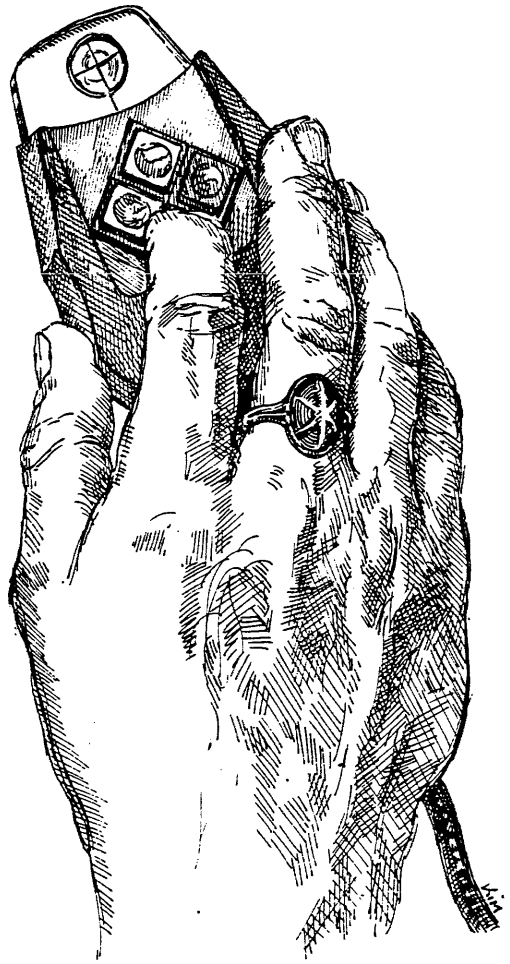
Procedural Interface to the Sapphire Window Manager

25 Aug 84



PERQ Systems Corporation
2600 Liberty Avenue
PO Box 2600
Pittsburgh, PA 15230
(412) 3550-0900

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 Interntional Computer 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.

Table of Contents

1 Introduction

Sapphire is a window manager for the Accènt operating system. (The operating system has also been called Solar, Spice, and Gold at different times by different groups. Accent will be used here to name the operating system.) Sapphire, which stands for SCREEN ALLOCATION PACKAGE PROVIDING HELPFUL ICONS AND RECTANGULAR ENVIRONMENTS, supports a powerful application and user interface. This document describes how Sapphire is used by application programs and *User's Guide to the Sapphire Window Manager* describes how it is used by users. The user interface document contains an introduction to window managers in general for those who are unfamiliar with window managers and the covered window paradigm. It also has an introduction to the Sapphire window manager so it might be appropriate to skim it before reading this document.

Although the window manager is currently one process, the interface to Sapphire is contained in two MatchMaker defs files: Sapph.defs and ViewPt.defs. Sapph.defs is implemented by WinManager.pas and ViewPt.defs is implemented by VPMain.pas, VPKeyRegion.pas, and VPGraphics.pas. The implementation files contain the full procedure headers that describe the procedures. Sapphire will raise two emergency messages in application programs. These are defined by the MatchMaker file SaphEmr.defs. Section 11 contains all the procedure headers from the implementation files, with additional documentation. Section A is an example application program.

In general, Sapphire supports the Covered Window Paradigm, where the screen area is divided into various rectangular areas, called Windows, which may be overlapping, and the graphics and text in one window does not affect the graphics and text in other windows. Sapphire supports graphic operations (which includes text) to windows even if the window is partially covered by other windows. Only the uncovered portions of the graphics will actually be displayed on the screen.

Sapphire also supports *Icons*, which are small pictures that represent windows. Sapphire's icons differ from icons in the Xerox Star or Apple Lisa in three important ways. First, the icons in Sapphire provide useful state information about the process running in the window. Secondly, Sapphire's icons exist for all windows, not just those that have been shrunk. This allows the icons to be used for controlling the window even when it is on the screen, and it allows the extra state information to be viewed by the user at any time. The third difference is that all of Sapphire's icons are collected together in one Sapphire window so that all the Icons can be manipulated as a group.

Sapphire will support either the portrait or landscape monitor on a Perq. It currently will not handle the color monitor at all.

2 Data Types

Sapphire 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.

A *Window* is a rectangular area. It may be on the screen or off the screen. Windows optionally have borders an title lines, and they also optionally have an icon associated with them. Windows also can be manipulated directly by the user via the puck. Each window is composed of one or two viewports.

A *Viewport* is the unit of graphical display in Sapphire. It is the viewport layer that knows how to do graphics to partially covered areas. Each window is composed of one or two (or more) viewports. One viewport is used to display the title line and borders of the window. A second viewport, created just inside the first viewport, is passed to the user to be used for all graphics in the window.

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 can only manipulate the inner viewport and not the border viewport.

In this document *Cursor* will be used to mean the picture that (usually) follows the puck on the screen. In Sapphire, all cursors come in groups called CursorSets. 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. The motivation for CursorSets is that most applications that use cursors will need more than one (Sapphire itself uses over 20, for example), so it is more efficient to store them together.

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. For example, an application can define a number of regions and associate different cursors with different regions. Sapphire would then change the cursor picture automatically 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.

The separation in function between viewports and windows is fairly clean. Viewports do not have any user-interface. Therefore, only windows have title lines, borders, icons, etc. The window package simply uses the viewport's graphics primitives to implement its graphics. Also, the handling of the puck user interface (with the various cursor pictures, etc.) is done entirely by the window manager. The viewport layer is supposed to be able to support a number of different window managers. The separation isn't entirely that clean, but that is the idea.

Since windows have a user interface, the user will be able to move them and change their size. Also, the user will be able to designate which window is to be the *Listener* (the window currently getting user input) by pointing. All Shells will run in windows. A viewport's parameters can only be changed by a program (and not by the user directly). Therefore, viewports are used instead of windows whenever the application program wants to divide up its area into units that only it controls. This might be useful for subdividing a window, for example.

For many of the routines in Sapphire, 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. The procedures that will take these values describe what they will do in the procedure headers (see 11 for details).

3 Creating Windows and 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 reasons for choosing between creating a viewport or a window, and will describe how to do each.

Windows can have borders, title lines, and icons and can be manipulated by the user. Thus, most applications that want an area will use windows. Viewports will typically only be used when an existing window is to be sub-divided into various sections. A window is composed of two viewports, one for the title line and borders and the other for the inner area that is returned to applications. When the application wants to do graphics in a window, it uses the viewport returned by the CreateWindow call (described below).

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 parets' 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.

```
Function CreateWindow(ServPort: 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;
```

CreateWindow is used to create windows. It takes a parent window. You can use Sapphport (returned from GetFullWindow) to make the parent the entire screen. If FixedPosition is true, then the window cannot be moved by the user (or application). If FixedSize is true, then the window cannot have its size changed. These 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. LeftX and topY are the initial coordinates for the window, and width and height are the initial size. These are the coordinates for the OUTSIDE of the entire window (including the border and title, if any). The inside of the window can be calculated using the constants TitleOverhead and BorderOverhead. If any of leftx, topy, width or height have the special value ASKUSER, then the parameters of the window are requested from the user. Any (or all) of these can have this value, and the rest of the parameters are fixed at the specified values. If ASKUSER, the coordinates are set to the actual values specified by the user. The user may choose to not create a window by aborting when asked to specify corners. In this case CreateWindow returns NullWindow and ServPort is NullViewport.

The coordinate system runs from 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.

HasTitle and Hasborder determine whether the window will have a title and border. They are independent. (The border is used to show whether the window is the current Listener). The title line is used for displaying the title text and the UtilProgress progress bar, and it allows the user to give six window manager commands by using the puck. The title string is the initial text to display in the title. The progName is the initial name for the window. Each window has a name which is shown in the icon. The name is guaranteed to be unique so it is modified by appending a digit if it conflicts with another window's name. The viewport returned is the viewport for the inside of the window. This is the viewport to use in all graphic operations to this window. CreateWindow returns the window created.

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 and determines how covered a window is. The lower the rank, the closer to the viewer (and the less covered) a window is. Rank 1 means that the window is not

covered, rank 2 means that the window is covered by one window, etc. A special value for the rank is BOTTOM which means it is covered by all windows. (CreateWindow always creates the window with Rank = 1.) All of the parameters to ModifyWindow can have the special value UNCHANGED which means that they will not be modified. ASKUSER is also a valid parameter value. If ASKUSER, the user may choose to abort when asked to specify corners. In this case ModifyWindow also aborts, returning the existing window.

In addition to creating and modifying windows, Sapphire also has procedures and functions to delete them, change the title, and control the pictures presented in the icon. For a full description, see 11.

```
Function MakeViewport(ServPort: Viewport;
                      x,y,w,h, rank: Integer;
                      memory, courteous,
                      transparent: Boolean): 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. Memory means that the viewport remembers the contents of the viewport at all times by using a special off-screen buffer to hold any covered parts of the viewport. If a viewport is created with memory = true and the x and y values are both the special value OFFSCREEN, then the viewport can serve as a buffer to hold pictures, etc. If the viewport does not have memory, then the application will be responsible for refreshing the viewport when it becomes uncovered. It 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. If a viewport is courteous, it remembers the screen area underneath it. Currently this is intended only for pop-up menus and other short-lived viewports that are under Sapphire's control. (Do not set this parameter to true. Courteous viewports are not fully implemented.) Transparent viewport does not cover the viewports or windows underneath it. 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. 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.

Once a viewport has been created, it can be modified or deleted, or graphics can be done in it. (Do not modify or delete the viewport returned by a window.) Each window or viewport has exactly one owner which is the process that calls the create function. 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.

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, using RasterOp 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 Accent 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 Sapphire routine will have to be called since only Sapphire knows how to do these operations when parts of the viewports are clipped. This optimization, however, will be entirely invisible to most applications since it will be added at the MatchMaker interface.

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

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

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, 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).

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 Sapphire where it will be performed. The Sapphire call for VPROP is ViewRop. Similarly, for the other graphics primitives named VP—, the Sapphire 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.

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

VPColorRect is much more efficient than VPROP when an area is to be set to white or black or

inverted. It takes only one viewport and set of coordinates and makes that rectangle black, white or inverted.

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

VPLine is used for drawing a line in a viewport. The line can be drawn as white, black or XOR.

Fonts in Sapphire are represented as viewports. A special procedure, LoadFont, takes a font name and returns the viewport that represents that font. This special font viewport can then be used in the routines that draw text in a viewport. LoadFont returns NULL.Viewport if the font name is not found. Fonts are allocated out of Accent's physical memory, so that having too many fonts allocated at one time will cause Accent to crash. Since fonts are viewports, they can be easily deallocated using DestroyViewport.

The three text writing procedures come in two forms. The first form 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 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. For each procedure VP—, the corresponding procedure that does not return any information is called VPPut—.

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

VPString is used to draw a string in a viewport. It takes the destination viewport and the font viewport, the RasterOp function to use, and the position for the string. VPString uses the fast StringOp (formerly DrawByte) microcode. It is set up to display as many characters as possible across a line and then report how many characters were displayed. It does not handle wrap-around. Dx is the starting x position. It is set to the x position directly after the last character displayed. Dy is the y position of the bottom of the character. The baseline of the font is ignored. If it has the special value DONTCARE, then the characters will go to the right edge of the viewport. Str is the string to display. FirstCh is the first character of the string to display. If the special value DONTCARE is provided, the value one will be used. LastCh is the last character to display. DONTCARE means to use length(str). LastCh is set to the last character actually displayed which may be less than the specified LastCh if maxX is reached first. 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.

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

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.

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

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 grey patterns which can be created by a program or by using the CursDesign program.

5 Emergency Messages

There are two emergency messages that may be enabled 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 Sapphire's user interface. This emergency message may be converted into a Pascal exception using the MatchMaker interface 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. For applications that do not do any graphics, the typescript manager will handle this, but if the application does do graphics, then only it knows how to refresh the viewport. The emergency message may be

converted into an exception in the same manner as the message for viewport size. 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 Function InValidateMemory as follows (where ra is the rectangle array):

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

In order to get around current MatchMaker restrictions, 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 Sapphire end to actually generate the emergency messages (note that this is backward). Due to a bug in MatchMaker, currently it is not possible to do graphics from the exception handler routine (which is exactly what one would typically want to do). This will hopefully be fixed soon.

6 Cursors, Regions, and Tracking

The viewport layer also handles all tracking and keyboard interface. Cursors come in sets, as described earlier. A cursorSet can be read from a file using the procedure LoadVPCursors. CursorSets can be most easily created using the CursDesign program. They have a special format (which is different from all previous cursor formats). 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 NIL if the file cannot be found. Application programs are not allowed to dereference the cursorSet pointer since the actual data is kept in the window manager process only.

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 with the special two 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.

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:

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

The cursorImage and cursIndex together specify which cursor is to be used whenever the cursor is in that region. CursFunc determines what the cursor function is. The only real 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, unlike POS.) Track determines whether the cursor follows the tablet while this region is active. In general, track will be true, 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.

The way the cursor tracks while the region is active can be controlled using the procedure SetRegionParms:

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

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. If absolute is true, then the upper left corner of the tablet will correspond to the upper left corner of the screen, etc. If absolute is false, then movements on the tablet will be converted into movements on the screen relative to the current position. Speed determines how the movements on the tablet are mapped to movements of the cursor in relative mode. Positive numbers mean that one increment on the tablet will be translated into SPEED increments on the screen. Negative numbers mean that SPEED increments on the tablet will be required before there is one increment on the screen. Speed is applicable only to relative mode. In absolute mode, a number is automatically picked for speed that will allow the entire screen to be addressed using the current tablet. A value of DONTCARE will use the system-chosen default for speed. Minx, maxx, miny, maxy form a rectangle in the region that the cursor is not allowed to leave; thus the cursor will be trapped by this

rectangle. If all are DONTCARE, the cursor is not trapped. As an example, if the minx and maxx values are the same, then the cursor will be restricted to move only vertically. The rest of the parameters are used for gridding. Modx is the grid factor in the x direction. The cursor will only be put on every "modx"th point. DONTCARE means there is no gridding in x. If modx is not equal to DONTCARE, posx determines the offset on which to put the cursor in the x direction. It should be less than modx. DONTCARE for posx means 0 is used. Mody and Posy do the same thing for the y direction.

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 Sapphire and will be much less efficient.

7 Listeners

The Listener is the viewport, window or process that is currently getting keyboard typing. Typically, application programs should never set the listener since users will choose which window should be the listener, using Sapphire's window manager. The window which is the Listener is marked with a special border. Therefore, if an application program wants to change the listener, it should use the window manager call MakeWinListener. This changes the border, etc. and then calls the viewport level procedure SetListener. If an application calls SetListener directly, one window will remain marked as the Listener and the input will be going to another window. Also, due to synchronization problems, it will never be possible to make sure that the application program is not overriding an explicit change listener command given by the user. When a window is made the listener, either by the user or explicitly by an application program, the inner viewport of that window is automatically made the listener viewport.

When a viewport is the Listener, it has complete control over the puck and keyboard. Therefore, no matter where the cursor is, the application program can control it. The regions set up for that viewport control the cursor picture and tracking parameters while the viewport is the listener.

In order to be the listener, a window or viewport must first be enabled. Although there is a procedure at the viewport level to specify a key translation table for a viewport, the procedure EnableWinListener for the window should be used instead because it calls the appropriate procedure for the viewport for that window. Calling the window's procedure rather than the viewport's will allow the user to point at the window to make it the Listener. These enable routines take a key translation table. If the table is not found, then an exception will be generated.

8 Keyboard and Puck Events

Whenever a keyboard key is hit or a puck button goes down or up, an *event* is said to have happened. Sapphire 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 described in another document (KeyTran.Definitions). The key translation mechanism supports prefix keys (such as Control-X in EMACS) and both up and down transitions of the puck. Each viewport that accepts input has a key translation table associated with it. One key translation table may define different commands for the same keyboard or puck event when the cursor is in different regions (regions are discussed in the previous section). 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.

To request a keyboard or puck event, use the procedure GetEvent:

```
Function GetEvent(ServPort: Viewport;
                  howWait: KeyHowWait): KeyEvent;
```

HowWait determines how to wait for the event. If it is KeyDontWait, GetEvent returns immediately with an event. This may be an actual event if one has happened or it may be a position response. Position response come in two forms: POSITION and DIFFPOSITION. DIFFPOSITION is used whenever the point being returned has different coordinates from the last point returned. POSITION is used when the coordinates are the same. These can be mapped to the same abstract command code in the key translation table if desired. If the viewport 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 the key translation table does not define a command for this special event, the application program will wait for the viewport to become the Listener before returning an event.

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 it has changed. KeyWaitDiffPos does not return an event if the viewport is not the Listener. If howWait is KeyWaitEvent, an actual keyboard or puck event is waited for. If howWait is KeyWaitDiffPos or KeyDontWait and an actual keyboard or puck event occurs, that event will be returned instead of a POSITION event. A proposal has been made for a new value for howWait: KeyPollButtons that will return a special event immediately with

the state of the buttons. Since there are events for button down and up (separately), this is not really necessary, but it will probably be added eventually.

Events are queued on per-viewport basis and the translation is not done until the event is requested by the application. This allows the application to change the key translation table for a viewport and have the type-ahead interpreted by the correct table. The LISP community has requested another mode of access where a special set of keys would be sent immediately to a special port. This would require that the key events be translated when typed rather than when requested since it would be the key translation table that specified which keys are special. To solve this problem, both the translated and untranslated keys would be kept around and, if the key translation table were changed, all the already translated events would simply be re-translated.

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 (Default.KText) and is defined as CONTROL-DEL (on PERQ1's) or SETUP (on PERQ2's). The commands that can be given after the prefix key are defined in the Sapphire user-interface document.

9 Errors and Exceptions

Unfortunately, error handling in Sapphire is not well designed currently. When creating or looking up something, 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. Clearly, this needs to be expanded so that applications can differentiate among different errors. Another exception, IMPOSSIBLE, is raised when Sapphire discovers a bug in its internal workings.

10 Initialization and Setup

10.1 Exported Type Definitions

module SapphDefs; ————————————————————————————

Abstract: 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.

Author: Brad Myers, PERQ Systems Corporation

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 and of keyboard type-in. 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 puck on the screen. (2) any symbolic picture which may at some time be connected to the puck. 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.
KeyEvent:	an event generated directly by the user by typing on the keyboard or pressing a puck button, consisting of a command code, a character, and a pointer position.

```
————————————————————————————————————————}

{$Version V1.0 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  = 'Fix13.Kst';
 SysFontHeight = 13;
 SysFontWidth  = 9;

{—— Exported Constants for key translations ——}

 WILDREGION = 31; {no matter what region}

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

{—— EXPORTED types ——}

Type
 VPStr255 = string[255];

 Viewport = Port;

 CursorSet = LONG;

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

  {used in GetEvent}
 KeyEvent = record {not packed since goes into a message}
        Cmd:  0..255;
        Ch:   char;
        region: Integer;
        X,Y:  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 ViewLine}

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

  {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;

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

11 Sapphire Procedure Headers

The following are the procedure headers for all of the procedures that applications might call to access the Sapphire window manager. The procedures are grouped by function, generally in the order in which they are discussed in the procedural interface document.

11.1 Version Number

11.1.1 Function Sapph_Version

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

Abstract: Returns the version number and name of Sapphire as a string 'V1.0'.

Parameter: ServPort - Any window (including SapphPort)

11.2 Windows and Viewports

11.2.1 Function CreateWindow

```
Function CreateWindow(ServPort: 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, no window may be created and NullWindow is returned.

Parameters: ServPort - the window that will be the parent of the new window. The parent is the entire screen if Sapphport is used.
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.
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 - 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.
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. The title area is also used for UtilProgress so if false, there will be no UtilProgress in the window. 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.
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.

11.2.2 Procedure DeleteWindow

Procedure DeleteWindow(ServPort: Window);

Abstract: Deletes a window and all its subwindows. Do not use the window after calling this procedure.

Parameter: ServPort - the window to delete.

11.2.3 Procedure ModifyWindow

Procedure ModifyWindow(ServPort: 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.

Parameters: ServPort - 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.

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

11.2.4 Procedure RemoveWindow

Procedure RemoveWindow(ServPort: 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: ServPort - the window to be removed from the screen.

11.2.5 Procedure RestoreWindow

Procedure RestoreWindow(ServPort: 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: ServPort - the window to be restored to the screen.

11.2.6 Procedure SetWindowError

Procedure SetWindowError(ServPort: 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: ServPort - the window to change the error flag for.
error - the new value of the error boolean.

11.2.7 Procedure SetWindowRequest

Procedure SetWindowRequest(ServPort: 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: ServPort - the window to change the requesting flag for.
error - the new value of the requesting boolean.

11.2.8 Procedure SetWindowAttention

Procedure SetWindowAttention(ServPort: Window;
attn: boolean);

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

Parameters: ServPort - the window to change the attention flag for.
attn - the new value of the attention boolean.

11.2.9 Procedure IdentifyWindow

Procedure IdentifyWindow(ServPort: 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: ServPort - the window to display the relationship for.

11.2.10 Function MakeViewport

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

Abstract: Create a new viewport.

Parameters: ServPort - the parent viewport. Should be NIL only for the very first viewport. The new viewport will be clipped inside of the parent viewport. To make this global inside the screen, use Full-Viewport.
x,y - the upper left corner of the new viewport with respect to ServPort. 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 sons of ServPort. 1 means that the new viewport covers all others and BOTTOM (or any very large number) means that all other sons 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. 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.
transparent - whether this viewport covers viewports it is on

top of. Most viewports will not be transparent. If a viewport is transparent then updates to viewports underneath it will show through.

Returns: The viewport created.

11.2.11 Procedure DestroyViewport

Procedure DestroyViewport(ServPort: 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: ServPort - the viewport to destroy.

11.2.12 Function GetVPRank

Function GetVPRank(ServPort: Viewport): Integer;

Abstract: Returns the rank of ServPort w.r.t. 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: ServPort - the viewport.

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

11.2.13 Procedure ViewportState

Procedure ViewportState(ServPort: Viewport;
var curIx, curty, curwidth, curheight, curRank: Integer;
var memory, courteous, transparent: boolean);

Abstract: Returns a description of ServPort.

Parameters: ServPort - 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.

11.2.14 Procedure ModifyVP

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

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

Parameters: ServPort - 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.

11.2.15 Function GetFullViewport

Function GetFullViewport(ServPort: Viewport): Viewport;

Abstract: Returns the viewport for the full screen.

Parameters: ServPort - ignored.

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

11.2.16 Procedure ReserveScreen

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

Abstract: Pretends thatServPort is not covered. Operations are still
      clipped to be inside ServPort, 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: ServPort - 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 ServPort dies, the
        screen is automatically un-reserved.

Errors: if reserve is true but screen is already reserved for some viewport.
```

11.2.17 Procedure GetScreenParameters

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

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

Parameters: ServPort - the window to get parameters for,
      width - width of screen
      height - height of screen
```

11.2.18 Procedure SetWindowTitle

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

Abstract: Sets the title of ServPort to be new string. If window has no title
      then this is a no-op. If the title is too long, then clipped.
```

Parameters: ServPort - the window to set the title of.
title - the new title.

11.2.19 Function GetFullWindow

Function GetFullWindow(ServPort: Window): Window;

Abstract: Returns the full window. DO NOT MODIFY This window in any way.

Parameters: ServPort - ignored. (Needed for message passing).

Returns: The window that is the full screen.

11.2.20 Procedure SetWindowName

Procedure SetWindowName(ServPort: Window;
var progName: ProgStr);

Abstract: Changes the progname for the window. Displays the new progname in the icon.

Parameters: ServPort - 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.

11.2.21 Procedure FullWindowState

Procedure FullWindowState(ServPort: 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 ServPort.

Parameters: ServPort - 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.

11.2.22 Procedure SetWindowProgress

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

Abstract: Shows progress in icon and title area if appropriate.
If max is 0 then random progress is done. If value >= Max then the progress bar is removed.

Parameters: ServPort - the window to display the progress for.
nestLevel - which utilProgress bar to show.
value - the current value.
max - the maximum value.

Errors: If nestLevel > NumProgressBars or < 1.

11.2.23 Procedure GetWinNames

```
Procedure GetWinNames(ServPort: Window;
            var names: pWinNameArray;
            var namesCnt: Long;
            var curListenIndex: Integer);
```

Abstract: Creates an array of names of all the current windows.

Parameters: ServPort - ignored.
names - storage is allocated and filled with all the names of the windows.
namesCnt - number of names

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.

11.2.24 Function WinForName

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

Abstract: Returns the window for a name.

Parameters: ServPort - ignored.
name - the name of the window to get.

Returns: the window refered to by name or NullWindow if name is not a legal window.

11.2.25 Procedure WindowViewport

```
Procedure WindowViewport(ServPort: 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 ServPort field.
Parameters: ServPort - 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.

11.2.26 Procedure DefineFullSize

```
Procedure DefineFullSize(ServPort, 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 changes size, the meaning of full will automatically change.

Parameters: ServPort - any valid window (ignored).
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.

11.2.27 Procedure ExpandWindow

Procedure ExpandWindow(ServPort: 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: ServPort - a window that is to be expanded.

11.2.28 Procedure ShrinkWindow

Procedure ShrinkWindow(ServPort: Window);

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

Parameters: ServPort - a window that is to be shrunk.

11.2.29 Function GetWinProcess

Function GetWinProcess(ServPort: Window): Port;

Abstract: Returns the port for the window. This is the port passed to the window manager when the window was enabled to be the listener. This port can be used for sending emergency messages about control characters.

Parameters: ServPort - the window to get the process for.

Returns: The port for the window or NullPort if no port specified.

11.2.30 Function WinForViewPort

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

Abstract: Returns the Window for the specified viewport.

Parameters: ServPort - any valid window; ignored.
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.

11.3 Icons

11.3.1 Procedure CompactIcons

Procedure CompactIcons(ServPort: Window);

Abstract: Compacts and redisplays the icons, removing any empty spaces.

Parameters: ServPort - a window that is ignored.

11.3.2 Procedure IconAutoUpdate

Procedure IconAutoUpdate(ServPort: 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 Err, Req, 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: ServPort - the window whose icon should not be updated.

11.3.3 Procedure GetIconViewport

Procedure GetIconViewport(ServPort: 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: ServPort - 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(ServPort, false) not called first.

11.3.4 Procedure DeAllocIconVP.

Procedure DeAllocIconVP(ServPort: Window);

Abstract: Eliminates the VP for the icon window. This erases the icon and redraws it. Does not Allow IconAutoUpdate.

Parameters: ServPort - the window whose icon viewport is to be destroyed.

11.3.5 Function GetIconWindow

Function GetIconWindow(ServPort: Window): Window;

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

Parameters: ServPort - ignored. (Needed for message passing).

Returns: The window that contains the icons.

11.4 Graphics Primitives

11.4.1 Procedure VPROP

```
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 exposed exception if portions in destination are not available in source. 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.

Design: Simply calls DoRop after figuring out the rasterOp direction.

11.4.2 Procedure VPColorRect

```
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 MUCH 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.

11.4.3 Procedure VPScroll

Procedure VPScroll(destvp: Viewport;
x, y, width, height, Xamt, Yamt: Integer);

Abstract: Scrolls a portion of a viewport up or down 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, Yamt - number of bits to move the area; negative numbers to move up and positive numbers to move down.

11.4.4 Procedure VPLine

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.

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: Due to the current microcode, the line may have holes in it.

11.4.5 Procedure VPString

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 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. VPPutString is similar to this procedure but it does not have the return values.

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 (bottom, left corner) 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.

Design: Uses GenLine to write text to a buffer using the microcode StringOp
and then uses normal ViewRop from there.

11.4.6 Procedure VPChArray

```
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 set array of characters.
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 (bottom, left corner)
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 (or maxx) was reached.

Errors: if fontVP is not a font.
If fontHeight is too big for the special buffer.

Design: Uses GenLine to write text to a buffer using the microcode StringOp
and then uses normal ViewRop from there.

11.4.7 Procedure VPChar

```
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 (bottom, left corner) 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.

Design: Does a ViewRop directly from the font onto the viewport so the text buffer is not used.

11.4.8 Procedure VPPutString

```
Procedure VPPutString(destvp, fontVP: Viewport;
          funct: RopFunct;
          dx, dy: Integer;
          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 (bottom, left corner) 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.

Design: Uses GenLine to write text to a buffer using the microcode StringOp and then uses normal ViewRop from there.

11.4.9 Procedure VPPutChArray

```
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 (bottom, left corner) 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.

Design: Uses GenLine to write text to a buffer using the microcode StringOp and then uses normal ViewRop from there.

11.4.10 Procedure VPPutChar

```
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 (bottom, left corner) 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.

Design: Does a ViewRop directly from the font onto the viewport so the text buffer is not used.

11.4.11 Procedure VPtoScreenCoords

```
Procedure VPtoScreenCoords(ServPort: 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: ServPort - the viewport the point is with respect to.
x,y - the coordinates of the point w.r.t. ServPort.
scrX, scrY - set to the corresponding point w.r.t. the screen.

11.4.12 Procedure ScreenToVPCoords

```
Procedure ScreenToVPCoords(ServPort: 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: ServPort - the viewport the point is inside of.
scrX, scrY - the point w.r.t. the screen.
x,y - set to the corresponding point in ServPort.

11.4.13 Function LoadFont

Function LoadFont(ServPort: Viewport;
fileName: VPStr255): Viewport;

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

Parameters: ServPort - ignored. (needed for message passing).
fileName - the string name of the font (including any extensions).
A normal file lookup is used to find the file.

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

Errors: if cannot allocate memory for font.

11.4.14 Procedure FontSize

Procedure FontSize(ServPort: 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: ServPort - a viewport representing a font.
name - set to the name of the font (e.g., "Helvetica" or "Computer Modern Roman")
size - 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 lower 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 ServPort is not a font.

11.4.15 Procedure FontCharWidthVector

Procedure FontCharWidth(ServPort: Viewport;
ch: char;
var dx, dy: Integer);

Abstract: Returns the width of the specified character.

Parameters: ServPort - 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 of the next character to be drawn.

Errors: if ServPort is not a font.

11.4.16 Function GetSysFont

Function GetSysFont(ServPort: 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: ServPort - ignored. (needed for message passing).

Returns: The font viewport for the standard system font.

11.4.17 Function FontStringWidthVector

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

Abstract: Returns the width of the specified portion of the string.

Parameters: ServPort - 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 - the width vector for the string portion: the displacement from the first character origin for the next character to be drawn after the string.

Returns: the width of str portion in screen pixels.

Errors: if ServPort is not a font.

11.4.18 Function LoadVPPicture

```
Function LoadVPPicture(ServPort: 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: ServPort - ignored. (needed for message passing).
fileName - the string name of the picture file (including any extensions). A normal file lookup is used to find the file. The picture is read starting at block 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.

11.4.19 Procedure PutViewportBit

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

Abstract: Set or clear a particular bit of the viewport.

Parameters: ServPort - the viewport to put the bit in
x, y - location of the bit
value - TRUE to set the bit, FALSE to clear the bit

11.4.20 Function GetViewportBit

```
Function GetViewportBit(ServPort: 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: ServPort - the viewport to get the bit from
x, y - location of the bit
value - set to TRUE if the bit is set; if not, set to FALSE.

Returns: TRUE if the bit is available, FALSE if not

11.4.21 Procedure PutViewportRectangle

```
Procedure PutViewportRectangle(ServPort: Viewport;
                               funct: Ropfunct;
                               x, y, width, height: integer;
                               Data: pVPIntegerArray;
                               DataCnt: Long;
                               WordsAcross, 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: ServPort - 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
DataCnt - 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

11.4.22 Function GetViewportRectangle

```
Function GetViewportRectangle(ServPort: Viewport;
                x, y, width, height: Integer;
              var data: pVPIntegerArray;
              var DataCnt: 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: ServPort - 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.
DataCnt - 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 if the rectangle is available, False if it is not.

11.5 Emergency Messages

11.5.1 Procedure EnableNotifyExceptions

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

Abstract: Allow 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 if 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: ServPort - 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 the any part of the viewport is exposed. If false, then disabled.

11.6 Cursors, Regions, and Tracking

11.6.1 Function LoadVPCursors

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

Abstract: Load a cursor set into memory so that it can be used to set the cursor to.

Parameters: ServPort - ignored.
fileName - the name of the file to read the cursors from.
numCursors - set to the number of cursors found in the file.

Returns: a reference that can be used when setting the cursor or NIL if file not found.

Errors: If file does not seem to be a valid cursor file.
NOTE: CURSORS SHOULD PROBABLY BE CHANGED TO BE VIEWPORTS (SO NO SPECIAL CURSORSET TYPE). HANDLE LIKE FONTS.

11.6.2 Procedure DestroyVPCursors

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

Abstract: Deallocates the cursors. WARNING: Do not destroy a cursorSet while it is still being used by any region.

Parameters: ServPort - the cursors to deallocate. Better not use them again.

11.6.3 Procedure ReserveCursor

```
Procedure ReserveCursor(ServPort: Viewport;
            reserve: boolean);
```

Abstract: Prevents the down presses on the cursor from being interpreted by

the window manager. All down presses go to ServPort when ServPort is the Listener. This is turned off by calling with reserve = false or when ServPort is destroyed. This is independant of screen reserving function. If ServPort is not the listener, then it is made the listener.

*** NOT YET IMPLEMENTED ***

Parameters: ServPort - the viewport 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, if ServPort cannot be the listener.

11.6.4 Procedure SetCursorPos

```
Procedure SetCursorPos(ServPort: 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: ServPort - the viewport to set the cursor for.
x,y - the position for the cursor with respect to ServPort.

11.6.5 Procedure SetRegionCursor

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

Abstract: Specifies the cursor to be used in a region.

Parameters: ServPort - 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. NIL 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.
track - whether the cursor should follow the puck or not.

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

11.6.6 Procedure GetRegionCursor

```
Procedure GetRegionCursor(ServPort: 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: ServPort - 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 puck and false if not.

Errors: Raises userError if region not there.

11.6.7 Procedure SetRegionParms

```
Procedure SetRegionParms(ServPort: Viewport;
            regionNum: Integer;
            absolute: boolean;
            speed, 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: ServPort - 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 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 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.

11.6.8 Procedure GetRegionParms

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

Abstract: Returns the cursor movement for a region.

Parameters: ServPort - 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.

11.6.9 Procedure PushRegion

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

Abstract: Pushes a new copy of regionNum onto ServPort. If regionNum is already a region of ServPort, 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 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: ServPort - 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 viewport or OUTREGION.

11.6.10 Procedure ModifyRegion

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

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

Parameters: ServPort - 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.

11.6.11 Procedure DeleteRegion

Procedure DeleteRegion(ServPort: 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: ServPort - the viewport containing the region.
regionNum - the region to delete.

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

11.6.12 Procedure DestroyRegions

Procedure DestroyRegions (ServPort: viewport);

Abstract: Deallocates all the regions for ServPort and each region's rectangle list.

11.7 Listeners

11.7.1 Procedure EnableWinListener

```
Procedure EnableWinListener(ServPort: Window;
                abortPort: Port;
                keytrantab: VPStr255;
                timeOut: Integer);
```

Abstract: Allows the window to be the Listener and allows the viewport for the window to take input. Calling this procedure will generate a changed listener event which is sent to the inner viewport of the window.

Parameters: ServPort - the window that can be the Listener.
abortPort - the Kernel port of the process that owns the window. This is used to send ↑Del emergency messages to when the window gets ↑C type aborts.
keytrantab - the name of the key translation table to use. If empty ('') then the system default key translation table is used.
timeout - the number of "ticks" to wait before returning a timeout event. 0 means wait forever.
***Timeout NOT IMPLEMENTED YET**

Calls: EnableInput on the Viewport for this window.

11.7.2 Procedure SetListener

```
Procedure SetListener(ServPort: Viewport);
```

Abstract: Specifies a new viewport to be the listener. All subsequent key events will go to the new viewport. If no viewport had been the listener, then all events queued in the Null queue are put at the end of ServPort's queue. NOTE: Applications cannot use the FullViewport as the Listener. It is reserved for use by the window manager itself.

Parameters: ServPort - the viewport to be the listener. If NULLViewport then the listener is set to no viewport and events are saved in a special queue for the next real viewport to be the

listener. EnableWinListener for this viewport's window and EnableInput for this viewport must have been called before calling SetListener.

11.7.3 Procedure MakeWinListener

Procedure MakeWinListener(ServPort: Window);

Abstract: Changes the Listener to the ServPort window. ServPort can be NullWindow in which case there will be no listener.

Parameters:ServPort - the window to be the new Listener.

11.7.4 Function GetListenerWindow

Function GetListenerWindow(ServPort: Window): Window;

Abstract: Returns the Window for the listener.

Parameters: ServPort - ignored.

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

11.7.5 ProcedureEnableInput

Procedure EnableInput(ServPort: Viewport;
keytrantab: VPStr255;
timeout: Integer);

Abstract: Allows the Listener to be set to the specified viewport. Also specifies the Key translation table to be used.

Parameters: ServPort - the viewport to be allowed to be the Listener.
keytrantab - the name of the key translation table to use. If empty (") then the system default key translation table is used.
timeout - the number of "ticks" to wait before returning a timeout event. 0 means wait forever. **NOT IMPLEMENTED YET**

11.8 Keyboard and Puck Events

11.8.1 Function GetEvent

Function GetEvent(ServPort: Viewport;
howWait: KeyHowWait): KeyEvent;

Abstract: Returns the next keyboard or puck 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 ServPort is not the Listener, then will wait for ServPort to be the Listener before returning unless howWait is KeyDontWait in which case returns the NoEvent event.

Parameters: ServPort - the viewport that an event is wanted for.
howWait - determines how to wait for the event.

Returns: a key event record for the event.

Errors: if input has not been enabled for this viewport.

11.8.2 Function FlushEvents

Function FlushEvents(ServPort: viewport): boolean;

Abstract: Flushes all queued events for a viewport.

Parameters: ServPort - the viewport to flush.

Returns: True if any events were outstanding, false otherwise.

11.8.3 Procedure GetEventPort

```
Procedure GetEventPort(ServPort:   Viewport;
            howWait:   KeyHowWait;
            retPort:   Port);
```

Abstract: Does a GetEvent from a named port. Allows asynchronous event receives from more than one viewport.

11.8.4 Procedure ExtractEvent

```
Procedure ExtractEvent(repMsg:   Pointer;
            var vp:      viewport;
            var k:       KeyEvent)
      : Boolean;
```

Abstract: Allows asynchronous event receives from more than one viewport.

Appendix A. Sample Sapphire Application Program

In order to use Sapphire, the application program must import Sapph from Sapphuser and Viewpt from ViewPtUser for the Matchmaker interface.

If handling emergency messages, import SaphEmrServer and SaphEmrExceptions.

To get the full window port, call GetFullWindow(Sapphport).

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

 Change Log

 17-jan-84  V0.1  Amy Butler   Change to run under Accent/Sapphire
                    Created
-------------------------------------}

Imports Sapph from SapphUser;
Imports ViewPt from ViewPtUser;

Imports PascalInit      from PascalInit;
Imports PathName        from Pathname;
Imports AccInt          from AccentUser;   {using SoftInterrupt}
Imports Except          from Except ;      {to get emerg msg exception}
imports AccCall         from AccCall;      {for receive}
Imports SaphEmrServer   from SaphEmrServer; {figure out which emerg msg}
Imports SaphEmrExceptions from SaphEmrExceptions; {the exceptions}

Imports TesterKDefs from TesterKDefs;

var
  wins: Array[1..256] of window;
  fullWindow, iconWindow: Window;
  gr: GeneralReturn;
  waithow: KeyHowWait;
  scrvpx, scrvpy: integer;
  hasTit, hasBor, fixedSize, fixedPos: boolean;
  title: TitStr;
  progName: ProgStr;
  strToPrint:VPStr255;
  win: Window;
  vp, scrvp, vpfont, SysFontVP: Viewport;
```

```
  names: pWinNameArray;
  keyEv: KeyEvent;
  myCursors: CursorSet;
  fname,f2name : path←name;

  num: Long;
  dum, pushed: boolean;
  c: Char;
  count, dummie:integer;
  x,x1,y,y1,w,w1,h,h1,x2,y2,x3,curWin, r, r1, minW, minH: Integer;
  x11, y11, w11, h11, r11:integer;

label

{————————————————————————————————————}
{                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 the fixed }
     case boolean of
      true: (head   : Msg;       { portion of all 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 WriteLn('** 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 **)
 if SoftInterrupt(kernelPort, false, dum) <> success then
   WriteLn('** failed');
 end;

Handler EViewPtChanged(vp1: ViewPort; x1,y1,w1,h1,r: Integer);
 var i: integer;
 begin
 for i := 1 to (w1 div 2) - 1 do
  ViewColorRect(vp1, RectInvert, i*2, 0, 1, h1);

 if r=1 then  { if top window }
     VPROP(vp1, RRPL, x1, y1, w1, h1, vp, x1, y1);

 w := w1;
 h := h1;
 ModifyRegion(vp1, reg1, 0, 0, w div 2, h);
 ModifyRegion(vp1, reg2, w div 2, 0, w div 2, h);
 if pushed then
    ModifyRegion(vp1, 5, 0, h div 3, w div 4, h div 3);
 end;

Handler EViewPtExposed(vp1: Viewport; ra: pRectArray; numRectangles: Long);
 var i,j: Integer;
 begin
 for i := 1 to Shrink(numRectangles) do
  with ra↑[i] do
   begin
   VPROP(vp1, RRPL, lx, ty, w, h, vp, lx, ty);
```

```
    {ViewColorRect(vp1, RectInvert, lx, ty, w, h);}
    for j := 1 to 30000 do;
    end;

 if InValidateMemory(KernelPort, recast(ra, VirtualAddress),
   wordsize(ra)*2) <> Success then
      WriteLn('** failed to deallocate memory');
 end;

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

Handler Impossible(s: String);
 begin
 WriteLn(chr(7), 'IMPOSSIBLE ***** ',s);
 goto 1;
 end;


Handler UserError(s: String);
 begin
 WriteLn(chr(7), 'USER ERROR ***** ',s);
 goto 1;
 end;
(********************************* MAIN *********************************)
begin

waithow := KeyWaitEvent;

fullWindow := GetFullWindow(sapphport);

{--------- Emergency Message stuff ---------}
Writeln('Enable emerg msgs ');
dum := true;
if SoftInterrupt(kernelPort, false, dum) <> success then
  WriteLn('** failed')
else writeln('old value = ',dum);
for x := 1 to 10 do
  begin
  New(pInMsg[x]);
  msgInUse[x] := false;
  end;
New(pRepMsg);
{-----------------------------------}

curWin := 1;
```

```
for x := 1 to 256 do
 wins[x] := NULLPort;

(* Create Window *)
x := ASKUSER;
y := ASKUSER;
w := ASKUSER;
h := ASKUSER;
title := ' VP offscreen Window test ';
progName := '';
win := CreateWindow(FullWindow, false, x,y, false,
          w,h,true, true, title, progName, true, scrvp);
if win = NullWindow then exit(Rtest);

(* key Translation *)
fname := 'tester.keytran';
if FindFileName(fname,'',false) <> success then
     begin
     writeln(fname, ' not found');
     exit(Rtest);
     end;

ViewportState(scrvp, x, y, w, h, r, dum, dum, dum);

r:=1;
vp:=MakeViewport(scrvp, OFFSCREEN, OFFSCREEN, w, h, r, TRUE, False, False);
ViewportState(scrvp, x, y, w, h, r, dum, dum, dum);

{SysFontVP := GetSysFont(scrvp);}
{Get Font file}
f2name := ':accent>steve>hbrw35.kst';
if FindFileName(f2name,'',false) <> success then
     begin
     writeln(f2name, ' not found');
     {exit(Rtest);}
     end;
vpfont := LoadFont(scrvp, f2name);
EnableNotifyExceptions(scrvp, DataPort, true, true);

EnableWinListener(win, DataPort, fname, 0);

{GetCursors}
fname := 'Tester.SCursor';
if FindFileName(fname,'',false) <> success then
     begin
     writeln('Rtest: ',fname, ' not found');
     exit(Rtest);
```

```
    end;
myCursors := LoadVPCursors(scrvp, fname, x);

{Create Regions and Set Cursors associated with the regions}
PushRegion(scrvp, reg1, 0,0,w div 2, h);
PushRegion(scrvp, reg2, w div 2,0,w div 2, h);

SetRegionCursor(scrvp, reg1, myCursors, 0, CFXor, true);
SetRegionCursor(scrvp, reg2, myCursors, 1, CFXor, true);

{main loop}
1:
count :=0;
scrvpx := 5;
pushed := false;
repeat
  keyEv := GETEVENT(scrvp, waitHow);
  if keyEv.x < 10 then keyEv.x := 10;
  if keyEv.y < 10 then keyEv.y := 10;
  if keyEv.x > w-9 then keyEv.x := w - 9;
  if keyEv.y > h-13 then keyEv.y := h - 13;
  x2 := 0; x3 := 0;
  { display character }
  VPCHAR(scrvp, vpfont, RNOT, x2, x3, chr(keyEv.cmd));
  VPCHAR(scrvp, vpfont, RXOR, keyEv.x, keyEv.y, keyEv.ch);

  if keyEv.ch = '1' then
     { don't wait }
     waitHow := KeyDontWait
  else if keyEv.ch = '2' then
     { wait for different position }
     waitHow := KeyWaitDiffPos
  else if keyEv.ch = '3' then
     { wait for press or keystroke }
     waitHow := KeyWaitEvent

  else if keyEv.ch = '4' then
     begin
     { a new region created over top of part of first region}
     pushed := true;
     PUSHREGION(scrvp, 5, 0, h div 3, w div 4, h div 3);
     SETREGIONCURSOR(scrvp, 5, myCursors, 2, CFXOR, TRUE);
     end
  else if keyEv.ch = '5' then
     begin
     { the new region deleted }
     pushed := false;
```

```
    DELETEREGION(scrvp, 5);
    end
  else if keyEv.ch = '6' then
    begin
    { the new region modified if it exists }
    if pushed then
        begin
        MODIFYREGION(scrvp, 5, w div 3 * 2, h div 3, w div 4,
                                h div 3);
        end;
    end
  else if keyEv.ch = ' then
    { erase screen }
    ViewColorRect(scrvp, RectWhite, 0, 0, w, h)
  else if keyEv.ch = '!' then
    exit(Rtest);

  count:=count+1;
  if count > 5 then
    begin
    { save viewport offscreen, blank onscreen viewport }
    { and refresh with offscreen }
    ViewportState(scrvp, x1, y1, w1, h1, r1, dum, dum, dum);
    ViewportState(vp, x11, y11, w11, h11, r11, dum, dum, dum);
    VPROP(vp, RRPL, 5, 5, w1-5, h1-5, scrvp, 5, 5);
    VPColorRect(scrvp, RectWhite, 0, 0, w1-5, h1-5);
    for count := 1 to 20000 do ;
    VPROP(scrvp, RRPL, 5, 5, w1-5, h1-5, vp, 5, 5);
    count:=0;
    end;
  if count = 4 then
    begin
    { draw a line, print a string using VPfont }
    VPLine(scrvp, drawline, 50, 250, 100, 250);
    if scrvpx > w1-100 then scrvpx:=5
              else scrvpx := scrvpx + 3;
    scrvpy:=250;
    strToPrint:='Perq';
    dummie:=dontcare;
    ViewString(scrvp, vpfont, RRPL, scrvpx, scrvpy, strToPrint,
        dummie,dummie);
    VPString(scrvp, vpfont, RRPL, scrvpx, scrvpy, strToPrint,
        dummie, dummie);
    end;
until false;
end.
```

Appendix B. Key Translation

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.

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. There may be blank lines.
- 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 normal control a. 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                          ! the definitions section is optional
if no
                                     ! commands or regions need to be
named.

region <name> = <2..30>              ! names are 1..25 characters in
length;
      <name> = <2..30>               ! case is preserved but irrelevant;
                                     ! all names must be unique.
command <name> = <2..255>
       <name> = <2..255>


                  ! 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

                  ! predefined command names are STDCOMMAND
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>]  ! 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.


                        ! 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
```

DEFINING PREFIXES

Before using a prefix in other <key desc>s, 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 code>s if they are to have different effects.

The ANY <key desc> is used with prefixes to specify that any char- acter 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.

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
GPIB
                     [MIDDLE] * [LEFT]  * [RIGHT] <keyboard key>   !for
KRIZ

<keyboard key> == [CONTROL] <key> |
                  [CONTROL] <0..127> |
                  <0..255>

<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 irrespective of control
                            ! or shift, e.g C for ↑SHIFT-C or ↑c or 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 key board.

          | 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
                      N0 | N1 | N2 | N3 | N4 | N5 | N6 | N7 | N8 | N9 |
                      PF1 | PF2 | PF3 | PF4 |
                      ENTER | NCOMMA | NMINUS | NPERIOD


<special actions> == ! blue no equivalent with Kriz tablet
                     BLUEDOWN | BLUEUP |

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

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

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

                     REGIONEXIT | TIMEOUT | POSITION | DIFFPOSITION |
                     NOEVENT | LISTENER
```

Example .KTEXT file

```
DEFINITIONS

region reg1 = 3
       reg2 = 4
       reg3 = 5
command

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
```

index