.

charwidth Number of bits wide.
NOTE: The new character must be the existing character's width or less. Once the character width has been made smaller, it cannot be increased.

WARNING: GetSysFont returns a pointer to the system font. If the system font is used as the font parameter in SetFontChar, the actual system font will be altered. We strongly advise against this use.

8.15. Finding the size of a font

```
Procedure FontSize(vp: Viewport; var name: string; var pointsize,
          rotation, facecode: Integer;
          var maxwidth, maxheight, xorigin, yorigin: integer:
          var fixedwidth, fixedheight: boolean);
```

Abstract:

Returns the parameters of the font.

Parameters:

vp — A viewport representing a font.

name — Set to the name of the font (e.g., "Helvetica" or "Computer Modern Roman")

pointsize — Set to the size of the font in points (has little to do with the size in pixels)

rotation — Set to the rotation of the font, in degrees counterclockwise from the positive X axis

facecode — Set to a number encoding the "face" of the font (bold, italic, normal, compressed, ...). The encoding is not specified here, but should be meaningful to a program that knows the Name of the font.

maxWidth, maxHeight
Set to the size of the bounding box for the font, the smallest rectangle containing all of the characters when their origins are aligned.

xorigin, yorigin
Set to the displacement of the origin from the upper left corner of the bounding box.

fixedWidth — Set to true if the font is fixed width, otherwise set to false.

fixedHeight
Set to true if the font is fixed height.

If FixedWidth is true for fonts at rotation 0 or 180, or FixedHeight is true for fonts at rotation 90 or 270, then FontStringWidth can be calculated without looking at the width vectors for individual characters.

Errors:

If vp is not a font.

Currently, all fonts have rotation 0, and the name, fontsize and facecode are returned as blank or 0. Fonts containing different values will be available in later releases.

8.16. Finding the width vector of a character

```
Procedure FontCharWidthVector(vp: Viewport;
                ch: Char; var dx, dy: integer);
```

Abstract:

Returns the width vector of the specified character.

Parameters:

vp	A font viewport.
ch	The character that the width is desired for.
dx, dy	Return the width vector for the character: the displacement from the character's origin to the origin for the next character to be drawn.

Errors:

If vp is not a font.

8.17. Getting the standard system font viewport

```
Function GetSysFont(vp: Viewport): Viewport;
```

Abstract:

Returns the standard system font viewport. Using NULLViewport in the drawing routines will use the System font, but if you need to inquire for font size or char size, use the return from this routine.

Parameters:

vp	Any valid viewport.

Returns:

The font viewport for the standard system font.

> WARNING: GetSysFont returns a pointer to the system font. If the system font is used as the font parameter in SetFontChar, the actual system font will be altered. We strongly advise against this

use.

8.18. Getting the width vector of a font string

```
Procedure FontStringWidthVector(vp: Viewport; var str: VPStr255;
                firstCh, lastch: Integer; var dx, dy: integer);
```

Abstract:

Returns the width vector of the specified portion of the string.

Parameters:

vp	A font viewport.
str	The string that a width is desired for.
firstCh	The character of the string to start at. If DONTCARE, then 1 is used (first character of the string).
lastch	The character of the string to stop at (from firstCh to lastCh INCLUSIVE is done). If DONTCARE, then length(str) is used.
dx, dy	Return the width vector for the string portion: the displacement from the first character's origin to the origin for the next character to be drawn after the string.

Errors:

If vp is not a font.

8.19. Setting or clearing one bit of a viewport

```
Procedure PutViewportBit(destvp: Viewport;
                x, y: Integer; value: boolean);
```

Abstract:

Set or clear a particular bit of the viewport.

Parameters:

destvp	The viewport to put the bit in
x, y	Location of the bit
value	TRUE to set the bit, FALSE to clear the bit

8.20. Reading one viewport bit

```
Function GetViewportBit(vp: Viewport; x, y: Integer;
                        var value: Boolean): boolean;
```

Abstract:

Read a particular bit of the viewport. Fails if the bit is not available because it is in a covered portion (of a viewport without memory) or if it is outside the viewport.

Parameters:

destvp	The viewport to get the bit from
x, y	Location of the bit
value	Set to TRUE if the bit is set; set to FALSE if the bit is clear

Returns:

True	if the bit is available
FALSE	if the bit is not available

8.21. Putting an array to a viewport

```
Procedure PutViewportRectangle(destvp: Viewport;
                funct: Ropfunct; x, y, width,
                height:  integer;
                Data: pVPIntegerArray; Data_Cnt: long;
                wordsacross: integer; ux, uy:  integer);
```

Abstract:

Performs a RasterOp operation from an array to a viewport. If the function is RRpl, this is the inverse of GetViewport.

Parameters:

vp	The viewport to modify.
funct	The RasterOp function to perform.
x, y	The upper left corner of the rectangle to write.
width, height	The size of the rectangle in pixels.
Data	A pointer to an integer array containing the contents of the rectangle.

Data_Cnt The size of the array in words.

WordsAcross
The number of words in one scan line of the array. Must be a multiple of 4.

ux, uy The upper left corner of the rectangle in the array.

8.22. Getting a rectangle from a viewport

```
Function GetViewportRectangle(vp: Viewport;
          x, y, width, height:  integer:
          data: pVPIntegerArray; var Data_Cnt: long;
                var WordsAcross: integer;
          ux, uy: integer): boolean;
```

Abstract:

Reads a rectangle from a viewport. Fails if any portion of the rectangle is covered (in a viewport without memory) or outside the viewport boundaries.

Parameters:

vp The viewport to read

x,y The upper left corner of the rectangle to read

width, height
The size of the rectangle in pixels

data Returns a pointer to an integer array containing the contents of the rectangle. The portion of the array not filled by the rectangle contents is zeroed.

Data_Cnt Returns the size of the array in words

WordsAcross
Returns the number of words in one scan line of the array. It will be a multiple of 4.

ux, uy The desired location for the upper left corner of the rectangle in the returned array

Returns:

True The rectangle is available

False It is not

8.23. Setting the screen color

```
Procedure SetScreenColor(vp: Viewport; invert: boolean);
```

Abstract:

Sets screen color to be black or white.

Parameters:

vp — Any new port.

invert

If TRUE, the background is BLACK - 0's on the screen will be black and 1's will be white.

If FALSE, the background is WHITE - 0's will be white and 1's will be black.

Note:

This call changes the color of the entire screen, including the background and all viewports and windows displayed.

9. Emergency Message Routines

9.1. Enabling exception notification

```
Procedure EnableNotifyExceptions(vp: Viewport; notifyPort: Port;
                                 changed, exposed: boolean);
```

Abstract:

Allows the client to be notified when the viewport is moved or if parts of it are exposed. If not enabled, then exposed parts are simply cleared to white. If the viewport has memory, then it will not usually get exposed exceptions. It will only get these when the viewport has gotten bigger or when a VPRop has taken place from another viewport that does not have memory, or when a VPRop source is outside a viewport.

Parameters:

vp	The viewport to enable or disable.
notifyPort	The port to send emergency messages to when the viewport is changed or exposed. If NULLPort, then no messages are sent.
changed	If true, then exceptions are enabled when the viewport's size or position is changed. If false, then disabled.
exposed	If true, then exceptions are enabled when any part of the viewport is exposed. If false, then disabled.

10. Cursor, Region, and Tracking Routines

10.1. Getting a bitmap and offset for one cursor

```
Procedure GetCursor(cursorImage  : CursorSet;
                    cursIndex    : Integer;
                var pCursorData  : pattern;
                var xOffset      : Integer;
                var yOffset      : Integer );
```

Abstract:

Get a single cursor's bitmap and offset from a given cursor set.

Parameters:

cursorImage
: The cursorSet containing the cursor.

cursIndex
: The index in cursorImage for the cursor desired. Cursor indices start at 0 and go to 126.

pCursorData
: Returns a pointer to the bit map.

xOffset, yOffset
: Returns which x,y bit in the cursor bit map is the actual pointer. [0..63]

Returns:

The bit map of cursor data and the offsets.

10.2. Getting information on a cursor set

```
Procedure GetCursorSet(cursorImage : CursorSet;
                   var Offsets     : OffsetPairArray;
                   var pCursorData : pPatternMapArray;
                   var numCursors  : Long );
```

Abstract:

Returns an entire cursor set's bit maps, the number of cursors, and each cursor's offset value.

Parameters:

cursorImage
The cursorSet containing the cursor.

offsets The x,y offset pair for each cursor.
NOTE: offset and cursor indices start at 0.

pCursorData
Pointer to a variable length array of 64 X 64 bit maps.

numCursors The number of (64 X 64) cursors.

10.3. Changing one cursor's bitmap and / or offset

```
Procedure SetCursor(cursorImage      : Cursorset;
                    cursIndex        : Integer;
                    pCursorData      : pattern;
                    xOffset          : Integer;
                    yOffset          : Integer );
```

Abstract:

Changes a single cursor's bitmap and / or offset within the given cursor set.

Parameters:

cursorImage
The cursorSet containing the cursor.

cursIndex The index in cursorImage for the cursor desired. Cursor indices start at 0 and go to 126.

pCursorData
The new bit map. If Nil, not changed.

xOffset, yOffset
Which x,y bit in the cursor bit map is the actual pointer. [0..63]
NOTE: Both x,y must be UNCHANGED or both must contain new valid offsets.

Errors:

Raises UserError for invalid cursor set, index, or offset.

10.4. Creating a Set of Cursors

```
Function CreateCursorSet(vp          : Viewport;
                         Offsets     : OffsetPairArray;
                         pCursorData : pPatternMapArray;
                         numCursors  : Long ): CursorSet;
```

Abstract:

Transforms raw 64 X 64 bit maps into a cursor set so they may be used to set the cursor.

Parameters:

vp Any viewport.

offsets The x,y offset pair for each cursor.
NOTE: offset and cursor indices start at 0.

pcursordata Pointer to a variable length array of 64 X 64 bit maps.

numcursors

The number of (64 X 64) cursors.

Returns:

A reference that can be used when setting the cursor or NULLPORT if something goes wrong.

Errors:

UserError If number of cursors is not 1..127.

10.5. Deallocating cursors

```
Procedure DestroyVPCursors(cursors: CursorSet);
```

Abstract:

Deallocates the cursors.

Parameters:

cursors The cursors to deallocate. Don't use them again.

10.6. Reserving the cursor

```
Procedure ReserveCursor(win: Window; reserve: boolean);
```

Abstract:

Prevents the down presses on the cursor from being interpreted by the window manager. All down presses go to win when win is the Listener. This is turned off by calling with reserve = false or when win is destroyed. This is independent of screen reserving function. If win is not the listener, then it is made the listener.

*** NOT YET IMPLEMENTED ***

Parameters:

win	The window that will reserve the cursor.
reserve	If true, then this cursor is reserved. If false, the the cursor is released. It is OK to call with false if cursor is not reserved.

Errors:

If some other process has the cursor reserved, or if win cannot be the listener.

10.7. Setting the cursor position

```
Procedure SetCursorPos(vp: Viewport; x,y: Integer);
```

Abstract:

Set the cursor position for this viewport. This overrules the settings of the cursor control for the region (gridding and trapping will NOT be applied to the coordinates). If the specified viewport is not the current listener, then this has no effect. If tracking is true, then this sets both the cursor and tablet position; otherwise, it sets only the cursor position.

Parameters:

vp	The viewport to set the cursor for.
x,y	The position for the cursor with respect to vp.

10.8. Specifying a region's cursor

```
Procedure SetRegionCursor(vp: Viewport; regionNum: Integer;
                cursorImage: CursorSet; cursIndex: Integer;
                cursFunc: CursorFunction; track: Boolean);
```

Abstract:

Specifies the cursor to be used in a region.

Parameters:

vp — The viewport for the region.

regionNum — The region in that viewport to modify.

cursorImage — The cursorSet containing the cursor. This cursorSet should have been returned by LoadVPCursors. NullPort will use the default cursor and cursIndex must be zero.

cursIndex — The index in cursorImage for the cursor desired. Cursor indices start at 0.

cursFunc — The cursor function to use. The choices are cfOR, cfXOR, or CFCursorOff. cfScreenOff means that the cursor is visible but the entire rest of the screen (not just this viewport) is invisible. cfBroken is a non-functional cursor function. (Note that the overall screen color is specified with an entirely separate function.)

track — Whether the cursor should follow the mouse or not while this region is active.

Errors:

Raises userError if region not there or cursIndex is out of bounds for cursorImage.

Notes:

In general, track will be true for this procedure, but if the application wants to de-couple the tablet and the cursor, false can be used. The procedure SetCursorPos will set the cursor position independent of the tablet position if track is false; otherwise it sets both.

10.9. Getting the state of a region's cursor

```
Procedure GetRegionCursor(vp: Viewport; regionNum: Integer;
                var cursorImage: CursorSet;
                var cursIndex: Integer;
                var cursFunc: CursorFunction;
                var track: Boolean);
```

Abstract:

Returns the current state of the region's cursor.

Parameters:

vp The viewport for the region.

regionNum The region to get information from.

cursorImage
Set to the cursorSet for the region.

cursIndex Set to the index in cursorImage for the cursor for the region.

cursFunc Set to the cursor function in use.

track Set to true if the cursor will follow the mouse and false if not.

Errors:

Raises userError if region not there.

10.10. Setting cursor control values for a region

```
Procedure SetRegionParms(vp: Viewport; regionNum: Integer;
                    absolute: boolean; speed: Integer;
                    minx, maxx, miny, maxy, modx, posx,
                    mody, posy: Integer);
```

Abstract:

Specify the cursor movement control parameters for a region. The special value DONTCARE can be used for most to get the defaults.

Parameters:

vp The viewport for the region.

regionNum The region to set up.

absolute Whether the tablet coordinates are directly mapped to screen coordinates or whether incremental movements on the tablet will be added (creating relative movements). If absolute is true then

speed is ignored.

speed — If relative (not applicable to absolute), then controls how movements on the tablet will map to movements on the screen. 1 means 1:1. Positive numbers mean that one increment on the tablet will be translated into that number of increments on the screen. Negative numbers mean that number of increments on the tablet will be translated into one increment on the screen. DONTCARE ==> 1.

minX, maxX, minY, maxY — These form a rectangle in the region that the cursor is not allowed to leave when it goes into that region. Thus the cursor will be trapped by this rectangle. If all are DONTCARE, then the cursor is not trapped.

modx — The grid factor in the x direction. The cursor will only be put on every "modx"th point. Must be >= 1 or DONTCARE. DONTCARE ==> 1.

posx — If modx <> 1 then this determines the offset to put the cursor on in the x direction. It should be less than modx. DONTCARE ==> 0.

mody — Same as modx, only for the y direction.

posy — Same as posx, only for the y direction.

10.11. Getting the cursor control values for a region

```
Procedure GetRegionParms(vp: Viewport; regionNum: Integer;
                var absolute: boolean; var speed: Integer;
                var minx, maxx, miny, maxy, modx, posx,
                mody, posy: Integer);
```

Abstract:

Returns the cursor movement control parameters for a region.

Parameters:

vp — The viewport for the region.

regionNum — The region to read.

absolute — Set to true if the tablet coordinates are directly mapped to screen coordinates or false if incremental movements on the tablet will be added (creating relative movements). If absolute is true then speed is ignored.

speed — If relative (not absolute), then set to the value that controls how movements on the tablet will map to movements on the screen. 1 means 1:1. Positive numbers mean that one increment on the tablet will be translated into that number of increments on the screen. Negative numbers mean that number of increments on the tablet will be translated into one increment on the screen.

minX, maxX, minY, maxY — Set to the values that form a rectangle in the region that the cursor is not allowed to leave when it goes into that region. Thus the cursor will be trapped by this rectangle. If all are DONTCARE, then not trapped.

modx — Set to the grid factor in the x direction. The cursor will only be put on every "modx"th point.

posx — Set to the offset to put the cursor on in the x direction if modx <> 1.

mody — Set to the grid factor in the y direction.

posy — Set to the offset to put the cursor on in the y direction.

10.12. Pushing a region

```
Procedure PushRegion(vp: Viewport; regionNum: Integer;
                     leftx, topy, width, height: Integer);
```

Abstract:

Pushes a new copy of regionNum onto VP. If regionNum is already a region of VP, then it is NOT deleted so that this region can be popped to go back to the old state. It is a bad idea to have the same region pushed twice with different sizes since the older one may not be fully covered by the newer one and therefore still be visible. If the regionNum is VPREGION or OUTREGION then the coordinates are ignored. If either, the rectangle for the region is set to the viewport's rectangle. The default parameters for the region are taken from the old region with the same number if present. If not present, then taken from the first region for this viewport. If that's not there, then taken from the parent of this viewport's first region if there, otherwise uses system defaults. This default can be overridden by subsequently calling SetRegionParms and SetRegionCursor.

Parameters:

vp The viewport to add a region for.

regionNum The region to add. If there already, then the new version hides the old one.

leftx, topy, width, height
The rectangle for this region with respect to the viewport. If regionNum is VPREGION or OUTREGION then ignored, otherwise, may not be UNCHANGED.

Errors:

Raises userError if try UNCHANGED and not VP or OUT REGION.

10.13. Modifying a region

```
Procedure ModifyRegion(vp: Viewport; regionNum: Integer;
                       leftx, topy, width, height: Integer);
```

Abstract:

Modifies an existing region to have a new shape and position.

Parameters:

vp The viewport the region is for.

regionNum The number for the region.

leftx, topy, width, height
The new parameters for the region. May be UNCHANGED. If the regionNum is OUTREGION or VPREGION then these parameters are ignored (the coordinates for those regions are the coordinates of the viewport).

Errors:

Raises UserError if region not there.

10.14. Deleting a region

```
Procedure DeleteRegion(vp: Viewport; regionNum: Integer);
```

Abstract:

Deletes the most recent copy of regionNum. If only one, then regionNum becomes illegal. If more than one had been pushed, then the older one becomes available.

Parameters:

vp The viewport containing the region.

regionNum

The region to delete.

Errors:

Raises userError if try to delete a region that isn't there.

10.15. Deallocating all regions

```
Procedure DestroyRegions (vp: viewport);
```

Abstract:

Deallocates all the regions for vp except for the first VPREGION and OUTREGION.

11. Listener Routines

11.1. Allowing a window to be listener

```
Procedure EnableWinListener(win: Window; emergPort: Port);
```

Abstract:

Allows the window to be the Listener and allows the window to take input.

Parameters:

win — The window that can be the Listener.

emergPort — This port is for emergency messages to the window. Currently, only viewports receive emergency messages, but it may be used in future versions of the window manager. (Most applications set it to NullPort.)

11.2. Changing a window to be listener

```
Procedure MakeWinListener(win: Window);
```

Abstract:

Changes the Listener to the win window. The window must have been enabled using EnableWinListener.

Parameters:

win — The window to be the new Listener.

**Caution-Read Section 2.6, Listener, before using this procedure.

11.3. Finding the window for the listener

```
Function GetListenerWindow(win: Window): Window;
```

Abstract:

Returns the Window for the listener.

Parameters:

win Any valid window.

Returns:

The window for the listener or NULLWindow if there is no Listener.

12. Input Routines

12.1. Using the GetEvent Routine

In the window manager, events are queued on a per-window basis. To request a keyboard or mouse event, use the procedure GetEvent explained below. However, if you want to wait on events coming in on several different ports or windows or use a loop to update the screen and get events, you may wish to import ModGetEvent (located in the user library and described in the document "Pascal Library" in the *Accent Languages Manual* and call the GetEventPort and ExtractEvent routines instead of GetEvent. In the latter case, using KeyDontWait and calling GetEvent will work, but more slowly.

GetEvent takes a window and HowWait as parameters and returns an untranslated EventRec. HowWait determines how to wait for the event. If it is KeyDontWait, GetEvent returns immediately with an untranslated event. This may be an actual event if one has happened or it may be a position or cursor response. Position response come in two forms: POSITION and DIFFPOSITION. POSITION is used whenever the point being returned has the same coordinates as the last point returned. DIFFPOSITION is used when the coordinates are different. These can be mapped to the same abstract command code in the key translation table if desired. If the window specified is not the Listener, then it can never get real events. KeyDontWait always returns immediately, however, so the special command NOEVENT is returned.

If HowWait is KeyWaitDiffPos, GetEvent returns the DIFFPOSITION event the next time the cursor is in a different position. (Obviously, with KeyWaitDiffPos, the POSITION

special event will never be generated since no response will happen if the position is not different.) KeyWaitDiffPos should make some applications more efficient if they only need to know the position when different. KeyWaitDiffPos does not return an event if the window is not the Listener. If HowWait is KeyWaitEvent, an actual keyboard or mouse event is waited for.

If HowWait is KeyWaitDiffPos or KeyDontWait and an actual keyboard or mouse event occurs, that event will be returned instead of a POSITION event. The values for HowWait and the returns are listed below.

HowWait	Possible Returns
KeyWaitEvent	Actual event. Waits to return if not the listener.
KeyDontWait	Actual event. POSITION DIFFPOSITION NOEVENT (if not listener)
KeyWaitDiffPos	Actual event. DIFFPOSITION NOEVENT (if not listener)

12.2. Getting the next keyboard or mouse event

```
Function GetEvent(win: Window; howWait: KeyHowWait): EventRec;
```

Abstract:

Returns the next untranslated keyboard or mouse event. GetEvent followed by a TranslateKey call produces a translated event. Alternatively, the user library routine GetTranslatedEvent can be used to return the translated event directly.

If howWait is KeyDontWait then returns immediately with a Position or DiffPosition event (or a regular event if one has been queued). If howWait is KeyWaitDiffPos then waits for the x,y position of the cursor to be different. If

howWait is KeyWaitEvent then waits for the next key or button transition. If win is not the Listener and howWait is KeyDontWait, returns the NoEvent event. Otherwise when win is not the Listener, will wait for win to be the Listener before returning.

Parameters:

win	The window that an event is wanted for.
howWait	Determines how to wait for the event.

Returns:

EventRec	an untranslated event.

Errors:

If input has not been enabled for this window.

12.3. Flushing queued events for a window

```
Function FlushEvents(win: window): boolean;
```

Abstract:

Flushes all queued events for a window.

Parameters:

win	The window to flush.

Returns:

True	If any events were outstanding (and are now lost).
False	Otherwise.

13. Routines for Loading Data Files

MODULE SAPPHLOADFILE (in file SapphLoadFile.pas in the Pascal library):

This module contains routines for obtaining information to pass on to the Window Manager, such as cursor data files, font data files and Picture files.

13.1. Reading in a font

```
Function LoadFontFile(vp: Viewport; fileName: VPStr255): Viewport;
```

Abstract:

Read in a font from the disk and creates a viewport for it.

Parameters:

vp — The parent viewport for the font. If the parent viewport is destroyed, the font viewport will also be destroyed.

fileName — The string name of the font file (including any extensions).

Returns:

The font viewport for the file or NULLViewport if not found.

Errors:

If cannot allocate memory for font.

If not a legal font.

13.2. Reading in a picture

```
Function LoadVPPicture(vp: Viewport; fileName: VPStr255;
                       width, height: Integer): Viewport;
```

Abstract:

Read in a picture from the disk and create a viewport for it. This is only useful when the size of the picture is known.

Parameters:

vp — The parent viewport of the picture viewport. If the parent viewport is destroyed, the picture viewport will also be destroyed.

fileName — The string name of the picture file (including any extensions). The picture is read starting at byte 0 of the file.

width, height — The desired dimensions of the picture. This must be consistent with the actual picture or it will not be read in correctly.

Returns:

The viewport for the picture or NullViewport if not found.

Notes:

LoadVPPicture allows a picture to be read from a file into an off-screen viewport with memory. We do not have a general format for pictures in files, so this routine simply assumes the picture is in the file with no header information (like the picture's width and height). LoadVPPicture therefore takes the width and height of the picture as parameters. If these do not correspond exactly with the parameters of the picture, then the picture will not be read in correctly. One application of LoadVPPicture is to load special pictures containing gray patterns which can be created by a program or by using the CursDesign program. PutViewportRectangle may also be used to load a picture from memory into an existing viewport.

13.3. Loading a cursor set into memory

```
Function LoadVPCursors(vp: Viewport; fileName: VPStr255;
                       var numCursors: Integer): CursorSet;
```

Abstract:

Reads a cursor file from disk and creates a cursorset from it.

Parameters:

vp Any valid viewport.

fileName The string name of the cursor file, including any extensions.

numCursors Returns the number of cursors found in the file.

Returns:

A reference that can be used when setting the cursor or NullPort if file not found.

Errors:

If file does not seem to be a valid cursor file.

14. Key Translation

14.1. Introduction

The keytranslation compiler (KeyTranCom) is a utility that enables application programs to translate keyboard events, puck presses, and cursor movements and therefore to define commands that can be executed by the application. Using this utility, you may:

1. Develop your own keyboard set.
2. Detect cursor positions and region exits.
3. Dynamically add, delete or change keymap entries in the .KEYTRAN keytranslation table file.
4. Save the newly changed .KEYTRAN file.
5. Change a version 3 .KEYTRAN file to a version 4 .KEYTRAN file without the .KTEXT file (using the function ConvertToNewVersion). Note, however, that the Definitions in the KDEFS and KTEXT files may not be changed dynamically.

Definitions of the constants used in connection with the key translation facility can be found in the file KeyTranDefs.Pas in the user library. This information is also contained in Section 3.3 of this document.

14.2. Application Program Requirements

Whenever a keyboard key is hit, a mouse button goes down or up, the tracker leaves a region, or the cursor takes a position, an "event" is said to have happened. The keytran facility allows these events to be converted into abstract command codes so that an application program can be written independent of the actual letters that cause the desired actions. This should make it easier for users to convert applications such as editors to use the command set they like best. Key translations are controlled by a "key translation table." These tables are created from source text using a special key translation table compiler (called KeyTranCom). The format for the source text to KeyTranCom is given in Section 14.5. The key translation mechanism supports prefix keys (such as CTRL-X in the editor) and both up and down transitions of the mouse. One key translation table may define different commands for the same keyboard, mouse or cursor event when the cursor is in different regions. This might be useful to an editor, for example, where a press with a particular button might mean "SCROLL-UP" in the scroll region, but mean "SELECT-WORD" in the text region.

The user library routine GetTranslatedEvent returns a translated event directly to the application program. However, when using the GetEvent routine discussed in Chapter 12, the TranslateKey routine must follow in order to return a translated event of type KeyEvent.

Key translation is not done until an event is requested by the application. Calling TranslateKey or GetTranslatedEvent with the appropriate key translation table allows the application to control which table is used.

The key translation table mechanism allows all the keys on the keyboard to be differentiated (it will take raw keyboard data).

One key is reserved for use as a prefix to the window manager. This key is defined in the system's key translation table (WindowMgr.KText) and is CTRL-DEL (on PERQ's) or SETUP (on PERQ2's). The commands that can be given after the prefix key are defined in the "User's Guide to the Window Manager" in the *Accent User's Manual.*

NOTE: Mouse buttons modify ANY keystroke, button press, position response, or other event. Each of the 16 (for the 4-button mouse) or 8 (for the 3-button mouse) possible combinations of mouse buttons with any key or event is a different event.

Keytranslation tables may be altered dynamically using the following procedures: AddToKeyTable, ChangeKeyTable, DeleteFromKeyTable, and MakeAnEmptyTable. You can load a table, change it, and save the altered table. There is also a function, ConvertToNewVersion, to change Version 3 (or earlier) tables to Version 4 tables. All of these routines are available from the module Keytran in the user library.

For more information on the routines discussed here, see the sections which follow.

In sum, in order to use key translation the application program must:

1. Import the XKDEFS.pas file, where x is the name of the file produced by the keytranslation compiler.
2. Call LoadKeyTable, which returns a pointer to the keytranslation table loaded into memory. Optionally, the program may use MakeAnEmptyTable to create an empty table which will return raw events.

3. Alter parts of the keytranslation table in memory by using AddToKeyTable, ChangeKeyTable, and/or DeleteFromKeyTable. If you want a change made with any these routines to become a permanent part of the table, use SaveKeyTable. (Optional)

4. Request events from the keyboard or mouse by calling either:

    a. GetEvent - which returns an untranslated event, followed by a call to TranslateKey using any loaded key table. Or,

    b. GetTranslatedEvent - which returns a translated event directly.

5. Handle the command. (Translated event contains x, y coordinates, region, viewport, command and character.)

14.3. Module Keytran

MODULE KEYTRAN (in file KeyTran.pas in the Pascal library):

This is the module to be imported for manipulations of the key translation table.

```
Const
        HashmaskNumber = 31;

        {Keyboard Keys value returned}
        KeyHELP = 7;
        KeyBACKSPACE = 8;
        KeyTAB = 9;
        KeyLF = 10;
        KeySETUP = 11;
        KeyNOSCROLL = 12;
        KeyRETURN = 13;

        KeyOOPS = 21;
        KeyINS = 27;
        KeyESC = 27;
```

```
KeySPACE = 32;

KeyDEL = 127;
KeyUPARROW = 128;
KeyBREAK = 128;
KeyDOWNARROW = 129;
KeySHIFTBREAK = 129;
KeyLEFTARROW = 130;
KeyRIGHTARROW = 131;

KeyPF1 = 132;
KeyPF2 = 133;
KeyPF3 = 134;
KeyPF4 = 139;
KeyENTER = 140;
KeyNCOMMA = 146;
KeyNMINUS = 147;
KeyNPERIOD = 148;
KeyN0 = 150;
KeyN1 = 151;
KeyN2 = 152;
KeyN3 = 153;
KeyN4 = 154;
KeyN5 = 156;
KeyN6 = 157;
KeyN7 = 158;
KeyN8 = 159;
KeyN9 = 160;

KeyControlSPACE = 128 + 32; { Perq 1 only }
KeyControlDEL = 128+127;
KeyControlHELP = 128+7;
KeyControlBACKSPACE = 128+8;
KeyControlTAB = 128+9;
KeyControlLF = 128+10;

KeyControlOOPS = 128+21;
KeyControlRETURN = 128+13;

KeyControlESC = 128+27;
KeyControlINS = 128+27;

KeyControlSETUP = 6;
KeyControlNOSCROLL = 14;
KeyControlUPARROW = 28;
KeyControlDOWNARROW = 29;
KeyControlLEFTARROW = 30;
KeyControlRIGHTARROW = 31;

KeyControlBREAK = 130;
KeyControlSHIFTBREAK = 131;

iYellowMask = #01;
iMiddleMask = #01;

iWhiteMask  = #02;
```

```
        iLeftMask   = #02;

        iBlueMask   = #04;

        iGreenMask  = #10;
        iRightMask  = #10;

imports SapphDefs       from SapphDefs;
imports PathName        from PathName;
imports KeyTranDefs     from KeyTranDefs;

type RawIn = packed record case integer of
     1:(region  : integer;
        code    : 0..#177; { key character coding }
        cntrlOn : boolean; { true for keybd chars, if cntrl
                             key was down }
        special : boolean; { true for transitions not generated
                             by keybd, i.e., buttons or
                             regionExit, etc. }
        escCl   : 0..#7;   { escape class - added before hashing,
                             comes in as zero }
        mButtons: 0..#17); { state of bit pad 1 buttons after
                             event }
     2:(reg     : integer;
        hashks  : integer);
        end;

     TranOut = record
        cmd     : 0..255;
        ch      : char;
        escTy   : escKind;
        end;
```

14.4. Key Translation Routines

14.4.1. Adding to a key translation table

```
Function AddToKeyTable(Key1          : RawIn;
                       Key2          : TranOut;
                       tab           : pKeyTab;
                   var conflictDepth : Integer): GeneralReturn;
```

Abstract:

Adds the new entry to the existing keytranslation TABLE.

Parameters:

Key1 — The UNTRANSLATED event information containing the region and the keystate

Key2 — The TRANSLATED event information containing the command,

the escape kind, and the character

tab The pointer to the keytranslation table

conflictDepth
If added an entry and there was a hash conflict, then the depth of the hash conflict is returned, otherwise 0.

Returns:

Success or GR value if failed. Returns the conflict depth if there was a conflict.

14.4.2. Changing a key translation table

```
Function ChangeKeyTable
           (OldKey : RawIn;
            Key1   : RawIn;
            Key2   : TranOut;
            tab    : pKeyTab): GeneralReturn;
```

Abstract:

Changes the old element into the new one. WARNING: THE TABLE MAY NOT BE IN THE SAME ORDER!

Parameters:

OldKey The old raw information

Key1 The new raw information

Key2 The new translated information

tab Pointer to the keytranslation table to alter

Returns:

Success or GR value if failed.

14.4.3. Deleting from a key translation table

```
Function DeleteFromKeyTable(Key : RawIn;
                            tab    : pKeyTab): GeneralReturn;
```

Abstract:

Deletes the element from the table.

Parameters:

Key The old raw information

tab Pointer to the keytranslation table to alter

Returns:

Success or GR value if failed

14.4.4. Converting an old keytran table to version 4

```
Function ConvertToNewVersion(OldKeyName : path_name;
                    var pKeyNew     : pKeyTab): GeneralReturn;
```

Abstract:

Changes the old version of a keytran file (versions 1-3) which hash using the size of the table, into the current version, which uses a constant since the table may be changed dynamically.

Parameters:

OldKeyName
Absolute pathname of old keytran file

pKeyNew Pointer to new version of table, or Nil if failed

Returns:

Success or GR value if failed. Also returns Success if the file is already the correct version and does not need to be converted.

14.4.5. Saving a key translation table

```
Function SaveKeyTable(Tab         : pKeyTab;
                      OutFileName: Path_Name): generalReturn;
```

Abstract:

Writes the key map entries to a KEYTRAN file.

Parameters:

Tab Pointer to Keytranslation Table

OutFileName
Absolute name of the file to which to write the table

Returns:

Success or GR value if failed.

14.4.6. Making an empty key translation table

```
Function MakeAnEmptyKeyTable(Raw  : boolean;
                             var pKey : pkeytab): GeneralReturn;
```

Abstract:

Creates an empty keytranslation table and puts it in memory. Use SaveKeyTable to write it to a file. Essentially it is an empty map table with Table^.rawkeyboard set to TRUE for RawKeyboard values and FALSE for ascii values to be returned. Escape class is set to standard.

Parameters:

Raw — If TRUE, sets table to be rawkeyboard; else ascii values are returned and nothing for mouse buttons, which is the default.

pKey — Pointer to table, Nil if failed

Returns:

Pointer to table or Nil if failed. Success or GR value if failed.

14.4.7. Deallocating a keytranslation table

```
Function DestroyKeyTable(Tab: pKeyTab): GeneralReturn;
```

Abstract:

Deallocates the specified key translation table. Do not use it afterwards.

Parameters:

tab — Pointer to table to deallocate

Returns:

Success or GR value if failed.

14.4.8. Loading a key translation table

```
Function LoadKeyTable(name: Path_Name;
                      var pKey: pKeyTab): generalreturn;
```

Abstract:

Reads in a key translation table from the disk. This file should have been created by the Key Translation Compiler (KeyTranCom) or created using MakeAnEmptyKeyTable and SaveKeyTable or ConvertToNewVersion and

SaveKeyTable.

Parameters:

name	Absolute name of the file to read the table from. Extension ".KeyTran" may be omitted.
pKey	Returns a pointer to the table of NIL if not loaded.

Returns:

success	Table was loaded.
otherwise	Error code describing why table was not loaded.

Errors:

If key translation table not found, if malformed, if it is the wrong version, or if the file does not seem to be a key translation table.

14.4.9. Translating a raw event

```
Function TranslateKey(     kt : pKeyTab;
                            e : EventRec;
                      var k : KeyEvent): GeneralReturn;
```

Abstract:

This is the main procedure to translate a raw key event into a translated event to be passed to the user.

Parameters:

kt	Pointer to key translation table to use in translation
e	Raw event from the tracker
k	Set to the translated event based on kt

Returns:

Success or GR value if failed

14.4.10. Getting a translated event

```
Function GetTranslatedEvent(pKey: pKeyTab;
                            win: window;
                            howwait: KeyHowWait): KeyEvent;
```

Abstract:

Returns the next keyboard or puck event as a TRANSLATED EVENT. If howWait is KeyDontWait then returns immediately with a Position or DiffPosition event (or a regular event if one has been queued). If howWait is KeyWaitDiffPos then waits for the x,y position of the cursor to be different. If howWait is KeyWaitEvent then waits for the next key or button transition. If window is not the Listener, then will wait for window to be the Listener before returning, unless howWait is KeyDontWait in which case returns the NoEvent event. (See the summary of possible returned events in Section 12.1.)

Parameters:

pKey	Pointer to translation table to use
win	Window that an event is wanted for
howWait	Determines how to wait for the event

Returns:

TRANSLATED EVENT record for the event

Errors:

If input has not been enabled for this window

14.5. Translation Tables

This section describes the format for the creation of a key translation table and provides an example .KTEXT file. Compiling this file using KeyTranCom produces a file foo.KEYTRAN which is used by the window manager and application programs and a file fooKDEFS.PAS which is imported by the program using the table.

To execute, type *keytrancom*. You will be prompted for the .KTEXT filename, the .KEYTRAN filename, the KDEFS.PAS

filename and the .ERR error filename. If you wish all the files to be called FOO, you may type KEYTRANCOM FOO and the files will be called FOO.KTEXT, FOO.KEYTRAN, FOOKDEFS.PAS and FOO.ERR. You will not be prompted for filenames.

To explain further, if your source file is called Test.KTEXT, typing *KeyTranCom Test* produces three files:

1. TEST.KEYTRAN - This is a binary file to be used by the window manager. The function LoadKeyTable uses this as a parameter and returns a pointer of type pKeyTab to be used with routines which take a keytranslation table as a parameter.
2. TESTKDEFS.PAS - This readable file contains the keytranslation definitions and must be imported by the application program. If the application is requesting only raw events, this file will not be generated.
3. TEST.ERR - This file contains any warnings or errors found while executing keytrancom.

The following conventions have been used in describing the format of a keytranslation table text file:

"<>" enclose non-terminals,
"{}" enclose comments,
"[]" enclose optional items,
"*" between items means that they can come in any order,
"|" between items means a choice.

Literals are shown in upper case but will not be case-sensitive.

Begin comments in the file with "!" and the rest of the line will be ignored. Blank lines are also ignored.

All special separator characters (literals in the text, such as "+" and "=") must have spaces around them.

Numbers can be in octal by preceeding them with a #. Numbers must be positive and less than 256.

When specifying a character, shift (case) is significant. Thus "CONTROL A" is "control shift a", whereas "CONTROL a" is just control a. Because this distinction is made, a shift prefix is therefore not needed (shift of special keys is not significant).

If you simply wish rawkeyboard events then the file should have the one word in it:

```
RAWKEYBOARD ! optionally followed by the word
END         ! Comments are permitted.
```

A Key translation text file will have the following format:

```
DEFINITIONS


region <name> = <2..30>
       <name> = <2..30>


command <name> = <2..255>
        <name> = <2..255>
```

The definitions section is optional if no commands or regions need to be named.

Region and command names must follow standard Pascal naming conventions. Names are 1..25 characters in length and must be unique. Case is preserved but irrelevant.

The region name is the name given to the window manager via the PushRegion routine. Predefined region names are VPREGION, OUTREGION, WILDREGION:

- VPREGION is the full area inside the viewport.

- OUTREGION is the full area outside the viewport.
- WILDREGION matches any region.

The command name is a Pascal constant which is returned in KeyEvent record when the event defined by that command is issued. Predefined command names are STDCOMMAND and NULLCOMMAND.

- STDCOMMAND is used when no translation is done.
- NULLCOMMAND is only used with prefixes, see below.

Region and Command definitions may repeat in any order.

```
KEYTRANSLATIONS

[Region <region name>]

                          ! standard form
<key desc> = <command name> [<char code>] [PREFIX]

                          ! prefix form
<key desc> + <key desc> =
       <command name> [<char code>] [STAYINMODE]

                          ! continue with additional regions

[END]                     ! if END present, then the
                          ! rest of file is ignored
```

If no region is specified, WILDREGION is used. The order for regions is important. If the same key is defined in a region and in WILDREGION, the one in the specific region should be defined first so it will take precedence over the definition in WILDREGION.

NOTE: A single quote is needed when <char code> is a character.

DEFINING PREFIXES

Before using a prefix in other <key descriptions>, the prefix first must be defined, by using the Standard form and the word PREFIX.

If the prefix is to return a value when typed, provide the value as

the <command name>. If no value is to be returned, specify NULLCOMMAND. NULLCOMMAND can only be used with prefixes.

The <char code> for prefixes must be 1..7 and defines the Escape Class of the prefix. Different prefixes should have different <char codes> if they are to have different effects.

The ANY <key desc> is used with prefixes to specify that any character after the prefix will have the same command number. For a prefix to be used before an ANY, define the prefix as:

```
prefix = NULLCOMMAND <char code 1..7> PREFIX

prefix + ANY = <command name>
```

The prefix must be defined with NULLCOMMAND. Any character typed after the prefix will have the command name specified on the "any" line. There should be no <char code> on this line as shown. The character typed after the prefix will be returned as the character code of the event.

DEFINITIONS OF NON-TERMINALS

```
The asterisks mean that the color prefixes can be in any order.

<key desc> ==
  [BLUE]*[YELLOW]*[WHITE]*[GREEN] <keyboard key> |
                                          !for 4-button mouse

  [MIDDLE]*[LEFT]*[RIGHT] <keyboard key>   !for 3-button mouse

<keyboard key> == [CONTROL] <key> | [CONTROL] <0...127> |
                  <0...255>      ! the ASCII values of a
                                 ! character

<key> == <'character> | ! any printing character (not SPACE)
         <special key name> |
         <special actions> |
         ANY

<char code> == [CONTROL] <'character>
                     ! No special actions allowed.

        | <0...255>
```

```
        | UPPERKEY    ! Upper case of key, e.g. C for c.

        | ROOTKEY     ! The key without the control bit; shift is
                      ! significant, e.g. C for ^SHIFT-C or c
                      ! for ^c. Special keys come in as
                      ! control codes.

        | FULLKEY     ! The exact number of that key as it
                      ! comes in as raw data from keyboard.

        | ASCIIKEY    ! The ascii value that corresponds to
                      ! the key. This is the default if no
                      ! character is given.


<special key name> ==
        INS | DEL | HELP | TAB | BACKSPACE |
        OOPS | RETURN | LF | SPACE | ESC |
           ! INS and ESC are the same.

           ! PERQ1 only (same as PERQ2 ENTER)
        CONTROLSPACE |

           ! PERQ2 keys
        NOSCROLL | SETUP |
        UPARROW | DOWNARROW | LEFTARROW |
        RIGHTARROW |
           ! Breaks are same as arrows
        BREAK | CONTROLBREAK | SHIFTBREAK |
        CNTRLSHFTBREAK |
           ! Control doesn't work with the next 3 lines of keys
           ! from the numeric keypad
        N0 | N1 | N2 | N3 | N4 | N5 | N6 | N7 | N8 | N9 |
        PF1 | PF2 | PF3 | PF4 |
        ENTER | NCOMMA | NMINUS | NPERIOD


<special actions> ==
           ! blue no equivalent on 3-button mouse
        BLUEDOWN | BLUEUP |

           ! white=left button
        WHITEDOWN | WHITEUP | LEFTDOWN | LEFTUP |

           ! green=right button
        GREENDOWN | GREENUP | RIGHTDOWN | RIGHTUP |

           ! yellow=middle button
        YELLOWDOWN | YELLOWUP | MIDDLEDOWN |
        MIDDLEUP |

        REGIONEXIT | POSITION | DIFFPOSITION |
        NOEVENT | LISTENER

        ! REGIONEXIT   - Event raised when the cursor moves out
                         of a region.
```

```
! POSITION     - Event raised when the cursor position
                 remains the same.
! DIFFPOSITION - Event raised when the cursor position
                 changes.
! NOEVENT      - Event raised when no events take place.
                 (This only happens when a non-listener
                 requests an event with KeyDontWait.)
```

Example .KTEXT file

```
DEFINITIONS

region regl = 3
       reg2 = 4
       reg3 = 5

command PosResponse = 65
        DiffPos = 66
        NoneEvent = 67

        YellowDown  = 68
        WhiteDown   = 69
        GreenDown   = 70
        BlueDown    = 71

        YellowUp    = 72
        WhiteUp     = 73
        GreenUp     = 74
        BlueUp      = 75

        YellDownReg2 = 76
        YellUpReg2   = 77
        YellDownReg3 = 78
        YellUpReg3   = 79

        NewRegionEvent = 80
        IMListener   = 81

KEYTRANSLATIONS

region reg2

  yellowdown = yellDownReg2 '[
  yellowup = yellUpReg2 ']

region reg3

  yellowdown = YellDownReg3 '*
  yellowup = YellUpReg3 '#

REGION WildRegion

  Position = PosResponse 'p
  yellow position = PosResponse 'p
  blue position = PosResponse 'p
  white position = PosResponse 'p
  green position = PosResponse 'p

  DiffPosition = DiffPos 'd
  yellow DiffPosition = DiffPos 'd
  blue DiffPosition = DiffPos 'd
  white DiffPosition = DiffPos 'd
  green DiffPosition = DiffPos 'd
```

```
    NoEvent = NoneEvent 'n

    RegionExit = NewRegionEvent '%

    Listener = IMListener '&

    yellowdown = yellowDown 'y
    yellowup = yellowUp 'Y

    whitedown = whiteDown 'w
    whiteup = whiteUp 'W

    greendown = greenDown 'g
    greenup = greenUp 'G

    ! blue is UNDEFINED for testing

END
```

14.6. Converting Existing Programs

In order to convert an existing program using key translation to function properly with the new key translation routines (that is, to convert a version 3 or earlier program to a version 4), the following steps must be followed.

1. Run the KeyTranCom program on your .Ktext file to produce a new .keytran and kdefs.pas file.
2. Import module KeyTran from file Keytran.Pas. (This will import KeyTranDefs from the file KeyTranDefs.Pas, which contains the definitions.)
3. Make the Sapphire Listener changes to your program (i.e., remove EnableInput and SetListener and add EnableWinListener).
4. Load the key translation table with the function LoadKeyTable.
5. Use function GetTranslatedEvent instead of function GetEvent. (Use of GetEvent necessitates a subsequent call to TranslateKey.)

If you do not have the original .KTEXT file and cannot perform the steps listed above, use the ConvertToNewVersion routine, followed by SaveKeyTable. These routines are explained in Section 14.4.

15. Errors and Exceptions

If a function or procedure is to return an item (for example, a window or viewport), a NULL value will typically be returned if the item is not there. All other errors are signalled by a single exception UserError which has a string parameter explaining the error. Another exception, IMPOSSIBLE, is raised when the window manager discovers a bug in its internal workings.

15.1. "Impossible" Error Messages

An error message that begins with the word "Impossible" indicates an operation that the window manager cannot handle. If such an error occurs, the user should report to PERQ Systems the error received and the circumstances. After such an error, the user must reboot.

WinManager

```
SetRingCurrent: Sets the current listener
        'Window not found in Ring'

InitWinManager:
        'Cursors not found': if LoadVPCursors fails
        'Wrong number of cursors':

DeleteWindow: Deletes a window and all its subwindows
        'window to be deallocated not found'

GetWinNames: Creates an array of names of all the current
         windows
        'Unable to allocate memory': if ValidateMemory fails
        'NumNames not right':

DestroyWinNameArray: Deallocates the storage used by a
         WinNameArray
        'Unable to Deallocate memory': if InValidateMemory fails
```

WinUserKey

```
HandleListenerMenu: Shows menu of listeners
        'Bad number of answers': more than one
         menu choice picked

DoGetWinManMenu: Popup menu of window manager
        'Bad number of answers': more than one
         menu choice picked

ShowIconCursor: set up regions and cursor for press in an icon
        'Bad icon button': a press other than the three buttons
         registered

AssertTitCursorShapes: set up regions and cursor for press
         in an icon
        'Bad tit button': a press other than the three buttons
         was registered

DoHandlePen: Takes a press down or up event
        'Bad command': something other than the three buttons
         was registered
```

WinAskUser

```
DoAskUserMove: Asks the user for a window position
        'Cant move from left top'
        'Cant move from right top'
        'Cant move from right bottom'
        'Cant move from left bottom'
        'NoWhere': no point chosen

DoAskUserGrow: Asks the user for a window size.
        'NoWhere': no point chosen
```

WinIconManager

```
InitIcons:
        'Icon pic not found': if LoadVPPicture fails

FindFreeIconNumber: Finds and returns an icon number.
        'No Icons free'

ReCalcIconWindow: Recalculates the positions of all the icons
        in the icon window.
        'Icon back link error'
```

VPMain

```
ReadPhysFile:Read a file from the disk
        'VPMain,ReadPhys:Unable to invalidate':
         if InvalidateMemory fails

DeAllocateMem:
        'Unable to destroy': if DestroySegment fails

RecalcCoveredRankChange: Recalculates coveredness
        'VpMain:RecalcCov:Didnt find vp or oldPos in list'

InitVP: Initialize all of the viewport system
        'VpMain:InitVp:Cant enable privs': if EnablePrivs fails
        'Sys font not found':

DestroyVPRectArray: Deallocate a rect array
        'VpMain:DestroyVPRectAr:Unable to Deallocate memory': if
        InValidateMemory fails

ModifyVP: Changes the position, size, and/or rank of a viewport
        '** Recursive call on ModifyVP'
```

VPRegion

```
DecCursorRefCount: Decrement the use count on a Cursor Set.
        'Unable to free cursor memory': if InvalidateMemory fails

DeQueue: Removes an event from the specified queue.
        'Q is NIL'

AddDefRegions: Adds the regions OUTREGION and VPREGION
         to the viewport
```

```
        'first region not nil'

getVPRegionForCoords: Searches the viewport for a region
        'Didnt find a rec rec'

QueueKey: Adds a key to the queue
        'Queue is NIL'

LoadFontData:
        'LoadFontData: Can not move data to physical memory'
```

VPIntersect

```
DestroyRectList:
        '** Bad chain'

DestroyRegRectList:
        '**Bad Reg Rect Chain'

InterRegion: Divides the region
        '***InterRegion doesnt work on OUTREGION'
```

VPGraphics

```
DoROP: Does the rasterOp
        'VPgraphics:AddToList:Unable to allocate memory'
        'VPgraphics:DOROP:** Bad rop ctrl'
        'VPgraphics:DOROP:** S clip failed'

ViewROP:
        'VPgraphics:ViewRop:** Bad ropDir'

MoveVPViewROP:
        'VPgraphics:MoveVPViewRop:** Bad ropDir'

ViewColorRect:
        'VPgraphics:ViewColorRect:** Illegal RectColorFunct'
        '** one color clip failed'

ViewChArray:
        'VPgraphics:Unable to free memory in ViewChArray'
```

```
PutViewportRectangle:
        'VPgraphics:Unable to free memory in GetViewport'

GetViewport:
        'VPgraphics:GetViewport - can''t dispose user mem'

SetCursor:
        'SetCursor: Unable to deallocate memory'
```

SapphConverter

```
AllocateBucket:
        'Port already in use'

DestroyBucket:
        'Wrong type on destroy'
        'Bucket to destroy not found'
```

Sapphire

```
DispatchAndSend:
        'Sapphire:Dispatch:Recursive Dispatch and send'

SendViewPtChanged:
        'Sapphire:SendViewPtChanged:notify is NULL for changed'

SendViewPtExposed:
        'Sapphire:SendViewPtExposed:notify is NULL for exposed'

SendSavedMsg:
        'Sapphire:SendSavedMsg: msg is NIL':

WaitKeyInEvent:
        'Sapphire:WaitKeyInEvent:** Impossible message recd'
```

SpopUpCurs

```
InitCurs: allocates and reads cursors.
        'Bad cursor file'
```

15.2. "User Error" Messages

Error messages that begin with "User Error" are the result of errors that originate in the program. These messages are listed below.

WinManager

```
CheckCoords: checks x,y,w,h of window
        'Coordinate or Size out of range': if x,y,w,h not in
             maxcoord,mincoord, maxsize
        'Size less than min for this window': if w,h smaller than
             minW,minH

AllocateWin: allocates a new window
         'No more windows': if window table contains
          maxwindows already

ShowListener: draws border when doing DEL +,-
         'Window cannot be Listener':

CreateWindow:
        'Unchanged is bad parameter': if leftx, topy, width
         or height is UNCHANGED

ModifyWindow:
        'Cant change width'
        'Cant change height'
        'Cant change TY'

DefineFullSize:
        'No free slots to save exceptwin': if more than 11
         windows are excepted

GetIconViewport:
        'Update Allowed on window': if iconautoupdate not
         called first

SetWindowProgress: shows progress in icon and title area
        'Illegal Nesting level':

SetPMPort: Sets process control port for Sapphire
        'Control port already set.'
```

VPMain

VPAllocMem: Allocates a physical segment
'Create mem error': if CreateSegment fails

ReserveScreen:
'Some vp is already reserved'

ModifyVP: Changes the position, size, and/or rank of a viewport
'Can''t change courteous vp': if vp^.courteous and try to change

VPRegion

DeleteRegion: Removes region from the Viewport.
'Region not found in delete'

ModifyRegion: Modifies an existing region to have a new shape and position.
'Region not there': if try to modify region that doesn't exist

PushRegion: Pushes a new copy of regionNum onto VP
'UNCHANG bad Reg param': can't use UNCHANGED as a parameter

SetRegionCursor: Specifies the cursor to be used in a region.
'SetRegionCursor:Region not found':
'cursIndex out of bounds'

GetRegionCursor: Returns the current state of the region's cursor.
'GetRegionCursor:Region not found':

SetRegionParms:
'SetRegionParms:Region not found':

GetRegionParms:
'GetRegionParms:Region not found':

GetEvent:
'Window has no KeyTab':

IntGetEvent: used by the window manager to wait for characters

```
        'IntGetEvent:VP is Null':
        'IntGetEvent:Window has no KeyTab'

CreateCursorSet:
        'CreateCursorSet: Number of cursors per cursor set
           is 1..127'

SetCursor:
        'Set Cursor: Index must be 0..126'
        'SetCursor: X and Y Offsets must be 0..63'
        'SetCursor: Both offsets must be UNCHANGED or values'
```

VPGraphics

```
GenLine: Generate a line of text into a special buffer
        'VPgraphics:GenLine: Font too tall'

ViewString:  Displays a string in a viewport.
        'VPgraphics:fontVP not a font in ViewString'

ViewChArray:
        'fontVP not a font in ViewChArray'
        'Last ch not in array'

ViewChar:
        'fontVP not a font in ViewChar'

FontSize: Returns the parameters of the font
        'Not a font in FontSize'

FontCharWidthVector: Returns the width vector of the
         specified character
        'Not a font in FontCharWidth'

FontStringWidthVector: Returns the width vector of the
         specified portion of the string
        'Not a font in FontStringWidth'

PutViewportRectangle:
        'GetViewport - too wide for SysROTemp':
        'PutViewport - WordsAcross mod 4 <> 0':
        'PutViewport - not enough data'

GetViewportRectangle:
        'GetViewport - too wide for SysROTemp'
        'GetViewport - can''t allocate memory'
```

SapphConverter

```
GetPointer: Looks up a port to convert to a pointer
        'Port not found': If Pointer for port is NIL
        'Wrong type on GetPtr':

SetUpVPConv: Set up the conversion from a viewport to a
         port and visa versa
        'No more ports for vp': if AllocatePort fails

SetUpWinConv: Set up the conversion from a window to a port
         and visa versa.
        'No more ports for win': if AllocatePort fails

ReleaseWinPort: Deallocates window's conversion
        'Cant delete port for win': if DeallocatePort fails

ReleaseViewportPort: Deallocates viewport's conversion.
        'Cant delete port for vp': if DeallocatePort fails

SetUpCursorConv: Set up the conversion from a cursorset to a
         port and visa versa
        'No more ports for cursorset': if AllocatePort fails

ReleaseCursorPort: Deallocates CursorSet's conversion
        'Cant delete port for win': if DeallocatePort fails
```

16. Sample Application Program

In the program that follows, line breaks have sometimes been inserted in places where the programer did not originally put them. This is necessary due to the constraints of the printed page, but we have tried to insert these breaks so that they do not hinder your understanding of this program.

```
Program Ttest;
{-------------------------------------------------------------
   Test program to illustrate parts of Sapphire interface
   CopyRight (c) 1983, 1984 - PERQ Systems Corporation

  Change Log:
 13-Dec-84  V1.0  Amy Butler    Example for manual.
    Uses data files:
      jj.cursor : contains one cursor bit pattern;
      talk.kst  : font of faces for animation;
      testerKdefs.pas, tester.keytran : created from
          Tester.Ktext by Keytrancom
      tester2.scursor : contains several cursors
-------------------------------------------------------------}

Imports Sapph              from SapphUser;
                              {Window Manager interface}
Imports ViewPt             from ViewPtUser;
                              {Window Manager interface}
Imports Sapphdefs          from SapphDefs;
                              {Window Manager interface}

Imports SapphFileDefs      from SapphFileDefs;
                              {for cursor, font definitions}
Imports SapphLoadFile      from SapphLoadFile;
                              {for loading cursors, fonts, pix}
Imports Keytran            from Keytran;
                              {for keytran table manipulations}

Imports SaphEmrServer      from SaphEmrServer;
                              {figure out which emerg msg}
Imports SaphEmrExceptions from SaphEmrExceptions;
                              {the exceptions}

Imports PascalInit         from PascalInit;
Imports PathName           from Pathname;
Imports AccInt             from AccentUser;
                              {using SoftEnable}
Imports Except             from Except ;
```

```
                              {to get emerg msg exception}
Imports AccCall             from AccCall;
                              {for receive}
Imports SaltError           from SaltError;

Imports TesterKDefs         from TesterKDefs;
                              {keytranslation table,}
                              {generated from Tester.Ktext}

var
    fullWindow, Win1, win2, win          : WINDOW;

    GR                                   : GeneralReturn;
    waithow                              : KEYHOWWAIT;
    KeyTable, NewKeyTable                : pKEYTAB;

    title                                : TITSTR;
    progName                             : PROGSTR;
    strToPrint                           : VPSTR255;

    offvp, vpkid1, vpkid2, iconVp,
    scrvp,
    vpfont, SysFontVP, scrvp2,
    talkfont, fontviewport,vpPix         : VIEWPORT;

    keyEv,rawKeyEv                       : KEYEVENT;
    EvRec                                : EVENTREC;

    myCursors, my2Cursors                : CURSORSET;
    pCursorData                          : PATTERN;
    numCursors                           : integer;

    fname,f2name                         : Path_Name;
    pDataRec                             : pVPINTEGERARRAY;
    arSize                               : Long;
    GotIt                                : Boolean;

    num                                  : Long;
    dum, fixedW, fixedH, color           : Boolean;
    pushed, IconH, IconW                 : Integer;
    c, whichchar, ch                     : Char;
    count, dummie, i, j, k, pointsize,
    x,xl,y,yl,w,wl,h,hl,
    r, rl, minW, minH,
    xll, yll, wll, hll, rll,char_width,
    maxwidth, maxheight, xorigin,
    yorigin, xpos, ypos, dx, dy, last    : Integer;
    fontname                             : string;


{-------------------------------------------------------------
{                      all for emerg msgs
{-------------------------------------------------------------}

const MaxMsgSize = 2048;          { Max message size we can }
```

```
                                { receive in bytes }
type Space = array [0 .. MaxMsgSize div 2 - 1] of integer;

     pDummyMsg  = ^DummyMsg;

     DummyMsg   = record      { A record large enough to hold }
        case boolean of           { the fixed portion of all }
          true: (head    : Msg;        { of our messages }
                 RetType : TypeType;
                 RetCode : Integer;
                 body    : Space);
          false: (nextFreeMsg: pDummyMsg);
        end;

var pInMsg      : array[1..10] of pDummyMsg;
                    { Pointer to a message we will receive }
    pRepMsg     : pDummyMsg;
                    { Pointer to a reply message we will send }
    msgInUse: array[1..10] of boolean;

Procedure HandleAllEmergMsgs;
 var i: Integer;
 begin
 for i := 1 to 10 do
  if msgInUse[i] then
    begin
    if not SaphEmrServer(pInMsg[i], pRepMsg) then
      writeLn('** Got unknown emerg msg: ', pInMsg[i]^.head.id:1);
    msgInUse[i] := false;
    end;
 end;

Handler EmergMsg;
  var gr: GeneralReturn;
      num, i: Integer;
  begin
  num := -1;
  for i := 1 to 10 do
    if not msgInUse[i] then num := i;
  if num = -1 then WriteLn('** all 10 msgs in use')
  else begin
       pInMsg[num]^.head.MsgSize := MaxMsgSize;
       GR := Receive(pInMsg[num]^.head, 0, AllPts, ReceiveIt);
                                                  { Get work }
       if gr <> success then
            GRWriteStdError(GR, GR_Error, 'Lost the emerg msg')
       else (** HandleEmgMsg ** Can't do this here due to
                    MatchMaker bug **)
            msgInUse[num] := true;
       end;
  dum := true;
  HandleAllEmergMsgs;  (** Due to matchMaker bug **)
  GR :=SoftEnable(false, dum);
  if GR <> success then
     GRWriteStdError(GR, GR_Error, '');
```

```
  end;

{-------------------------------------------------------------------}

Handler EViewPtChanged(vpl: VIEWPORT; xl,yl,wl,hl,r: Integer);
  var i: integer;
  begin
  if vpl = iconVP then
       begin
       VPCOLORRECT(vpl, RectBlack, xl, yl, wl, hl);
       VPCOLORRECT(iconVP, rectwhite, 0, 0, iconW div 2,
                   iconH div 2);
       VPCOLORRECT(iconVP, rectwhite, iconW div 2, iconH div 2,
                   iconW, iconH);
       end
  else begin
       if r=1 then  { if top window }
            VPROP(vpl, RRPL, xl, yl, wl, hl, offvp, xl, yl);

       w := wl;
       h := hl;
       if vpl = scrvp then
         begin
         MODIFYREGION(vpl, regl, 0, 0, w div 2, h);
         MODIFYREGION(vpl, reg2, w div 2, 0, w div 2, h);
         if pushed>0 then
           MODIFYREGION(vpl, reg3, 0, h div 3, w div 4, h div 3);
         end
       else if vpl = scrvp2 then
           MODIFYREGION(vpl, regl, 0, 0, w, h);
       end;
  end;

Handler EViewPtExposed(vpl: VIEWPORT; ra: pRECTARRAY;
                       numRectangles: Long);
  var i,j: Integer;
  begin
  { refresh all uncovered area with saved offscreen info }
  if vpl = scrvp then
    for i := 1 to Shrink(numRectangles) do
      with ra^[i] do
        VPROP(vpl, RRPL, lx, ty, w, h, offvp, lx, ty);
  if vpl = iconVP then
    with ra^[1] do
     begin
     VPCOLORRECT(vpl, RectBlack, lx, ty, w, h);
     VPColorRect(iconVP, rectwhite, 0, 0, iconW div 2,
                 iconH div 2);
     VPColorRect(iconVP, rectwhite, iconW div 2, iconH div 2,
                 iconW, iconH);
     end;
  { deallocate memory }
  GR := InValidateMemory(KernelPort, recast(ra, VirtualAddress),
        wordsize(rectarray)*2);
  if GR <> Success then
    GRWriteStdError(GR, GR_Error, 'Failed to deallocate memory');
```

```
  end;

{-------------------------------------------------------------------}

Handler Impossible(s: String);
  begin
  WriteLn(chr(7), 'IMPOSSIBLE ***** ',s);
  exit(Ttest);
  end;


Handler UserError(s: String);
  begin
  WriteLn(chr(7), 'USER ERROR ***** ',s);
  exit(Ttest);
  end;

{-----------------------------}
 Procedure Pictures;
{-----------------------------}
var x,y: integer;
begin
vpPix := LOADVPPICTURE(scrvp, 'jj.cursor',56, 64);
write('Enter x:'); Readln(x);
write('Enter y:'); Readln(y);
VPROP(scrvp, RRPL, x, y, 56, 64, vpPix, 0, 0);
end;

{-----------------------------}
Procedure InitializeFonts;
{-----------------------------}
begin
SysFontVp := GETSYSFONT(scrvp);

{Get Font files}
vpfont := sysfontvp;

f2name := 'talk.kst';
GR := FindFileName(f2name,'',false);
if GR <> success then
        GRWriteStdError(GR, GR_FatalError, f2name);

talkfont := LOADFONTFILE(scrvp, f2name);
fontname := 'talkfont';
FONTSIZE(talkfont, fontname, pointsize, dummie, dummie, maxwidth,
         maxheight, xorigin, yorigin, fixedW, fixedH);
end;

{-----------------------------------}
Procedure PlayWithFonts(ch:char);
{-----------------------------------}
var arsize              : long;
    i                   : integer;
    bits                : pVPINTEGERARRAY;
    fontname            : string;
    pointsize, dummie,
```

```
    maxwidth, maxheight,
    xorigin, yorigin     : integer;
    fixedW, fixedH       : boolean;
    dx, dy               : integer;
    wordsacross          : long;
begin

{change the given char to black and white in vpfont}
FONTSIZE(vpfont, fontname, pointsize, dummie, dummie, maxwidth,
        maxheight, xorigin, yorigin, fixedW, fixedH);
FONTCHARWIDTHVECTOR(vpfont, ch, dx, dy);
bits := nil;
wordsacross := stretch((dx + 15) div 16);
if wordsacross < 4 then wordsacross := 4;
arsize := wordsacross * maxheight;
gr := validatememory(KernelPort, RECAST(bits,virtualaddress),
        arsize, -1);
if gr<>success then
   GRWriteStdError(GR, GR_FatalError, 'Can''t validate')
else begin
   {$R-}
   for i := 0 to arsize-1 do
    if (i mod 2)=1 then
         bits^[i] := #125252
    else bits^[i] := #052525;
   {$R=}
   SETFONTCHAR(vpfont,bits,arsize,shrink(wordsacross),ch,dx);
   gr := InvalidateMemory(KernelPort, RECAST(bits, virtualaddress),
                          arsize);
   if gr<>success then
      GRWriteStdError(GR, GR_FatalError, 'Can''t invalidate');
   end;
end;
{--------------------}
Procedure Load;
{--------------------}
var pfont: FONTMAP;
begin
new(pfont);

with pFont^ do
     begin
     height := 8;
     base := 8;
     index[0].offset := 0;
     index[0].line := 0;
     index[0].width := 16;
     filler[0] := 0;
     filler[1] := 0;
     {$R-}
     for i := 0 to 7 do
        if ((i*4) mod 8) = 0 then
             Pat[i*4] := #177400
        else Pat[i*4] := #000377;
     {$R=}
     arsize := stretch(wordsize(height) +
```

```
                        wordsize(base) +
                        wordsize(index) +
                        wordsize(filler) +
                        (height * fontwordwidth));
     end;

fontviewport := LOADFONTDATA(scrvp, RECAST(pfont, pVPINTEGERARRAY),
                             arsize);
VPPUTCHAR(scrvp, fontviewport, RRPL, 100, 100, chr(0));
end;

{------------------------------}
Procedure PlayWithCursors;
{------------------------------}
var i,j, xpos, ypos      : integer;
    pCursorData          : PATTERN;
    pCursorDataArray     : pPATTERNMAPARRAY;
    xOffset, yOffset     : integer;
    Offsets              : OFFSETPAIRARRAY;
    numcurs              : long;
begin
{get individual cursors and print in the viewport}
new(pCursorData);
xpos := 10; ypos := 10;
for i := 0 to numcursors-1 do
     begin
     GETCURSOR(myCursors, i, pCursorData, xoffset, yoffset);
     PUTVIEWPORTRECTANGLE(scrvp2, RRPL, xpos, ypos, 64, 64,
                          RECAST(pCursorData, pVPINTEGERARRAY),
                          Stretch(wordsize(PATTERNMAP)), 4, 0, 0);
     xpos := xpos + 64;
     end;

{get an entire cursor set and print in the viewport}
xpos := 10; ypos := 75;
GETCURSORSET(myCursors, Offsets, pCursorDataArray, numCurs);
{$R-}
for i := 0 to numCursors-1 do
     begin
     pCursorData^ := pCursorDataArray^[i];
     PUTVIEWPORTRECTANGLE(scrvp2, RRPL, xpos, ypos, 64, 64,
                          RECAST(pCursorData, pVPINTEGERARRAY),
                          Stretch(wordsize(PATTERNMAP)), 4, 0, 0);
        xpos := xpos + 64;
        end;
{$R=}

{set cursor(2) to be cursor(5)}
{$R-}
pCursorData^ := pCursorDataArray^[5];
{$R=}
SETCURSOR(myCursors, 2, pCursorData, unchanged, unchanged);

{set cursor(7) to be a checkerboard gray}
     {$R-}
      for i := 0 to 63 do
```

```
        if (i mod 2)=1 then
             begin
             pcursordata^[i][0] := #125252;
             pcursordata^[i][1] := #052525;
             pcursordata^[i][2] := #125252;
             pcursordata^[i][3] := #052525;
             end
        else begin
             pcursordata^[i][0] := #052525;
             pcursordata^[i][1] := #125252;
             pcursordata^[i][2] := #052525;
             pcursordata^[i][3] := #125252;
             end;
     {$R=}
SETCURSOR(myCursors, 7, pCursorData, unchanged, unchanged);

xpos := 10; ypos := 150;
{$R-}
for i := 0 to numCursors-1 do
     begin
     pCursorData^ := pCursorDataArray^[i];
     PUTVIEWPORTRECTANGLE(scrvp2, RRPL, xpos, ypos, 64, 64,
                          RECAST(pCursorData, pVPINTEGERARRAY),
                          Stretch(wordsize(PATTERNMAP)), 4, 0, 0);
        xpos := xpos + 64;
        end;
{$R=}

my2Cursors := CREATECURSORSET(scrvp2, offsets, pCursorDataArray,
                              numcurs);
SETREGIONCURSOR(scrvp2, 0, my2Cursors, 8, cfXor, true);
end;

{--------------------------}
 Procedure LoadKeysAndPlay;
{--------------------------}
begin
fname := 'tester.keytran';
if FindFileName(fname,'',false) <> success then
        begin
        writeln(fname, ' not found');
        exit(Ttest);
        end;
gr := LOADKEYTABLE(fname, KeyTable);
if gr <> success then
        GRWriteStdError(GR, GR_FatalError, fname);

gr := CONVERTTONEWVERSION(fname, NewKeyTable);
if gr <> success then
        GRWriteStdError(GR, GR_FatalError, fname);

end;
```

```
(* ************************** MAIN ***************************** *)

begin
waithow := KEYWAITEVENT;
FULLWINDOW := GETFULLWINDOW(SAPPHPORT);

{--------------- Emergency Message ----------------}
{Enable emerg msgs}
dum := true;
gr := SOFTENABLE(false, dum);
for x := 1 to 10 do
   begin
   New(pInMsg[x]);
   msgInUse[x] := false;
   end;
New(pRepMsg);
{-------------------------------------------------------}

(* Create Window *)
x := 100;
y := 100;
w := 500;
h := 500;
title := ' ** VP  Window  Listener  Test **';
progName := ' Test ';
winl := CREATEWINDOW(FULLWINDOW, false, x, y, false,
               w, h, true, true, title, progName, true, scrvp);
if winl = NULLWINDOW then exit(Ttest);
x := 100;
y := 100;
w := 500;
h := 500;
win2 := CREATEWINDOW(FULLWINDOW, false, x, y, false,
               w, h, true, true, title, progName, true, scrvp2);
if win2 = NULLWINDOW then exit(Ttest);

{ take icon away from window manager and update it myself }
ICONAUTOUPDATE(win2, false);
GETICONVIEWPORT(win2, iconVp, iconW, iconH);
ENABLENOTIFYEXCEPTIONS(iconvp, DataPort, true, true);
{ Put a checkerboard in it }
VPCOLORRECT(iconVP, rectblack, 0, 0, iconW div 2, iconH div 2);
VPCOLORRECT(iconVP, rectBlack, iconW div 2, iconH div 2, iconW,
            iconH);

(* key Translation *)
LoadKeysAndPlay;
VIEWPORTSTATE(scrvp, x, y, w, h, r, dum, dum, dum);

{ make offscreen vp, and 2 children vp's of the window }
offvp:=MAKEVIEWPORT(scrvp, OFFSCREEN, OFFSCREEN, w, h, 1, TRUE,
                    False, False);
vpkidl:=MAKEVIEWPORT(scrvp, (w div 3), 0, w div 3, h,
                    1, false, False, False);
VPCOLORRECT(vpkidl, RECTINVERT, 0, 0, w div 3, h);
vpkid2:=MAKEVIEWPORT(vpkidl, (w div 3), (h div 3), w div 6,
                    h div 3, 1, false, False, False);
VPCOLORRECT(vpkid2, RECTWHITE, 0, 0, w div 6, h div 3);
```

```
(********** listening *******)
ENABLENOTIFYEXCEPTIONS(scrvp, DataPort, true, true);
ENABLEWINLISTENER(winl, DataPort);
ENABLENOTIFYEXCEPTIONS(scrvp2, DataPort, true, true);
ENABLEWINLISTENER(win2, DataPort);

InitializeFonts;

(**** cursor ***)
fname := 'Tester2.SCursor'; {Cursor file created using Cursdesign}
GR := FindFileName(fname,'',false);
if gr <> success then
       GRWriteStdError(GR, GR_FatalError, fname);
numcursors := 9;
myCursors := LOADVPCURSORS(scrvp, fname, numcursors);

{Create Regions and Set Cursors associated with the regions}
VIEWPORTSTATE(scrvp, x, y, w, h, r, dum, dum, dum);
PUSHREGION(scrvp, regl, 0, 0, w div 2, h);
PUSHREGION(scrvp, reg2, w div 2, 0, w div 2, h);

VIEWPORTSTATE(vpkidl, x, y, w, h, r, dum, dum, dum);
PUSHREGION(vpKidl, reg3, 0, 0, w, h);

VIEWPORTSTATE(scrvp2, x, y, w, h, r, dum, dum, dum);
PUSHREGION(scrvp2, regl, 0, 0, w, h);

VIEWPORTSTATE(scrvp, x, y, w, h, r, dum, dum, dum);
SETREGIONCURSOR(scrvp, regl, myCursors, 0, CFXor, true);
SETREGIONCURSOR(scrvp, reg2, myCursors, 1, CFXor, true);
SETREGIONCURSOR(vpkidl, reg3, mycursors, 2, CFXor, true);

SETREGIONCURSOR(scrvp2, regl, mycursors, 6, CFXor, true);



{main loop}
win := winl;
count :=0;
pushed := 0;
color := false;
repeat
   EvRec := GETEVENT(win, waithow);
   gr := TRANSLATEKEY(KeyTable, evRec, keyEv);
   if gr <> success then
        GRWriteStdError(GR, GR_Error, 'keyev');

   { display character }
    VPCHAR(keyev.vp, vpfont, RXOR, keyev.x, keyev.y, keyev.ch);

    case keyev.ch of
        '0':  { draw a line and invert the screen}
              begin
              VPLINE(scrvp, DRAWLINE, 0, 0, 300, 300);
              if color then
                   color := false
```

```
        else color := true;
        SetScreenColor(scrvp, color);
        end;
'1':  { don't wait }
        waitHow := KEYDONTWAIT;
'2':  { wait for different position }
        waitHow := KEYWAITDIFFPOS;
'3':  { wait for press or keystroke }
        waitHow := KEYWAITEVENT;
'4':  { a new region created over top of part of first
        region }
        begin
        pushed := pushed + 1;
        PUSHREGION(scrvp, reg3, 0, h div 3, w div 4,
                h div 3);
        SETREGIONCURSOR(scrvp, reg3, myCursors, 2, CFXOR,
                TRUE);
        end;
'5':  { the most recently pushed new region deleted }
        begin
        if pushed>0 then
                begin
                pushed := pushed - 1;
                DELETEREGION(scrvp, reg3);
                end;
        end;
'6':  { the new region modified if it exists }
        begin
        if pushed>0 then
                begin
                MODIFYREGION(scrvp, reg3, w div 3 * 2,
                        h div 3, w div 4, h div 3);
                end;
        end;

'7':  { get a rectangle of the screen, put in array,
        then put it back on the screen viewport in a
        different place }
        begin
        GotIt := GETVIEWPORTRECTANGLE(scrvp, 5, 5, 100,
                100, pDataRec, arSize, dummie, 0, 0);
        PUTVIEWPORTRECTANGLE(scrvp, RRpl, 50, 50, 100,
                100, pDataRec, arSize, dummie, 0, 0);
        end;
'8':    begin
        REMOVEWINDOW(win);
        end;
'9':    RESTOREWINDOW(win);
'c':    PlayWithCursors;
'f':    begin
        write('enter char to change:');
        readln(ch);
        PlayWithFonts(ch);
        end;
'l':    Load;
'p':    Pictures;
```

```
        ':  { erase viewport area }
                VIEWCOLORRECT(scrvp, RECTWHITE, 0, 0, w, h);
        '!':    exit(Ttest);
        end; { case }

    count:=count+1;
    if (count mod 5)=0 then
        begin
        { save viewport offscreen, blank onscreen viewport }
        { and refresh with offscreen }
        VIEWPORTSTATE(scrvp, xl, yl, wl, hl, rl, dum, dum, dum);
        VIEWPORTSTATE(offvp, xll, yll, wll, hll, rll, dum, dum,
                    dum);
        VPROP(offvp, RRPL, 5, 5, wl-5, hl-5, scrvp, 5, 5);
        VPCOLORRECT(scrvp, RECTINVERT, 0, 0, wl-5, hl-5);
        for i := 1 to 20000 do ;
        VPROP(scrvp, RRPL, 5, 5, wl-5, hl-5, offvp, 5, 5);
        end;
    if count = 25 then
        begin
        { animate using a font }
        xpos :=100;
        ypos :=100;
        for k := 1 to 10 do
        for i := ord('a') to ord('m') do
                begin
                whichchar := chr(i) ;
                VPPUTCHAR(ScrVp, TalkFont, RRPL, xpos, ypos,
                        whichchar);
                end;
        count := 0;
        end;
until false;
end.
```