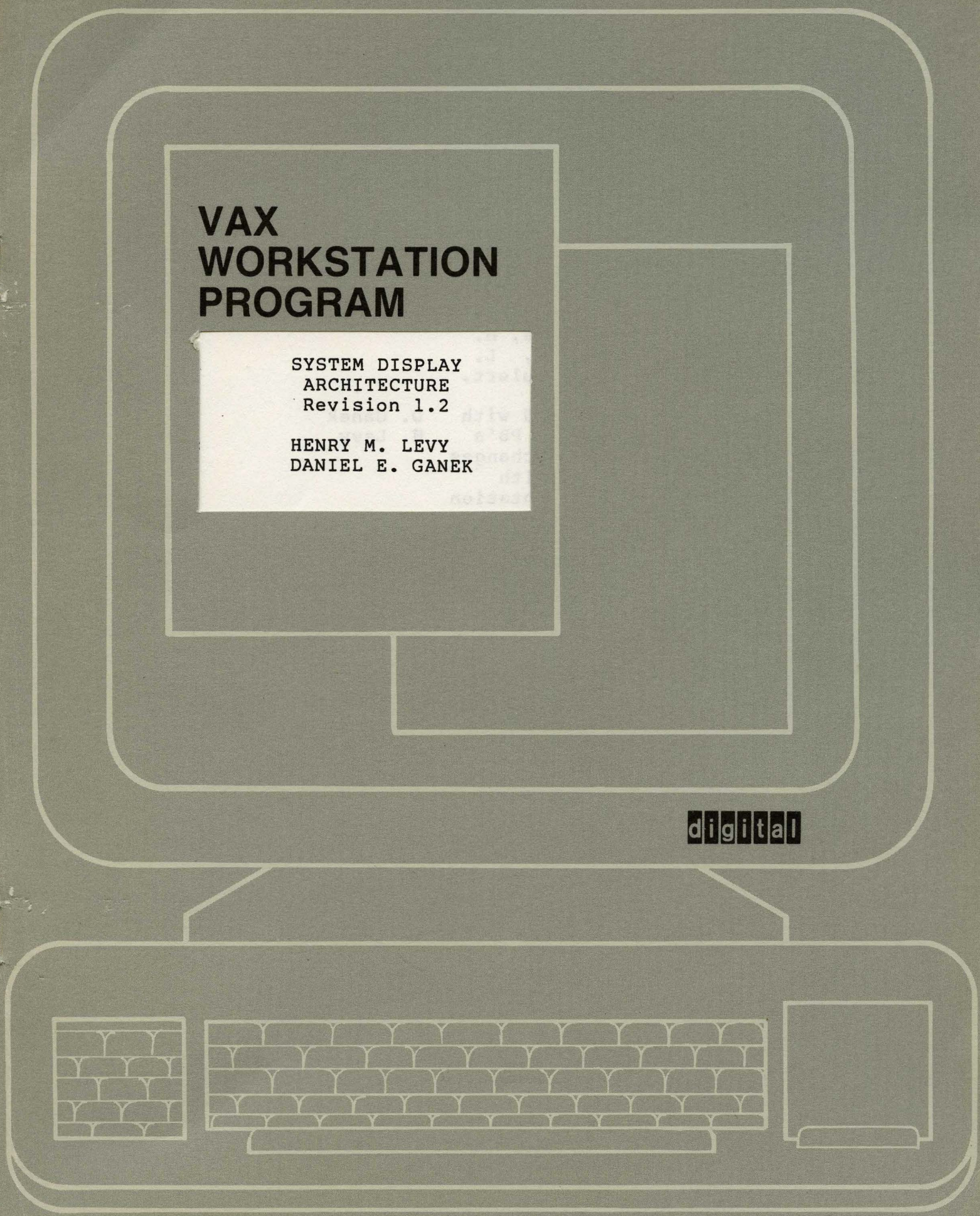
VAX WORKSTATION PROGRAM

SYSTEM DISPLAY
ARCHITECTURE
Revision 1.2

HENRY M. LEVY
DANIEL E. GANEK



Date: 27 Oct 82

File: EUBIE::DOC$PUBLIC:SDA.MEM

Title: SYSTEM DISPLAY
ARCHITECTURE
Revision 1.2

Author: HENRY M. LEVY
DANIEL E. GANEK

Abstract:

This document describes the System Display Architecture, a high-level programming interface to display terminals, including input, output, and display space management.

The new high-resolution, bit-mapped display terminals have more sophisticated capabilities than existing ASCII terminals and thus require a different programming interface. First, the possibility of full-page or larger displays allows sharing of the screen by several simultaneous activities. The user of the display is allowed to arrange the activities on the screen as he or she wishes in a manner largely transparent to the application programs. Second, the terminals support graphics, multiple fonts, proportionally spaced text, and digitized images.

Because the display screen and keyboard can be multiplexed among independent processes, applications cannot have direct access to the display hardware. In addition, programs must be able to operate on a family of display processors with varying characteristics. Therefore, the goal of this architecture is to provide a high-level procedural interface for display control, isolating the programmer from the characteristics and current state of the particular display device. This document defines an architectural model for the control and programming of display devices, and the interface seen by application programs. A model for layering of the system and emulation of existing terminal devices is also provided.

Revision:	Description	Author	Date
Ø	Prelim. design with D. Ganek, K. Lefebvre	H. Levy	5-Feb-1982
1	First Revision with K. Lefebvre, E. Osman, L. Samberg. This revision was for internal review only	D. Ganek	1-May-1982
1.1	Rev. 1 modified with additional comments from K. Lefebvre, H. Levy, E. Osman, L. Samberg, A. Schulert.	D. Ganek	12-Aug-1982
1.2	Rev 1.1 modified with expanded Intro, PB's on PB's, minor changes to align spec with current implementation	D. Ganek H. Levy	

TABLE of CONTENTS

PART II - SYSTEM SPECIFICATION

CHAPTER 6 VIRTUAL DISPLAY SERVICES OPERATIONS

CHAPTER 7 VIRTUAL SCREEN SERVICES OPERATIONS

PART I

SYSTEM DESCRIPTION

CHAPTER 1

INTRODUCTION

With new video technologies and an increase in computing cycles available to individual users, interest in interactive systems and the human interface has substantially increased. As a result, a new generation of high-resolution raster display systems is replacing the standard video terminal in many engineering, programming, and office applications. Such displays are characterized by a relatively large (typically 15" to 19" diagonal) display surface containing on the order of 1000 x 1000 addressable picture elements at a resolution of 70 to 90 elements per inch.

High-resolution raster systems allow the user to create, view, and manipulate images containing graphics, multiple type fonts, and pictures either separately or within a single document or activity [Newman and Sproull 79, Ingalls 81, Foley and vam Dam 82]. These capabilities are made possible by the high-resolution format. Another powerful concept typically available on such displays is that of the window management system [Kay and Goldberg 77, Teitelman 79, Lantz and Rashid 79, Meyrowitz and Moser 81]. A window management system allows multiple independent processes to share the display screen, the output from each process appearing in a rectangular area known as a window. Windows allow the user to coordinate the activities of multiple processes, responding to those that he or she chooses, and moving from activity to activity as the need arises (for example, leaving an editing window to read and reply to mail in a mail window, then returning to the editing session). Although window systems have been implemented on more traditional display terminals [McCrossin et al 78], the full potential of window systems is better realized on the larger screen area provided by newer displays.

1.1 THE GOALS OF A WINDOW SYSTEM

Unlike an ordinary video terminal, which is generally used by only one process at a time, a display is a shareable device. Sharing of the display implies, as it does for example with a disk, that user programs cannot directly read or write arbitrary

areas of the device as they please. Some intelligence must control device access in order to prevent chaos. Thus, the primary goal of the windowing system is to manage the use of physical display space. The window manager partitions the available screen space among some number of processes.

A second goal of the window management system is to provide primitives for application programs to construct images. That is, the window manager establishes an interface that allows programs to write text, graphics, and pictures to the screen. This interface can be specified at many possible levels. For example, it can export only the operations available on the physical display hardware, or it can implement a higher level interface involving more complex abstractions and operations. Some number of interfaces can be provided for compatibility, as the system might allow programs written for existing terminal devices to use a new display without change.

Moreover, the window manager must provide a level of indirection between the application program and the screen because of the shareable nature of the device. The program must be written in a way that is independent of the physical location of its output on the display monitor. Although the program can build a complex image composed of many parts in well-defined spatial relationships, it cannot in general rely on either the position of that complex image on the display or the relationship of that image to other images on the display.

Finally, the window manager is responsible for interacting with the user (the display operator) about the location, size, and status of windows on the screen. In general, the user is given total control of the screen layout. The user can create, modify, move, and destroy windows on the display under his or her control. Multiple processes can thus be created, controlled, and destroyed from a single device.

Using commands provided by the window manager, the display system user arranges the screen with activities of interest in an ordering that suits the user's needs. The window management system supplies the user with a command or menu interface for this purpose.

While the user creates a work area by arranging activities on the screen, each of the images produced by the activities can itself be composed of several parts. That is, from an implementation point of view, it may be convenient for the application to construct its image out of sections that are created by different procedures, modules, or processes. How the image is produced is completely up to the application program, and the fact that it is composed of several pieces may or may not be visible to the user at the screen. For example, in the Smalltalk system [Ingalls 78] the screen is divided into a number of panes, each containing a different class of information, such as Smalltalk code, menus, lists, and so on. Each of the panes has a title and a

surrounding border to distinguish it. However, one might imagine a document containing graphics and text that is maintained by the application as separate graphics and text sections. This fact is hidden from the viewer, as the sectioning merely simplifies implementation.

Many existing window management systems therefore allow the construction of images from several parts. In most cases, this image construction is based on a **decomposition** mechanism. The application program is given a fixed-sized resource object, typically a segment of physical memory (a bitmap), in which to build its image. This object represents a potentially viewable rectangular region and in some cases may actually be a section of the frame buffer from which the display is refreshed. The system then provides primitives for subdividing the object; the program can create any number of rectangular subregions, and these subregions can be passed to other procedures or processes to be written. A subregion is just a bounded rectangular subset of the original region object. Conceptually, all regions or subregions have identical type, in that they are fixed-sized rectangular segments that can be written and/or further subdivided. The window management system ensures that no data is written outside of the boundaries of a specified region object.

The decomposition mechanism allows the application programmer to build a library of useful procedures, each capable of creating output within a region object passed to it as a parameter. In order to perform their task, these procedures can choose to further decompose the problem by subdividing the initial region objects and passing these to other local or global routines, and so on.

One complexity in this scheme arises from the finite single-resource nature of the original region object. That is, the region is represented by a single segment of physical memory. The division of a region into subregions creates objects that indirectly address sections of this single memory segment. Applications often need to overlap regions temporarily, or to create several overlapping regions to be written by independent activities. For example, an application may temporarily obscure part of a region to present the user with a menu, scroll bar, or some new information. Later, the new information must be removed, leaving the previous contents intact. In these cases, the application is often responsible for providing backing storage for any overwritten image that is to be later restored, or for knowing how to regenerate the image. Synchronization might also be required between procedures or processes writing to overlapping regions.

On a more global level, the same storage management problems exist in the sharing of the physical screen. If the frame buffer is divided among several independent processes, and one process' window overlaps another's, then the process controlling the overlapped window must be stopped from writing, even if most of

its window is visible. Or, its output can be clipped to the visible portion, and the process can be asked at a later time to update the entire window if more becomes visible.

An alternative to the decomposition scheme is a mechanism based on the principle of **composition.** In this case, the application begins with a collection of fundamental regions. Each region is represented by a fixed-sized physical resource, as above. However, instead of subdividing a single resource, the application program produces a composite image from the multiple regions. This composite image is specified by describing the spatial relationship of the regions within a two-dimensional coordinate system. That is, the regions are "pasted up" on a plane to form a composite image.

1.2 THE SYSTEM DISPLAY ARCHITECTURE -- A MODEL FOR A COMPOSITION-BASED SYSTEM

The System Display Architecture is based upon a number of fundamental principles, most derived from goals stated in previous sections. First, programs must be written in a manner independent of the location of their output on the physical screen and the relationship of that output to other images on the screen. Second, programs should be independent of the physical characteristics of the display. That is, as much as possible, programs should not count on details such as resolution, screen size, number of bit planes, etc. Finally, the user operating the display system should have total control over the arrangement of applications on the screen.

A critical separation in the architecture is made between the application program and the user of the display. To enforce the distinction between user and program, the architecture provides a distinct set of objects for user control and a different set of objects for application program control. The application, through a procedural interface, manipulates objects provided for the construction of images; the user, through a command interface, manipulates objects provided for the arrangement of screen space. Following sections describe the objects available in each domain.

1.2.1 Program Objects

This section describes the interface that the window management system makes available to the application program for the purpose of composing images. We describe a set of objects, the properties of the objects, and the operations provided on the objects.

1.2.1.1 Virtual Displays -

The fundamental building block for the application program is the **virtual display.** A virtual display is a rectangular display object that has the properties of a display device but is not necessarily implemented on physical display hardware. The virtual display is created as a fixed-sized entity; its main attributes are its height and width. A process can create as many virtual displays as it needs, and each can be a different size to suit a different purpose.

A virtual display, once created, can be passed from procedure to procedure or from process to process. The program can output text and/or graphics to the virtual display. This output can be performed even if the virtual display is not visible or is partially occluded on the display screen. In many cases, applications may wish to perform a series of operations on a virtual display before it is made visible, in order to construct an image prior to viewing.

One of the previously stated goals was to isolate from programs the physical nature of the display device. For this purpose, two high-level interfaces to virtual display output are provided: a text interface and a graphics interface. The text interface imposes a text structure on the virtual display. It allows output and manipulation of variable-height text lines, where each line is composed of a number of fields that contain characters. Fields allow for absolute positioning on a line. Primitives are provided for inserting and deleting characters, fields, and lines, as well as for justifying, scrolling, and so on. Characters, fields, and lines can have attributes, such as reverse video and underline. Multiple fixed- and proportionally-spaced fonts are supported. Since fonts are defined by name and size (e.g., Helvetica 12 point) the program need not be concerned with resolution and font representation issues. The text interface is thus device independent.

The graphics interface consists of a rather standard graphics core package. Once again, the use of a high-level graphics package removes from the program concern for issues such as mapping physical distances into pixels. The program executes procedure calls to draw graphics in the virtual display, using a standard coordinate system. Primitives are provided to map from the graphics coordinate system to character positions within the text structure for programs wishing to combine text and graphics.

1.2.1.2 Pasteboards -

The virtual display described above is the basic output entity in the display architecture. In the simplest case, a program creates a single virtual display that is used as a virtual terminal. The entire virtual display is made visible on some

portion of the display screen. However, the application can also build complex images using virtual displays as the building blocks. The mechanism for combining virtual displays into a single image is called a **pasteboard.**

A pasteboard is simply a two-dimensional cartesian coordinate space used to specify the spatial relationship between virtual displays. A pasteboard is an object that can be created, destroyed, and passed between environments. Virtual displays are **pasted** on the pasteboard through a paste operation, specifying the pasteboard coordinates at which the virtual display origin is to be placed. The unpaste operation removes a virtual display from the pasteboard. Any number of virtual displays can be pasted on a pasteboard and, more important, virtual displays can overlap one another. When a virtual display is pasted partially or fully on top of another, it occludes the view of any previously pasted surfaces that it covers. (A pasteboard is actually a three-dimensional space in the sense that pasted virtual displays have a stacking order.)

Pasteboards have no associated resource, except of course a data structure that indicates the current state of pasted virtual displays. The only attribute of a pasteboard is its color, i.e., what one would see when looking at a pasteboard with no virtual displays pasted on it.

The pasteboard mechanism allows the application program to produce a dynamically changing visual environment without concern for the management of underlying storage. A virtual display maintains its storage database independent of its position on a pasteboard. Pasting, unpasting, or moving a virtual display on a pasteboard only changes the data structure describing the pasteboard. The pasteboard data structure is used to produce an image when the pasteboard is made visible.

For example, virtual displays can be constructed containing standard menus, scroll bars, images, etc. When needed, these virtual displays are pasted on a pasteboard, possibly covering some existing area. When removed, that area of the pasteboard returns to its original condition. In addition, there is no need for synchronization between processes writing to separate virtual displays, whether or not they overlap on the same pasteboard.

Finally, note that a single virtual display could be pasted on several pasteboards at a time, and that a pasteboard could be made visible on one or more physical display screens at a time. In addition, a recursive relation between pasteboards can by allowing pasteboards images to pasted on other pasteboards.

1.2.1.3 Windows -

So far, we have described how the program creates an image but not how the image is made visible. Before an area of a pasteboard can be seen, the application program must create a **window** over that area. A window defines a rectangular area of a pasteboard that is potentially viewable on the screen. A program can define one or more windows on a pasteboard, and the windows can overlap. In the normal case, a window will cover the entire image produced on the pasteboard, that is, the area on which virtual displays have been pasted. However, the application could make a smaller area visible, allowing the window to travel around the pasteboard to observe other areas if desired. Or, the application could construct several disjoint images on the pasteboard, switching between them by moving the window.

1.2.2 Screen Objects

The objects described above provide a programming facility for the composition and viewing of images. The virtual display and the pasteboard allow the program to construct composite images, while the window specifies the section of that image that can be viewed by the user. These objects are under the total control of the application program. Program objects allow screen-position-independent output from the program's point of view. It is the human at the display that determines the **physical** location of images on the display monitor. Therefore, there must be a separate set of objects that the human manipulates to arrange the screen.

Where the window determines a viewable area of a pasteboard, a **viewport** provides screen space for a window on the monitor. A viewport is a rectangular area of the display screen in which a window is viewed. The viewport shows a full border-to-border image from a window. It is the viewport that the user moves on the screen in order to relocate an image. Moving the viewport moves the position of the image on the screen, while moving the window changes the image (the part of the pasteboard) that is viewed.

In fact, viewports are actually positioned on an area larger than the physical screen itself, called the **virtual screen.** At any point in time, the physical screen shows a section of the virtual screen. As the physical screen is panned around the virtual screen, different viewports become visible on the monitor or disappear from view. Or, the physical screen can be zoomed back to show the entire virtual screen, scaled to size, or forward to focus on a smaller area.

The user thus directly controls a potentially large area called the virtual screen space. Viewports are positioned on the virtual screen. The physical screen is positioned on a section

of the virtual screen to make underlying viewports visible on the monitor.

1.2.3 The SDA Specification

The System Display Architecture, as described in this document, consists of two services: the Virtual Display Service (VDS) and the Virtual Screen Service (VSS). Both sets of services are meant to be used by system designer for developing systems which take full advantage of the latest state-of-art user interface devices (high resolution displays) and which meet the needs of the future user -- the non-programming professional.

The Virtual Display Service is used to design the application program interfaces, such as,

- Virtual Terminal Emulators (e.g., VT1ØØ, VT125, Tektronix 141Ø),
- Graphics Program Interface (e.g., XTIG, SIGgraph CORE standard, GKS)
- Interactive Text/Graphics System (e.g., Office Automation, Typesetting)

The Virtual Screen Service and the Virtual Display Service are used by the human interface designer to develop user interfaces.

1.3 REFERENCES

[Foley and vam Dam 82]
James D. Foley and Andries van Dam, Fundamentals of Interactive Computer Graphics, Addison-Wesley Publishing Company, 1982.

[Ingalls 78]
Daniel H. H. Ingalls, The Smalltalk-76 Programming System Design and Implementation, Proceedings of the 5th ACM Symposium on Principles of Programming Languages, January 1978.

[Ingalls 81]
Daniel H. H. Ingalls, The Smalltalk Graphics Kernel, Byte, 6(8), August 1981.

[Kay and Goldberg 77]
Alan Kay and Adele Goldberg, Personal Dynamic Media,

Computer, 10(3), March 1977.

[Lantz and Rashid 79]
Keith A. Lantz and Richard F. Rashid, Virtual Terminal Management in a Multiple Process Environment, Proceedings of the Seventh Symposium on Operating Systems Principles, December 1979.

[McCrossin et al 78]
J.M. McCrossin, R.P. O'Hara, and L.R. Koster, A Time-Sharing Display Terminal Session Manager, IBM Systems Journal, 17(3), 1978.

[Meyrowitz and Moser 81]
Norman Meyrowitz and Margaret Moser, BRUWIN: An Adaptable Design Strategy for Window Manager/Virtual Terminal Systems, Proceedings of the Eighth Symposium on Operating Systems Principles, December 1981.

[Newman and Sproull 79]
W.M. Newman and R.F. Sproull, Principles of Interactive Computer Graphics, McGraw-Hill, 1979.

[Sproull 79]
Robert F. Sproull, Raster Graphics for Interactive Programming Environments, Xerox Palo Alto Research Center, CSL-79-6, June 1979.

[Teitelman 77]
Warren Teitelman, A Display-Oriented Programmer's Assistant, Report CSL-77-3, Xerox Palo Alto Research Center, March 1977.

CHAPTER 2

DISPLAY VOCABULARY

2.1 DISPLAY

For the purposes of this document, a physical device consisting of a high-resolution, raster-scan monitor, keyboard, pointing device, control processor and microcode. A memory in the display specifies the intensity or color for each pixel on the monitor. Each pixel is individually addressable from the host.

2.2 PIXEL

A single picture element or addressable point on a display. Each pixel has a value, represented by one or more bits, that describes its state (i.e., the intensity or color of that point).

2.3 RASTER

A rectangular array of pixels specified by an origin in a Cartesian coordinate system and an extent (its height and width). The sides of a raster are parallel to the X and Y axes of the coordinate system.

2.4 RECTANGLE

Same as RASTER.

2.5 FRAME BUFFER

The memory used to store the current value of each pixel from which the physical display is refreshed.

2.6 BITBLT

BIT BLock Transfer (pronounced "bit blit"). The transfer of a bit string or block from one location to another.

2.7 RASTEROP

A raster operation. The copying (or bitblt) from one raster to another. The destination receiving a logical function of the source and destination rasters (e.g., XOR or OR).

2.8 VIRTUAL

Refers to an object that supports the interface or abstraction of a physical device but is not that physical device.

2.9 VIRTUAL TERMINAL

An object supporting the interface of a particular terminal. For example, a virtual VT100 terminal supports the VT100 programming interface and creates the illusion of a VT100 to the operator and to the application program, but is not implemented on a physical VT100 terminal.

2.10 VIRTUAL DISPLAY

An object created by the Virtual Display Service of the System Display Architecture that has the appearance of a raster-scan display. A virtual display is a rectangular coordinate system to which text and graphics output is done and from which input can be received. All virtual display operations are performed through use of the Virtual Display Service procedural Interface.

2.11 PASTEBOARD

A rectangular coordinate system on which application programs arrange virtual displays and windows for viewing.

2.12 WINDOW

A rectangular area defined on a pasteboard that can potentially be viewed on the display monitor. The window bounds the area that can be viewed and provides a unique ID for that area. Moving the window across the pasteboard potentially changes the picture that would be seen.

2.13 VIRTUAL SCREEN

A rectangular memory space in which viewports lie. The physical screen is fully contained in the virtual screen, and therefore always shows part of the virtual screen. The physical screen can be moved within the virtual screen, providing a panning over the viewports.

2.14 VIEWPORT

An object that provides virtual screen space for a window. A viewport is a rectangle located on the virtual screen that maps an entire window into its display space. Moving a viewport changes the location of that view on the virtual screen, but not the view itself. Many viewports can be visible on a display, and viewports can overlap.

2.15 OCCLUDING VIEWPORT

A viewport that covers or obstructs part or all of another viewport.

2.16 CHARACTER SET

An ordered sequence of symbols. The symbols area commonly refered to as "characters". Each symbol has a name as defined in the Coded Charater Set Register of ISO 2375.

2.17 CHARACTER

Same as a character code.

2.18 CHARACTER CODE

A 8 or 16-bit index into a character set.

2.19 CHARACTER TYPEFACE

A particular stylization of the symbols that make up a character set. Common typeface designations are Gothic, Helvatica, etc. Character typeface does not include character size or rendition.

2.20 CHARACTER RENDITION

Modifications to the appearance of a character other than its typeface and size. These modifications include weight (e.g.FAINT, NORMAL, BOLD), blink mode (OFF,SLOW,FAST), writing mode (NORMAL,REVERSE), style (ITALICS or NORMAL), underlining(ON,OFF), crossout(ON,OFF), proportional spacing(ON,OFF), etc.

2.21 CHARACTER FONT

A specific combination of character typeface, size, and rendition. This is equivalent to printer's "type".

2.22 CLIPPING RECTANGLE

A rectangle whose intersection with another rectangle constrains operations on pixels in the intersected rectangle to the non-overlapping part. An operation on a rectangle may be constrained by several clipping rectangles.

2.23 CURSOR

A small raster displayed on the screen to indicate a point of interest.

2.24 OUTPUT CURSOR

A cursor that indicates the position at which the next character output will be displayed.

2.25 POINTER

The cursor that is controlled by the pointing device.

2.26 POINTING DEVICE

A physical device used to position the pointer to a location on the physical screen. The pointing device and associated pointer are under control of the display operator.

2.27 MOUSE

A pointing device consisting of a small plastic box with several buttons, generally resting on a sphere that is x-y encoded.

2.28 TOUCH PAD

A pointing device consisting of a rectangular area capable of sensing the x-y position at which it is touched with a finger or object.

CHAPTER 3

LAYERING

The System Display Architecture is part of a layered system model, each layer provides a different level of service and manipulates objects at a different level of abstraction, as shown below in Figure 1. Bi-directional communications exist between each of the layers in the system architecture. The System Display Architecture encompasses the layers identified as Virtual Display Service and Virtual Screen Service. It defines the interface between the layers above it in the following diagram.

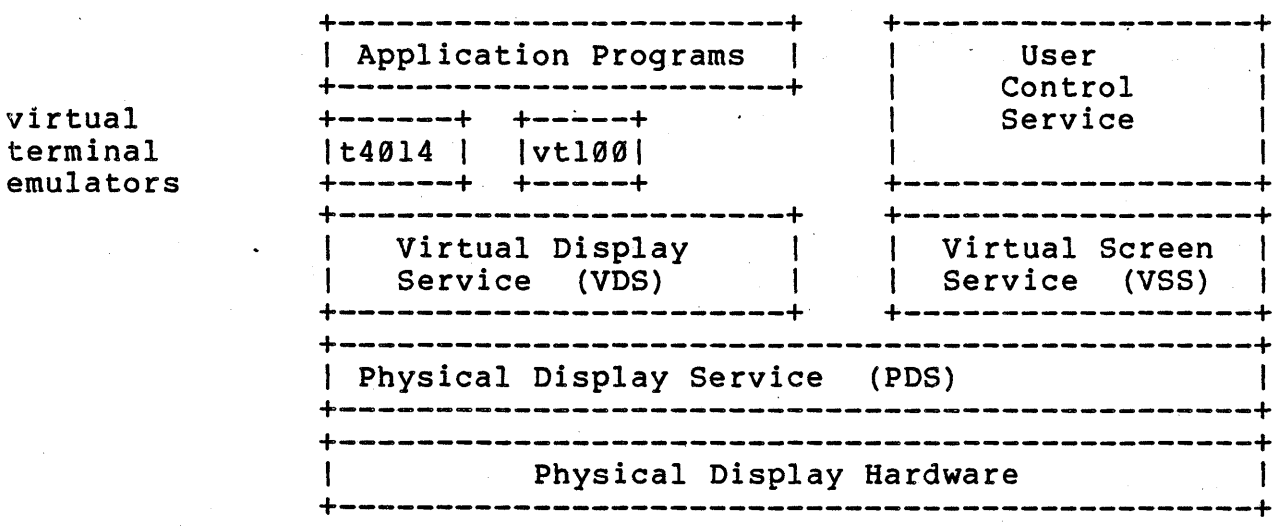


Figure 1. System Architecture Layers

At the highest layer in Figure 1 are the user application programs. Applications have two options in manipulating the display: virtual terminal emulators (VTEMS) or Virtual Display Service. VTEMS are packages that simulate existing graphics or ASCII terminals, such as the Tektronix 4014 or the DEC VT100 family. Existing programs that require the characteristics of such terminals can continue to operate without change; programs transmit ASCII characters, escape sequences, and graphics commands as though to an actual device. The user will see a viewport on the screen whose size and characteristics are those of the simulated terminal. Although VTEMS exactly match the

programming interface of the simulated device, it may not be possible to exactly duplicate the visual properties of the device on the display screen.

The next two layers, Virtual Display Service and Virtual Screen Service, form the basis of the System Display Architecture. These layers manage the objects that allow the multiplexing of the physical screen and input devices among many application processes. The two sets of services provide a separation between the management of programmed input and output objects and the management of screen space objects. Moreover, programs manipulate virtual display objects whether or not the objects are visible on the display screen. A single process in the system allows the user to control the appearance of the display screen and the layout on the screen of views of various virtual displays.

Thus, the Virtual Display Service (VDS) supports creation and manipulation of the objects available for input and output. The purpose of VDS is to supply virtual display objects that isolate the program from the current state of the physical display. Applications can create, write to, and read from virtual display devices, programming each as if it were a single autonomous device. Newer applications may use the VDS interface directly, while VTEMS will use VDS on behalf of existing applications that communicate via existing terminal protocols, such as ASCII, REGIS, etc.

While VDS is responsible for the display programming interface, the Virtual Screen Service (VSS) is responsible for manipulation of objects on the display screen. That is, VSS manages the relationship between virtual displays and their appearance as viewports on the physical screen. In general a single privileged process is responsible for responding to user's screen management requests and calling VSS procedures to manage the allocation of physical screen space and input devices. This process will be refered to as the User Control Service (UCS). The UCS performs the functions of a traditional screen manager.

Following the Virtual Display and Virtual Screen Service in the hierarchy is the Physical Display Service (PDS). Physical display service is provided by what is traditionally called a device driver. The PDS layer is responsible for processing physical display requests from the higher levels, for building physical display command lists, and transmitting commands to the physical display in the form in which they are required. The PDS layer also manages I/O space and mapping registers, and receives and processes interrupts. The PDS layer isolates higher layers from the characteristics of the physical device and interconnection scheme.

The final layer in this structure is the display hardware that includes the physical display monitor, the keyboard and pointing device, and any controlling processor and microcode.

CHAPTER 4

VIRTUAL DISPLAY AND VIRTUAL SCREEN OBJECTS

4.1 INTRODUCTION

The application interface to the System Display Architecture is provided by the Virtual Display and Virtual Screen Service. The division of a single layer into two service packages implies a conceptual division between application-visible objects and operator-visible objects. That is, applications perform input and output operations to virtual devices, without regard to the state of the physical screen or the mapping of their virtual devices to the screen. A separate set of services, generally under control of a single process commanded by the display operator, allows for manipulation of visible screen objects (viewports). The followings sections define the objects supported by each of the services.

4.2 VIRTUAL DISPLAY OBJECTS

The Virtual Display Service forms the programming interface between the application and the display. The objective of VDS is to isolate the program from the details of screen space management. The application performs input and output operations on logical objects, and constructs views that can be made visible on the display screen by the display operator. VDS maintains a database describing the current state of the output objects for each program.

4.2.1 Virtual Displays

The virtual display is the basic object that VDS provides for output. Each virtual display is a rectangular coordinate space to which text and graphics can be written. An application can create and manipulate several virtual displays at a time. Output is automatically constrained to the fixed boundaries of the virtual display.

As an example, the virtual terminal emulator responsible for VT100 emulation will create a single virtual display capable of holding standard 80 column by 24 line VT100 text. Of course, this virtual display will be used only for text, as the VT100 does not support graphics. Other applications can create virtual displays of different sizes and characteristics, to which they may output text, graphics, or both.

A virtual display is represented by a database that contains data structures describing its state. Separate data structures maintain the virtual display's text and graphics context. Therefore, text and graphics operations do not interfere with each other. When a virtual display is shown on the screen, the text and graphics are superimposed to form the screen image (i.e., logically ORed for a one-plane display).

When a virtual display is created, default values are specified for various attributes of the virtual display. For example, the virtual display's background color or intensity could be specified, also, a color for writing text or graphics, a default text font and so on. The defaults can be subsequently modified. In the simplest case, text written to the virtual display assumes the defaults set for the virtual display, however these attributes can be changed on a local basis for a particular line, field, or character.

A program can create and manipulate several virtual displays at a time. The figure below shows three virtual displays, one used for text output, one used for graphics output, and one used for both.

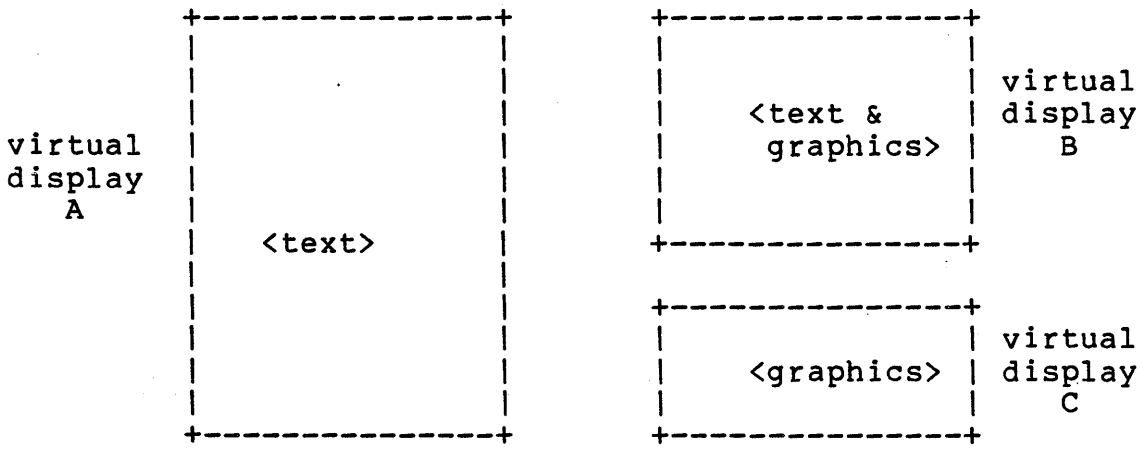


Figure 2. Example Virtual Displays

4.2.2 Pasteboards

The pasteboard is an application mechanism for forming virtual displays into simple or complex arrangements for viewing. It is a rectangular area on which virtual displays can be placed. In the simple case, a pasteboard might contain a single virtual display that emulates a VT100. However, to form a more complex picture, such as a two-column document page including graphics, several virtual displays can be "pasted" together on the pasteboard to form the composite document. The pasteboard simply provides an area in which the position of each virtual display can be specified. A pasteboard has a finite size; a virtual display can be positioned anywhere within the pasteboard area, but no part of a virtual display can be placed outside the area. The figure below shows a pasteboard with virtual displays placed on it. The dotted lines in the figure indicate virtual display boundaries; the dashed lines, the border of the pasteboard. The application can request that virtual display boundaries be either visible or invisible. Pasteboards borders can never been displayed.

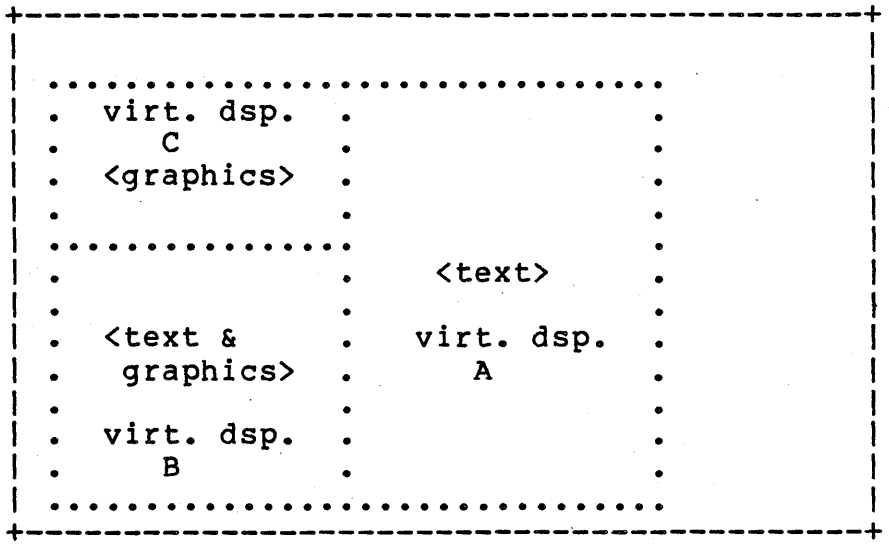


Figure 3. Example Pasteboard

In this example, an application has chosen to implement the three sections of a document as separate virtual displays. Although this pasteboard could have been constructed with a single virtual display to which both text and graphics are written, the use of several virtual displays simplifies the application's management of both text and graphics. Note that each virtual display is an independent device. The pasteboard merely provides a frame of reference in which they can be located.

In more complex relationships, virtual displays can overlap on the pasteboard. A virtual display that overlaps another virtual display fully or partially occludes its view. The stacking order

is determined by the order in which virtual displays are pasted. Virtual displays can be dynamically placed on, moved across , and removed from the pasteboard.

A user can create multiple pasteboards, and a single virtual display can be pasted on several pasteboards at a time.

In addition to pasting virtual displays on pasteboards, pasteboards themselves can be pasted onto other pasteboards. This is an especially powerful facility which allows segmentation of functions. In this case, the pasted pasteboard acts like a virtual display image -- a flat 2-dimensional structure. Virtual displays which may reside on the pasted pasteboard have no effect on the internal structure of the bottom pasteboard. The pasted pasteboard cannot be larger than the bottom pasteboard. In addition, a pastebaord can not be pasted onto itself, nor can it be pasted on to any pasteboard that would result in a similar self-recursive relationship.

4.2.3 Windows

A window defines a rectangular area on a pasteboard that can be presented to the physical display screen. A particular window may or may not be visible on the screen at any particular time. Many windows can be defined for a single pasteboard, and windows can overlap. A window can also overlap several virtual displays. Windows can be created, moved, enlarged, reduced, and destroyed. No part of a window can be placed outside the area of the pasteboard. Note that output does not take place to a window and is not constrained to any particular window. Output takes place to virtual displays. If a window exactly overlaps a virtual display on the pasteboard, output to that virtual display will appear to be clipped to the window.

In the simple case of a VT1ØØ virtual terminal, the pasteboard will contain a single virtual display with a single overlapping window. In a more complex application, a small window could be moved around within the pasteboard to pan over a large area composed of several virtual displays.

4.3 VIRTUAL SCREEN OBJECTS

The Virtual Screen Service (VSS) provides primitives to manage the physical display space. It controls the creation and arrangement of viewports on the screen. It is assumed that virtual screen service is under the control of a single privileged application (UCS). This application responds to operator commands to allocate and arrange objects on the screen. VSS thus provides the link between VDS objects and physical screen space. It manages a database describing the state of the

screen and the relationship of viewports to pasteboard windows.

4.3.1 The Virtual Screen And Physical Screen

The virtual screen is a fixed-sized rectangular area, minimally the size of the physical screen. Fully contained within this rectangular area is the physical display screen. The Virtual Screen Service manages the virtual screen and determines what part is visible at any time. Moving the physical screen around in a larger virtual screen space has the visual effect of panning a camera over an area. The size of the virtual screen is implementation dependent.

4.3.2 Viewports

A viewport is a rectangular area that is a mapping of a window onto a specific area on the virtual screen. The viewport is defined by its position on the virtual screen its size, and its associated window. Viewports can be moved within the virtual screen transparently to the application program.

At any point in time, many viewports may be defined on the virtual screen, and viewports can overlap. A viewport that overlaps another obstructs its view, i.e., the first occludes the second. The stacking of viewports is determined by the order in which they are created or moved.

Viewports can be created, moved within virtual screen space, altered in size, connected and disconnected to windows, and destroyed. A viewport maps an entire window into its virtual screen space. Changing the size of the viewport causes scaling of the mapped window image.

A number of events are defined that can be reported back to the application whose virtual display is visible in a viewport. For example, the application may wish to know when the virtual display (in the viewport) becomes occluded, when the pointer enters the virtual display, or when the pointer leaves the virtual display. This information is fed back from VSS through VDS, that is, from the viewport database to the virtual display database.

4.3.3 Example

Figure 4 shows an example of the use of the objects just described. Two pasteboards are in use, named pasteboard 1 and pasteboard 2. Pasteboard 1 has the single virtual display A defined. A single window, P, contains part of virtual display A.

Pasteboard 2 has three virtual displays named B, C and D. Window Q coincides with virtual display D. These are the basic objects related to the pasteboards and their manipulation by the program.

Below the pasteboards are the objects related to the visual interface. The virtual screen defines a rectangle in which viewports can lie. The physical screen is entirely contained within the virtual screen, along with viewports X and Y. The arrowheads indicate that viewport X is connected to window Q, and viewport Y is connected to window P. Note that viewport X is located partially outside of the physical screen rectangle, and thus virtual display D will only be partially visible on the display screen. Viewport X will show a picture of that part of virtual display A inside of it, along with the values of the other enclosed pixels that are part of pasteboard 1's master virtual display.

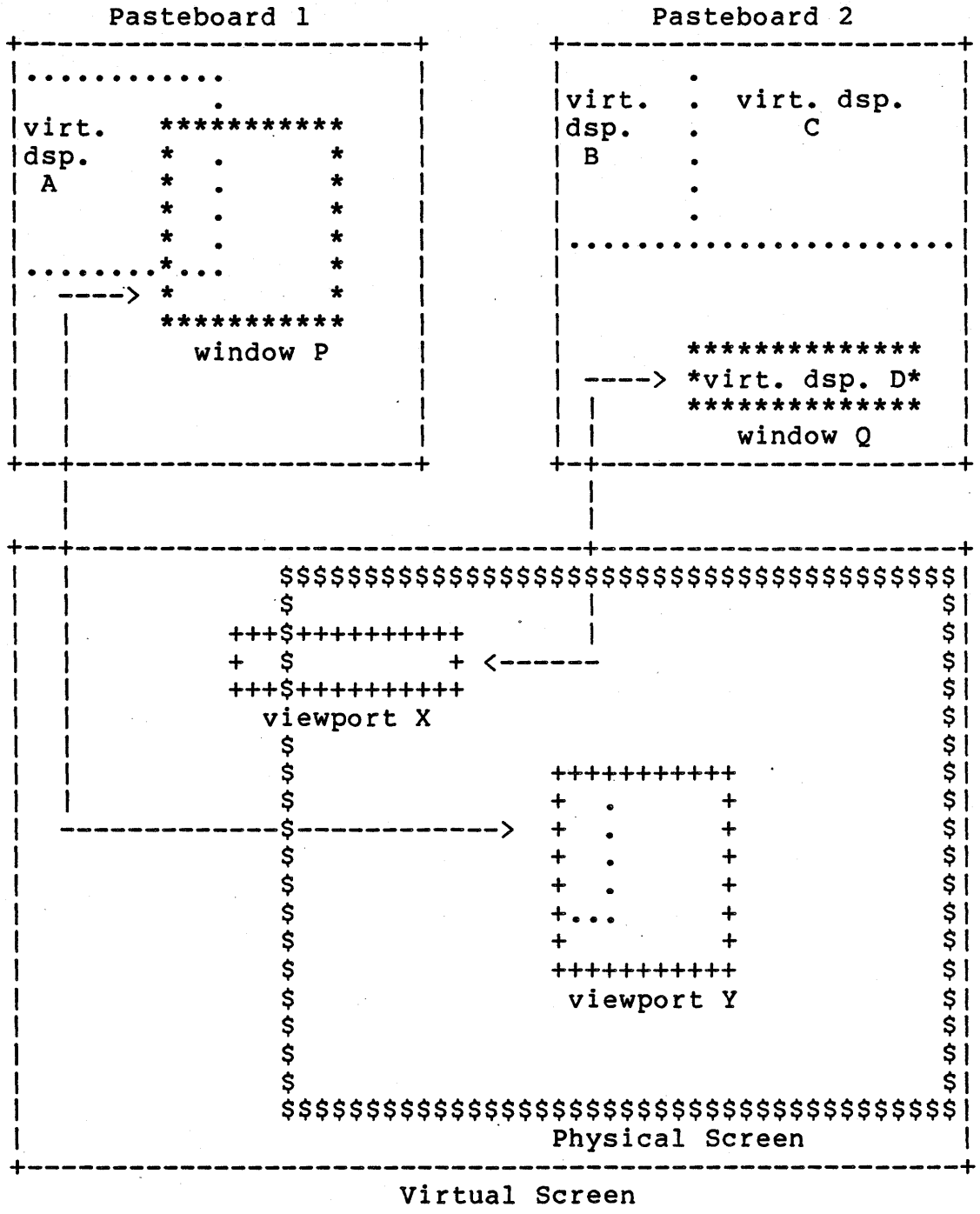


Figure 4. Example window and viewport mapping.

4.4 VISIBLE EFFECTS OF WINDOW AND VIEWPORT OPERATIONS

A number of operations can be performed on windows and viewports, each operation having a different effect on the user image. In this section we list the various visual operations that users may wish to perform, and how they can be accomplished. All are described in terms of the image, that is, the current picture seen by the display operator inside the borders of a viewport.

1. Move Image. The operator wishes to move an image from one location on the physical screen to another location on the physical screen. This operation is performed by simply moving the viewport.

2. Magnify or Reduce Image. The operator wishes to enlarge or shrink the size of the picture. Since a viewport is a scaled border-to-border representation of the window, enlarging the viewport will cause the window data to be scaled and hence magnified. Shrinking the viewport size causes the image to be reduced.

3. Clip Image. The operator wishes to look at only a portion of the current image, for example, at only 12 lines of a 24-line text image. This operation is accomplished by reducing the size of both the window and viewport proportionately.

4. Zoom Image. The operator wishes to zoom in on a smaller area of the image, causing the smaller image to fill the current viewport and be magnified. This operation is accomplished by reducing the size of the window while keeping the viewport the same size.

5. Pan Image. The operator wishes to move the image around in an area larger than that visible in the viewport. This operation is performed by moving the window within the pasteboard.

Architecturally, all of these operations are possible with the two mechanisms defined: windows and viewports. Some implementations may not be able to accomplish all functions, for example, scaling may be difficult without special hardware assistance. Also, some of the operations will generally require interaction between both the application program and the screen manager. For example, the clip image operation requires that both the viewport and window be modified "simultaneously" to give the desired effect.

4.5 BACKGROUND AND WRITING COLORS

All objects in the architecture have both a background color and a foreground color. Foreground color is sometimes referred to as the writing color. Some objects may be primarily background such as virtuals display and pasteboards - only their borders, if they have any, are written in the foreground color. Other object may be primary foreground, e.g. text. The graphics operations of flood and fill are done with background colors.

4.5.1 Color Specification

There are a number of common methods by which color can be specified, such as RGB (Color monitor gamut), NTSC (television color standard). These are hardware related standards that have little resemblance to the physiological and psychological aspects of color preception. The SDA specifies color via the HLS specification. This refers to Hue, Luminence and Saturation; these attributes of color representation have a closer relation to the manner in which color is perceived by humans.

This system specifies color as three real numbers:

Hue (H) The hue of the color expressed as an angle on the color wheel.

Luminence (L) The relative brightness of the color expressed as a percentage of full brightness.

Saturation (S) The relative saturation expressed as a percentage of the fully saturated hue.

There are a minimum of 8 background colors which can be alternately specified by name: white, black, blue, red, green, magenta, cyan, yellow. There are a minimum of two foreground colors: black and white.

The HLS specification for these colors are:

Color	H	L	S	R	G	B
White		100%		1	1	1
Blue	0	50%	100%	0	0	1
Magenta	60	50%	100%	1	0	1
Red	120	50%	100%	1	0	0
Yellow	180	50%	100%	1	1	0
Green	240	50%	100%	0	1	0
Cyan	300	50%	100%	0	1	1
Black		0%		0	0	0

4.5.2 CLEAR Specification

There is an additional "color" which isn't really a color since it specifies the absences of color. That color specification is CLEAR. When CLEAR is specified as the color of an object, the object does not occlude any underlying objects. For example, if a virtual display has a background color of CLEAR and is paste to a pasteboard which has color RED. Then the RED shows though the virtual display at all places where there is nothing written on the virtual display. CLEAR is an OPTION, except as a text background color where CLEAR is required.

4.5.3 Grey Scale Representation

Those system that cannot represent color, but do have intensity capabilities, will map color to grey scale following the conventions established by the photographic and broadcast industry.

Gray Scale	Color
0.0	Black (no color, dark)
|	Blue
|	Red
|	Magenta
|	Green
|	Cyan
V	Yellow
1.0	White

4.5.4 Halftones

Certain hardware implementations may have less than the minimum number of colors or intensities. Indeed, the most common (i.e., inexpensive) system will probably have only one level of intensity. A very effective emulation of grey scale intensities can be created via halftoning. Thus, those system which cannot implement the minimum number of intensities will be allowed to use a set of halftones for background shading.

****
ISSUE: Should we explicitly support "color by index"? In multi-viewport system such support requires separate HW color look-up tables for each viewport. It is doubtful that such features will ever exist in the HW.
****

4.6 INPUT DEVICES

Associated with the display station will be a set of input devices. Input devices can be classified as either text input or graphics input devices. Text input devices are used by the user to input character codes. Examples of text input devices are keyboards, numeric pads, etc.

Graphics input device are used to input positional (N-dimensional) information. Examples of graphics input devices are joystick, graphic tablet, mouse, trackball, etc. Graphics devices can also have operator-controlled state information associated with them. Such state information is usually generated by buttons or keys on the device such as the three buttons on a mouse.

Text and graphics input devices allow the operator to transmit information to an application program. In order to be used an application must create a virtual equivalent and associate it with a pasteboard.

4.6.1 Virtual Input Devices

A virtual input device is the basic object that the VDS uses for input. Just as an application creates a set of virtual displays which allow it to display information to the operator; an application can create virtual input devices in order to obtain information from the operator. Virtual input devices are attached to pasteboards in a manner analogous to pasting virtual displays to a pasteboard.

A virtual input device is represented by a database in the form of a FIFO buffer. Each entry in the buffer consists of a character code for keyboards or positioner co-ordinates and state information. In addition each entry also contains the the position and state information of the pointing device at the time the information was entered by the user. The relevance of this information is discussed below.

4.6.2 Assigning Physical Input Devices

Physical Input devices are assigned to a pasteboard via the VSS. Note that these operation are normally performed by the UCS. Thus, just as the user controls the display of pasteboard information via viewports, the user and the UCS control the assignment of specific physical input devices to pasteboards.

4.7 THE POINTER

The pointer is a small icon (cursor) on the display that tracks the position of the display's pointing device. One of the workstations graphics input devices is designated as the pointing device. The user positions the pointer to indicate an object of interest. The position of the pointer is normally used by the UCS to determine what actions will occur when a key on the keyboard, pointing device, or other graphics devices are pressed or released. The pointer is under the complete physical control of the user (operator), that is, its position cannot be modified by application programs or UCS. However, the icon can be changed by the UCS and the UCS can disassociate itself from the a pointer.

Although the System Display Architecture does not specifically dictate how the pointer is to be used by the UCS or applications, the following was basic in its conception.

The pointing device is solely "owned" by the user; the architecture guarantees that

1. a pointer icon exists on the screen at all times,
2. the user can always move the pointer to any place on the physical screen.

It is highly recommended that the pointer icon and physical pointing device be physically controlled by the hardware display controller. This is to insure that the pointer is response to the operators movements of the pointing device.

As its name implies, the primary purpose of the pointing device is to point to objects on the physical screen. Some of the objects on the screen belong to or are under control of the UCS itself. The rest of the objects on the screen belong to application programs.

4.7.1 Examples Of Pointing

Some of the common uses of the pointer by the UCS or applications are:

1. To activate viewports, pasteboards, and their associated processes,
2. To simplify the sharing of input devices such as the keyboard.

4.7.1.1 Activating Viewports And Pasteboards -

Since overlapping viewports occlude, the pointer can only be inside of one viewport at a time. This viewport is said to be the active viewport. The active viewport is mapped to a specific window which in turn is associated with a specific pasteboard. This pasteboard is then said to be the active pasteboard.

Within this pasteboard, the pointer is either within a virtual display or outside all virtual displays of that pasteboard. Again, as overlapping virtual displays occlude, the pointer can only be inside of one virtual display at a time. If it is within a virtual display, that virtual display is said to be active.

Thus, the UCS or an application program can identify a set of objects that the user is pointing to. Depending on the objects pointed to or the context of the user, the UCS or application program can take appropriate action. For example, if the user moves the pointer to a partially-occluded viewport, the UCS can bring that viewport to the top of the display stack and activate the application associated with it.

An application program can switch its own context as the user moves the pointer from object to object (e.g. virtual display to virtual display). Thus, it's not necessary for the user to explicitly give mode changing commands to the application. This simplifies the user interface because mode changes can become implicit rather than explicit.

4.7.1.2 Sharing Input Devices -

Some physical input devices, such as the keyboard, are shared by a number of pasteboards. The assignment of the physical keyboard or other physical input devices to virtual devices can be changed by the UCS as the pointer moves from pasteboard to pasteboard. Thus, as the operator move the pointer across the physical screen, the UCS activates various pasteboards and switches assignment of the shared input devices such as the keyboard. Practical or physical limitations may require the operator to perform a state change in the pointing device, i.e. press a button on the device.

4.8 VDS TEXT MANAGEMENT OBJECTS

The Virtual Display Service supports output of text to virtual displays. This section describes the basic text objects and the operations available for text output. The basic text objects consist of an hierarchy of entities: virtual display text plane, lines, fields, subfields, and characters. Each object has a set of attributes; the default attributes are determined from higher order objects. Thus, the default character font of a field is the character font of the line in which it is defined. Any defaulted attribute is automatically changed whenever the higher attribute is modified. For example, changing the character size of a line changes the character size of all fields, subfields, and individual characters which where not specifically enumerated.

4.8.1 Lines

For the purposes of text output, a virtual display is divided into a sequence of lines, numbered consecutively 1 through N, where N lines have been defined. A line is defined by a line number, height, baseline position, and a set of attributes. A line spans the entire horizontal extent of the virtual display. Lines cannot overlap, however different lines in a virtual display can have different heights and attributes. Although a line has a fixed height, there is no requirement that all characters within the line be the same size. Characters on a line can be selected from different typefaces and have different heights and widths. The presentation of characters whose size exceeds the height of a line is UNDEFINED.

4.8.2 Fields

Each line consists of one or more fields. A field is defined by its starting and ending position within the line, a set of attributes, and one or more vertically positioned subfields. A fields height is identical to the line in which it is defined. A field is addressed by its starting position.

A field allows the application to force placement of a strings starting at a specific horizontal position within the line. This function would otherwise be difficult with variable-spaced characters.

Placement of a field within a line causes a logical boundary to be defined. The first field in a line, created automatically when a line is created, is starts at position 1 and spans the entire line. Fields are not required to be adjacent, and they must not overlap. Before a new field can be created, any underlying fields must be deleted. If a line is not completely

divided into fields, there will be some space where characters cannot be written.

4.8.3 Subfields

A field is further divided into vertically positioned subfields. A subfield is defined by its vertical position within a field, its height, and a set of attributes. A subfields horizontal extent is equal to the fields extent. Like fields, subfields within a field can not overlap. The first subfield in a field, created automatically when the field is created, starts at position 1 and spans the entire height of the field. Before a new subfield can be created any underlying subfields must be deleted.

Any characters on deleted subfields are erased. If a field is not completely divided into subfields, there will be some space where characters cannot be written.

The character strings within a subfield are formatted according to the text format specification of the subfield. The text string can be left-justified (default), right-justified, centered, or filled. Whenever a character or string of characters is inserted, deleted, or replaced within a subfield, the entire subfield is reformatted according to the format specification of the subfield. Characters output to the subfield are constrained to the subfield horizontal boundaries.

4.8.4 Characters

A character defines a symbol to be output within a subfield. A set of the symbols is refered to as a character set. Each character of a set is identified by a character number or index, called a character code. Character codes are either 8- or 16-bit integer. Displayed characters possess a set of three attributes: typeface, size, and rendition. These attributes are NOT part of the character set, but of the display or imaging process. See the following section on Character Imaging below.

4.8.5 Character Addressing

Within each virtual display, then, there exists a set of lines, fields, subfields, and characters making up the textual state of the virtual display. Each individual character is addressed by a quartet, <line,field,subfield,char>, that specifies its location, characteristics, and attributes. For simple characters sets and traditional cell text, the lines and characters may form a simple rectangular grid. However, in the more general case, lines and

characters within a virtual display may be of different sizes and characteristics.

4.8.6 Text Manipulation

Text management within the virtual display operates largely through a system of defaults. When the virtual display is created, a number of defaults are specified that determine the attributes of text output to the display. These include line height, set, background color, and so on. Changing the default values does not affect existing virtual display text state, but only changes those defaults with respect to future text output.

When the virtual display is created, it is automatically partitioned into a set of contiguous equal-sized lines. Lines are numbered from the top of the virtual display to the bottom. The virtual display always contains an integral number of lines. It is impossible to manipulate lines so that only a slice of the line is in the display. Given that the virtual display height may not be an integral multiple of the line size, some space can exist at the top or bottom of the virtual display to which text cannot be written.

4.8.7 Scrolling

Scrolling is the vertical movement of a contiguous set of text lines The scrolling directions supported are up and down only, that is, there is no horizontal scrolling of virtual display text, since horizontal scrolling of text can not be defined for proportionally spaced text. The direction to scroll is specified at each occurrence of a deletion or insertion. These is no default mode.

Scrolling a set of lines up is accomplished as follows: The first (top) line of the set is deleted. The remaining lines of the set are renumbered by decrementing their respective line numbers. A new line is added at the bottom of the scrolled lines. The screen is updated to reflect the change in the state of the virtual display text. Note that one line is deleted and one line is added; there is no change in the lines outside of the scroll region.

The line that is added is not required to be of the same line height as the deleted line or any other lines of the set. If the added line and the deleted line have different heights, all lines below the scroll set are adjusted so that the display is contiguous. That is, if the new line is smaller than the deleted line, all line below the scroll set are moved up. If the new line is larger, all lines below the scroll set are moved down.

Scrolling down is done in a similar manner. In this case, the last (bottom) line of the set is deleted, the remaining lines renumbered by incrementing their line numbers, and a new line added at the top of the set. If the new line is not the same height as the deleted line, all lines below the scroll set are adjusted as in the scrolling up procedure. Note that the set of lines above the scroll region are never moved - it is always the set below the scroll region which is adjusted.

4.8.8 Character Imaging

Characters are symbols that are obtained by indexing into a 'character set' using 'character codes'. Each symbol is unique as specified by the Coded Charater Set Register of ISO 2375. These character symbols only define the shape of the symbols; they not include character size, or physical attributes, such weight (e.g., BOLD), typeface (e.g., GOTHIC), highlighting or blinking, etc. In other words, an "A" is an "a" is an "A".

When characters are displayed or imaged on the display screen, they are embodied with a set of three attributes, size, typeface, and renditions.

Size is the physical size of the character on the screen, it is measured in printers points (1/72 of an inch). The default size is 12 pts.

Typeface is a particular stylization of the character symbol. Examples of typefaces are Helvetica, Times Roman, Gothic, DEC_VT100, etc.

Character rendition is the collection of ALL other "attributes" associated with the presentation of characters. This include such things as blinking, bold, underlining, italics, etc. All renditions are said to be orthogonal, i.e., they are independent of one another and do not interact with one other. There is a set of default or normal renditions. Hardware constraints may limit the implementation of certain renditions; for these cases there is a defined fallback presentation (which may be to ignore the rendition).

4.8.9 Character Renditions

The following defines characters renditions, their normal mode (default), and fallback presentation.

4.8.9.1 Weight -

Weight describes the thickness of the lines used to generate the character symbol. Three weights are defined: LIGHT, NORMAL, and BOLD. NORMAL is the default. Fallback presentation for LIGHT is NORMAL. BOLD and NORMAL are required at all levels of compliance.

4.8.9.2 Style -

Style has two states: ITALIC and NORMAL. The fallback presentation is NORMAL, i.e., ignore the rendition. Italics is an option.

4.8.9.3 Blink -

Blink refers to the operation of changing the weight of a character from LIGHT to BOLD at a fixed rate. For SLOW blink the rate is slower or equal to twice a second. FAST is defined as a rate faster than twice a second. Note, that blink refers to the character weight and NOT the background. The VT1ØØ reverse video blink does not meet this definition. All levels of compliance require at least one blink rate (SLOW or FAST). The fallback rate for the undefined rate is the implemented rate.

4.8.9.4 Reverse Writing -

When reverse writing is in effect, the use of the background color and writing color are inverted. That is, the background color is used to write the character and the writing color is used for the background. Reverse writing is required

4.8.9.5 Underlining And Cross-out -

Underlining and cross-out refer to the addition of a horizontal stroke either below the character symbol (underlining) or through the center of the character symbol (e.g. virtual display) . Underling is required. Cross-out is an option. The fallback for crossout is no crossout, i.e. ignore the rendition.

4.8.9.6 Proportional Spacing -

Proportional spacing refers to the display procedure that allows each character to be placed in an area proportional to the size of its symbol. That is, only enough space is used as is required by the physical extent of the character symbol. For example, the letter "i" requires much less space than the letter "w". Most current terminals do not implement proportional spacing, but instead use "cell text". With cell text the physical space on the screen is divided into a rectangular array of fixed cells into which character symbols are placed. Proportional spacing is an option. The fallback is cell text.

4.9 VDS GRAPHICS MANAGEMENT

The current VDS graphics management facilities are undefined. It is expected that the propose ANSI VDI (Virtual Device Interface) will be used as the basis for the SDA graphics management facilities.

CHAPTER 5

EXAMPLE USES OF THE SDA IN APPLICATIONS AND USER INTERFACES

5.1 INTRODUCTION

The SDA is a powerful display architecture that can implement very simple to very sophisticated user interfaces. It can also be used by application to generate complex images with a minimum of effort on part of the application and its developer. This chapter will describe a few examples; these are not necessarily describe the best implementation. In some cases descriptions may have simplified in order to keep the explanation manageable. The purpose here is to give the reader an rough idea of how the SDA is intended to be used.

5.2 A MULTIPLE-LOGICAL TERMINAL DISPLAY SYSTEM

[Describe the implementation of the VAXStation Version 1 user interface]

5.3 A COMPOSITE DOCUMENT MODEL SYSTEM

[Describe the Workstation A/D Composite Document Model]

5.4 GKS - GRAPHICS KERNEL SYSTEM

[Describe how the GKS normalized device co-ordinate and workstation objects would be implemented using SDA objects]

PART II

SYSTEM SPECIFICATION

CHAPTER 6

VIRTUAL DISPLAY SERVICES OPERATIONS

This chapter describes the primitives for creating and manipulating Virtual Display Services objects. The primitives are described by a Pascal procedural interface. The interface uses the following Pascal definitions.

```
type

        { define data types }

        BUFFER = packed array [1..max_buf_size] of byte;

        STRING = packed array [1..max_string_length] of char;

        ANGLE = 0..360;             {angle in degrees}

        PERCENT = 0..100;           { 0% - 100% }

        BUF_PTR = ^BUFFER;          { general buffer and pointer to same }

        POINTER = INTEGER;          { general address pointer }

        POINT = record              { define point addressing }
                X: INTEGER;         { X coordinate }
                Y: INTEGER          { Y coordinate }
                end;

        EXTENT = record             { define rectangular size }
                HEIGHT: INTEGER;    { height of rectangle }
                LENGTH: INTEGER     { length of rectangle }
                end;

        RECTANGLE = record          { define a relative rectangle }
                ORIGIN: POINT;      { origin relative to defining structure
                SIZE: EXTENT        { side lengths }
                end;
```

```
ID = record                { general user-visible ID }
        SEQUENCE: INTEGER; { sequence number }
        INDEX: INTEGER  { index into array }
        end;


COLOR_NAME = (BLACK,WHITE,RED,GREEN,BLUE,
                MAGENTA,CYAN,YELLOW,CLEAR,
                HLS,HALFTONE); { standard color names}

COLOR = record                                  {Color specific options}
        Case NAME :  COLOR_NAME of              { HLS }
                HLS:(   HUE:ANGLE;
                        LUMINENCE:PERCENT;
                        SATURATION:PERCENT);
                BLACK:();                       { By common names}
                WHITE:();
                RED:();
                GREEN:();
                BLUE:();
                MAGENTA:();
                CYAN:();
                YELLOW:();
                CLEAR:();
                HALFTONE:(INTEGER)              { Halftones }
        end;


DIRECTION = (UP,DOWN);  { scrolling directions }

TEXT_FMT = (LEFT_JUST,RIGHT_JUST,
                CENTER); { simple text formatting }

TEXT_PTR = record       { text pointer record }
        LINE: INTEGER;  { line number }
        FIELD: INTEGER; { field position }
        SUBFIELD: INTEGER { subfield position}
        CHAR: INTEGER   { character number }
        end;

ON_OFF_SWITCH = (OFF,ON);                       {On/Off switch}

TYPEFACE = (HELVETICA,TIMES_ROMAN,VT100); {Typeface names}

RENDITION = record                      {character renditions}
        STYLE:(NORMAL,ITALIC);
        WEIGHT:(LIGHT,MEDIUM,BOLD);
        BLINK:(NONE,SLOW,FAST);
        WRITING_MODE:(POSITIVE,NEGATIVE);
        UNDERLINE:ON_OFF_SWITCH;
        CROSS_OUT:ON_OFF_SWITCH;
        end;
```

```
        FONT = record              {character set attributes}
                TYPEFACE_NAME: TYPEFACE; {typeface name}
                SIZE: INTEGER;          {Character Size}
                CHAR_RENDITION: RENDITION {Rendition spec}
                end;

        FSTATUS = (SUCCESS,FAILURE,INVALID); { function return status }

        VDB_INDEX = INTEGER;     { virtual display control block index }

        PCB_INDEX = INTEGER;     { pasteboard control block index }

        WCB_INDEX = INTEGER;     { window control block index }

        VCB_INDEX = INTEGER;     { viewport control block index }

        VSB_INDEX = INTEGER;     { virtual screen control block index }

        PVP_INDEX = INTEGER;     { pasted virtual display or pastebaord

        VKB_INDEX = INTEGER;     { virtual keyboard control block index

        VPB_INDEX = INTEGER;     { virtual postioner control block index
{typend}
```

6.1 VIRTUAL DISPLAY OPERATIONS

The following procedures manipulate virtual displays, which are the basic I/O objects provided by VDS.

6.1.1 Create Virtual Display

This operation creates a new virtual display object, specifying the defaults for output to the virtual display. A unique ID is returned by which the virtual display can be referenced.

```
function CREATE_DISPLAY(
        SIZE: EXTENT;
        BACKGROUND: COLOR;
        FOREGROUND: COLOR;
        DEFAULT_FONT: FONT;
        TEXT_LINESIZE: INTEGER;
        TEXT_BASELINE: INTEGER;
        var DISPLAY_ID: VDB_INDEX):
        FSTATUS;
```

where:

SIZE	The height and width of the virtual display.
BACKGROUND	The default background color or intensity for the virtual display.
FOREGROUND	The default writing color or intensity for the virtual display.
DEFAULT_FONT	The default font (typeface, size, renditions).
TEXT_LINESIZE	The default size for text lines in the virtual display. The virtual display is initialized with lines of this size, the number depending on the linesize and the height of the virtual display.
TEXT_BASELINE	The default text baseline within the line, as an offset from the base of the line. The baseline is used for positioning text on the line.

DISPLAY_ID	The returned unique ID for the virtual display.

6.1.2 Get Virtual Display Characteristics

This operation returns the current defaults for a given virtual display.

```
procedure GET_DISPLAY_CHAR(
        DISPLAY_ID: VDB_INDEX;
        var BACKGROUND: COLOR;
        var FOREGROUND: COLOR;
        var DEFAULT_FONT: FONT;
        var TEXT_LINESIZE: INTEGER;
        var TEXT_BASE_LINE: INTEGER);
```

where:

DISPLAY_ID	The unique ID of the virtual display to be examined.
SIZE	The height and width of the virtual display.
BACKGROUND	The default background color or intensity for the virtual display.
FOREGROUND	The default writing color or intensity for the virtual display.
DEFAULT_FONT	The default font (typeface, size, renditions).
TEXT_LINESIZE	The default size for text lines in the virtual display.
TEXT_BASELINE	The default text baseline within the line, as an offset from the base of the line. The baseline is used for positioning text on the line.

6.1.3 Set Virtual Display Characteristics

This operation changes the default values for the virtual display. Changing the defaults does not affect existing virtual display state, but merely changes the defaults with respect to future operations.

```
function SET_DISPLAY_CHAR(
        DISPLAY_ID: VDB_INDEX;
        SIZE: EXTENT;
        BACKGROUND: COLOR;
        FOREGROUND: COLOR;
        DEFAULT_FONT: FONT;
        TEXT_LINESIZE: INTEGER;
        TEXT_BASELINE: INTEGER;
        FSTATUS;
```

where:

DISPLAY_ID	The ID of the virtual display to be modified.
SIZE	The height and width of the virtual display.
BACKGROUND	The default background color or intensity for the virtual display.
FOREGROUND	The default writing color or intensity for the virtual display.
DEFAULT_FONT	The default font (typeface, size, renditions).
TEXT_LINESIZE	The default size for text lines in the virtual display.
TEXT_BASELINE	The default text baseline within the line, as an offset from the base of the line.

6.1.4 Delete Virtual Display

This operation destroys a virtual display object. All state associated with the virtual display is destroyed. The virtual display is removed from any pasteboards to which it had been pasted. Any part of the virtual display that may be visible is erased.

```
procedure DELETE_DISPLAY(
        DISPLAY_ID: VDB_INDEX);
```

where:

DISPLAY_ID	The ID of the virtual display to be destroyed.

6.1.5 Create Virtual Keyboard

This operation creates a new virtual keyboard. A unique ID is returned by which the virtual keyboard can be referenced. Before data can be read from the virtual keyboard, it must be assigned to a pasteboard.

```
function CREATE_VIRTUAL_KEYBOARD(
        var KEYBOARD_ID: VKB_INDEX):
        FSTATUS;
```

where:

KEYBOARD_ID	The returned unique ID for the virtual keyboard.

6.1.6 Delete Virtual Keyboard

This operation destroys a virtual Keyboard. All state associated with the virtual keyboard is destroyed. The keyboard is deassigned from any pasteboards towhich it had been assigned. Any unread data within the keyboard buffers are lost.

```
procedure DELETE_VIRTUAL_KEYBOARD(
        KEYBOARD_ID: VKB_INDEX);
```

where:

KEYBOARD_ID	The ID of the virtual keyboard to be destroyed.

6.1.7 Create Virtual Positioner

This operation creates a new virtual graphics input device. A unique ID is returned by which the virtual display can be referenced. Before data can be read from the virtual positioner it must be assigned to a pasteboard.

```
function CREATE_VIRTUAL_POSITIONER(
        var POSITION_ID: VPB_INDEX):
        FSTATUS;
```

where:

POSITION_ID	The returned unique ID for the graphics positioner

6.1.8 Delete Virtual Positioner

This operation destroys a virtual graphics positioning devices All state associated with the virtual positioner is destroyed. The virtual postioner is deassigned from any pasteboards to which it may have been assigned. Any unread data are lost.

```
procedure DELETE_POSITIONER(
        POSITIONER_ID: VPB_INDEX);
```

where:

POSITIONER_ID	The ID of the virtual positioner to be destroyed.

6.2 PASTEBOARD OPERATIONS

The following operations manipulate pasteboards, which provide the basic application mechanism for specifying spatial relationships between virtual displays. Pasteboards allow for composite pictures to be constructed from multiple virtual displays, and provide a mechanism for the application to specify what parts of those pictures can be made visible.

6.2.1 Create Pasteboard

This operation creates a new pasteboard and returns a unique ID by which it can be specified. A pasteboard has a height and width, and a background color.

```
function CREATE_PASTEBOARD(
        BACKGROUND: COLOR;
        SIZE: EXTENT;
        var PASTEBOARD_ID: PCB_INDEX):
        FSTATUS;
```

where:

BACKGROUND	The default background color or intensity that will be seen if a window covers an empty part of the pasteboard.
SIZE	The height and width of the pasteboard
PASTEBOARD_ID	The unique ID for this pasteboard.

6.2.2 Delete Pasteboard

This operation destroys a pasteboard. Any windows that are attached to this pasteboard are also destroyed, and any currently visible data erased from the screen. Any virtual displays or pasteboards pasted to the pasted border are unpasted. Any virtual input devices are assigned to the pasteboards are deassigned and any physical input devices attached to the pasteboard are detached.

```
procedure DELETE_PASTEBOARD(
        PASTEBOARD_ID: PCB_INDEX);
```

where:

PASTEBOARD_ID	The unique ID of the pasteboard to be deleted.

6.2.3 Paste Virtual Display

This operation causes a virtual display image to be pasted on the specified pasteboard. The virtual display can be positioned anywhere within the pasteboard bounds, but no part of the virtual display may exceed the pasteboard boundaries. The stacking position is specified relative to an existing pasted object (virtual display or other pasteboard); or it can be placed on top of all other pasted objects. A unique ID is returned which identifies this pasting instance. That is, a given virtual display can be pasted to any number of pasteboards, and it can also be pasted to the same pasteboard more than once. The pasting ID uniquely identifies a specific pasting operation.

```
function PASTE_DISPLAY_TO_PASTEBOARD(
        DISPLAY_ID: VDB_INDEX;
        PASTEBOARD_ID: PCB_INDEX;
        ORIGIN: POINT;
        BORDERS: BOOLEAN;
        varPASTING_ID: PVP_INDEX):
        FSTATUS;
```

where:

DISPLAY_ID	The unique ID of the virtual display to be pasted.
PASTEBOARD_ID	The unique ID of the pasteboard to be used.
ORIGIN	The position in the pasteboard coordinate space at which the origin of the virtual display is to be placed.
BORDERS	Boolean indicating whether or not the virtual display border should be made visible.
PASTING_ID	A unique ID which identifies this specific pasting instance.

6.2.4 Paste Pasteboard To Pasteboard

This operation causes a pasteboard image (called the "top pasteboard") to be pasted onto the another pasteboard ("bottom pasteboard"). For this pasting operation the top pasteboard is treated like a virtual display image. That is, it is considered to be a flat 2-dimensional entity; the internal structure (pastings and windows) of the top pasteboard have no effect on the internal structure of the bottom pasteboard, and vica-versa. The only effect is a visual effect within windows of the BOTTOM pasteboard. This pasting instance has NO effect within any windows attached to the TOP pasteboard.

A pasteboard may be pasted to any number of other pasteboards, however, a pasteboard can not be pasted to itself, nor can any pasting operation be specified that results in such a self-recursive relationship.

A unique ID is returned which identifies this pasting instance.

```
function PASTE_PASTEBOARD_TO_PASTEBOARD(
        DISPLAY_ID: PCB_INDEX;
        PASTEBOARD_ID: PCB_INDEX;
        ORIGIN: POINT;
        varPASTING_ID: PVP_INDEX):
        FSTATUS;
```

where:

DISPLAY_ID	The unique ID of the pasteboard to be pasted.
PASTEBOARD_ID	The unique ID of the pasteboard to be used.
ORIGIN	The position in the pasteboard coordinate space at which the origin of the virtual display is to be placed.
PASTING_ID	A unique ID which identifies this specific pasting instance.

6.2.5 Unpaste Display From Pasteboard

This operation removes a pasted object (virtual display or pasteboard) from a pasteboard. The operation removes a single pasting instance. If the same object is pasted to more than one pasteboard or pasted to the same pasteboard more than once, only the specified instance is removed. If the object were visible, it is removed from the screen and any occluded images are made visible.

```
procedure UNPASTE_DISPLAY_FROM_PASTEBOARD(
        PASTING_ID: PVP_INDEX);
```

where:

PASTING_ID	The unique ID of the pasting be removed.

6.2.6 Move Display On Pasteboard

The operation moves a pasted object from a point on the pasteboard to another point. The stacking order of the display is not changed.

```
Procedure MOVE_DISPLAY_ON_PASTEBOARD(
        PASTING_ID:PVP_INDEX,
        ORIGIN: POINT);
```

Where:

PASTING_ID	The unique ID of the pasted object to be used.
ORIGIN	The position in the pasteboard coordinate space at which the origin of the virtual display is to be placed.

6.2.7 Move Display To Top Of Pasteboard

The operation moves a pasted object to the top of the pasteboard stack. It spatial position is not changed.

```
Procedure MOVE_DISPLAY_ON_PASTEBOARD(
        PASTING_ID:PVP_INDEX);
```

Where:

PASTING_ID	The unique ID of the pasted object to be used.

page

6.2.8 Attach Virtual Keyboard To Pasteboard

This operation establishes a connection between a virtual keyboard and a pasteboard. A unique ID is returned identify the connection

```
procedure ATTACH_VIRTUAL_KEYBOARD_TO_PASTEBOARD(
        KEYBOARD_ID: VKB_INDEX;
        PASTEBOARD_ID: VDB_INDEX
        varATTACH_ID: AKB_INDEX)
        FSTATUS;
```

where:

KEYBOARD_ID	The unique ID of the virtual keyboard.
PASTEBOARD_ID	The unique ID of the pasteboard to be associated with the keyboard
ATTACH_ID	The unique ID returned which identifies the attachment.

6.2.9 Detach Keyboard

This operation destroys a connection between a virtual keyboard and the pasteboard to which it is attached. This operation does not destroy any state information about the virtual keyboard

```
procedure DETACH_KEYBOARD(
        ATTACH_ID: AKB_INDEX);
```

where:

ATTACH_ID	The unique ID which identifies the attachment.

6.2.10 Attach Virtual Positioner To Pasteboard

This operation establishes a connection between a virtual positioner and a pasteboard.

```
procedure ATTACH_VIRTUAL_POSITIONER_TO_PASTEBOARD(
        POSITIONER_ID: VGB_INDEX;
        PASTEBOARD_ID: VDB_INDEX)
        varATTACH_ID: APB_INDEX)
        FSTATUS;
```

where:

POSITIONER_ID	The unique ID of the virtual positioner.
PASTEBOARD_ID	The unique ID of the virtual to be associated with the positioner.
ATTACH_ID	The unique ID returned which identifies the attachment.

6.2.11 Detach Positioner

This operation destroys a connection between a virtual positioner and the pasteboard to which it is attached. This operation does not destroy any state information about the virtual positioner

```
procedure DETACH_POSITIONER(
        ATTACH_ID: APB_INDEX)
```

where:

ATTACH_ID	The unique ID returned which identifies the attachment.

6.3 WINDOW OPERATIONS

The following operations manipulate windows, which are application objects that specify a rectangular section of a pasteboard that can be made visible on the display.

6.3.1 Create Window

This operation creates a new window and associates it with an existing pasteboard. A unique ID is returned by which the window can be referenced. The position and size of the window are specified. Any number of windows can be created on a pasteboard. Windows simply define a rectangular area on the pasteboard; windows do not interact with one another, i.e. they do not occlude each other.

```
function CREATE_WINDOW(
        PASTEBOARD_ID: PCB_INDEX;
        ORIGIN: POINT;
        SIZE: EXTENT;
        var WINDOW_ID: WCB_INDEX):
        FSTATUS;
```

where:

PASTEBOARD_ID	The unique ID of the pasteboard on which the window is to be created.
ORIGIN	The origin (upper left corner) of the window within the pasteboard coordinate space.
SIZE	The extent (height and width) of the window rectangle.
WINDOW_ID	The unique ID of the newly created window.

6.3.2 Get Window Characteristics

This operation returns information about a window.

```
procedure GET_WINDOW_CHAR(
        WINDOW_ID: WCB_INDEX;
        var PASTEBOARD_ID: PCB_INDEX;
        var ORIGIN: POINT;
        var SIZE: EXTENT);
```

where:

WINDOW_ID	The unique ID of the window to be examined.
PASTEBOARD_ID	The unique ID of the pasteboard on which the window is defined.
ORIGIN	The origin of the window within the pasteboard coordinate space.
SIZE	The extent (height and width) of the window rectangle.

6.3.3 Set Window Characteristics

This operation modifies the position or size of a window. This operation is used to either move a window on a pasteboard or change its size. If the window's size is changed and the origin is not, the right and bottom edges of the window are moved.

```
function SET_WINDOW_CHAR(
        WINDOW_ID: WCB_INDEX;
        ORIGIN: POINT;
        SIZE: EXTENT):
        FSTATUS;
```

where:

WINDOW_ID	The unique ID of the window to be modified.
PASTEBOARD_ID	The unique ID of the pasteboard on which the window is defined.
ORIGIN	The origin of the window within the pasteboard coordinate space.
SIZE	The extent (height and width) of the window rectangle.

6.3.4 Get Associated Viewport Characteristics

This operation returns to the application information about the viewport that is currently attached to the specified window. That is, given the window ID, the user can request status about the visibility of the window on the display screen.

```
procedure GET_ASSOC_VIEWPORT(
        WINDOW_ID: WCB_INDEX;
        SIZE: EXTENT;
        VISIBILITY: INTEGER);
```

where:

WINDOW_ID	The unique ID of the window under interrogation.
SIZE	The size of the viewport associated with this window.
VISIBILITY	Indication of whether the associated viewport is fully or partially visible on the physical display screen.

6.3.5 Delete Window

This operation deletes a window. Any information visible in an associated viewport is erased.

```
procedure DELETE_WINDOW(
        WINDOW_ID: WCB_INDEX);
```

where:

WINDOW_ID	The unique ID of the window to be deleted.

CHAPTER 7

VIRTUAL SCREEN SERVICES OPERATIONS

This chapter describes the primitives for creating and manipulating Virtual Screen Services objects. In general, these primitives are available only to a single process known as the screen manager or User Interface. The screen manager responds to requests from the display operator to create, modify, move, and destroy viewports on the screen.

7.1 VIEWPORT OPERATIONS

7.1.1 Create Viewport

This operation creates a new viewport within a specified virtual screen.

```
function CREATE_VIEWPORT(
        VS_ID: VSB_INDEX;
        ORIGIN: POINT;
        SIZE: EXTENT;
        VISIBLE: BOOLEAN;
        var VIEWPORT_ID: VCB_INDEX):
        FSTATUS;
```

where:

VS_ID	The ID of the virtual screen in which the viewport will be placed.
ORIGIN	The origin of the viewport rectangle within the virtual screen.
SIZE	The height and width of the viewport rectangle.
TYPE	The type of viewport. Viewport type indicates the characteristics of the visual border surrounding the viewport, including size and fonts for the header and trailer. For example, some viewports might have "handles" on them, some might look like file folders.
VISIBLE	A boolean that specifies whether the viewport should be visible or invisible.
VIEWPORT_ID	Unique ID of the newly created viewport.

7.1.2 Get Viewport Characteristics

This operation returns information about the specified viewport.

```
procedure GET_VIEWPORT_CHAR(
        VIEWPORT_ID: VCB_INDEX;
        ORIGIN: POINT;
        SIZE: EXTENT;
        VISIBLE: BOOLEAN;
        STACKING: INTEGER;
        COVERED: INTEGER);
```

where:

VIEWPORT_ID	The unique ID of the viewport to be examined.
ORIGIN	The origin of the viewport rectangle within the virtual screen.
SIZE	The height and width of the viewport rectangle.
TYPE	The type of viewport, defining the appearance of its border.
VISIBLE	A boolean indicating whether the viewport is visible or invisible.
STACKING	The stacking priority of the viewport.
COVERED	An indication of the current visual state of the viewport, i.e., whether it is partially or fully occluded.

7.1.3 Set Viewport Characteristics

This operation modifies the current characteristics of a viewport.

```
function SET_VIEWPORT_CHAR(
        VIEWPORT_ID: VCB_INDEX;
        ORIGIN: POINT;
        SIZE: EXTENT;
        VISIBLE: BOOLEAN;
        STACKING: INTEGER):
        FSTATUS;
```

where:

VIEWPORT_ID	The unique ID of the viewport to be modified.
ORIGIN	The origin of the viewport rectangle within the virtual screen.
SIZE	The height and width of the viewport rectangle.
TYPE	The type of viewport, defining the appearance of its border.
VISIBLE	A boolean indicating whether the viewport should be visible or invisible.
STACKING	The stacking priority of the viewport.

7.1.4 Attach Viewport To Window

This operation establishes a connection between a viewport and a window, making the information in the window visible on the screen if the viewport is visible. Only one viewport can be attached to a window.

```
function ATTACH_VIEWPORT(
        VIEWPORT_ID: VCB_INDEX;
        WINDOW_ID: WCB_INDEX):
        FSTATUS;
```

where:

VIEWPORT_ID	The unique ID of the viewport to be attached.
WINDOW_ID	The unique ID of the window be attached to the viewport.

7.1.5 Detach Viewport

This operation removes the connection between a viewport and window. Any image inside of the viewport is erased.

```
procedure DETACH_VIEWPORT(
        VIEWPORT_ID: VCB_INDEX);
```

where:

VIEWPORT_ID	Unique ID of the viewport to be detached.

7.1.6 Delete Viewport

This operation deletes a viewport from the virtual screen. If visible on the physical screen, the image is erased and any occluded viewports become visible.

```
procedure DELETE_VIEWPORT(
        VIEWPORT_ID: VCB_INDEX);
```

where:

VIEWPORT_ID	The unique ID of the viewport to be deleted.

7.2 VIRTUAL SCREEN OPERATIONS

7.2.1 Create Virtual Screen

This operation creates a new virtual screen.

```
procedure CREATE_VIRTUAL_SCREEN(
        var VIRT_SCREEN_ID:VSB_INDEX;
        VIRT_SCREEN_COLOR: COLOR):
        FSTATUS;
```

where:

VIRT_SCREEN_ID The ID of the virtual screen in which the viewport will be placed.

VIRT_SCREEN_COLOR Color of the virtual screen.

7.2.2 Delete Virtual Screen

This operation deletes a virtual screen. All viewports attached to the virtual screen are deleted.

```
procedure DELETE_VIRTUAL_SCREEN(
        VIRT_SCREEN_ID:VSB_INDEX):
        FSTATUS;
```

where:

VIRT_SCREEN_ID The unique ID of the virtual screen to be deleted.

7.2.3 Attach Viewport To Virtual Screen

This operation establishes a connection between a viewport and a virtual screen.

```
procedure ATTACH_VIEWPORT_TO_SCREEN(
        VIEWPORT_ID: VPB_INDEX;
        VIRT_SCREEN_ID: VSB_INDEX;
        ORIGIN:POINT):
        FSTATUS;
```

where:

VIEWPORT_ID	The unique ID of the viewport to be attached.
VIRT_SCREEN_ID	The unique ID of the window be attached to the viewport.
ORIGIN	The origin of the viewport rectangle within the virtual screen.

7.2.4 Detach Viewport From Virtual Screen

This operation removes the connection between a viewport and virtual screen. Any image of the viewport is erased from the screen.

```
procedure DETACH_VIEWPORT_FROM_SCREEN(
        VIEWPORT_ID: VPB_INDEX;
        FSTATUS;
```

where:

VIEWPORT_ID	Unique ID of the viewport to be detached.

7.2.5 Move Viewport On Virtual SCreen

The operation moves a viewport from one point on a virtual screen to another point. The stacking order is not changed.

```
Procedure MOVE_VIEWPORT_ON_SCREEN(
        VIEWPORT_ID: VPB_INDEX,
        VIRT_SCREEN_ID:VSB_INDEX,
        ORIGIN: POINT);
```

Where:

VIEWPORT_ID	The unique ID of the viewport to be moved.
VIRT_SCREEN_ID	The unique ID of the virtual screen to be used.
ORIGIN	The position in the virtual screen coordinate space at which the origin of the viewport is to be placed.

7.2.6 Move Viewport To Top Of Virtual Screen

The operation moves a viewport to the top of the virtual screen stack. It spatial position is not changed.

```
Procedure MOVE_VIEWPORT_TO_TOP(
        VIEWPORT_ID: VPB_INDEX,
        VIRT_SCREEN_ID:VSB_INDEX);
```

Where:

VIEWPORT_ID	The unique ID of the viewport to be moved.
VIRT_SCREEN_ID	The unique ID of the virtual screen to be used.

7.3 PHYSICAL SCREEN OPERATIONS

7.3.1 Assign Physical Screen

This operation assigns a specific physical screen on the specified virtual screen.

```
function ASSIGN_PHYSICAL_SCREEN(
        PHYS_SCREEN_SPEC: PSB_SPEC;
        VIRT_SCREEN_ID: VSB_INDEX;
        SIZE: EXTENT;
        ORIGIN: POINT;
        var PHYS_SCREEN_ID: PSB_INDEX):
        FSTATUS;
```

where:

PHYS_SCREEN_SPEC	The system dependent specification of a physical screen.
VIRT_SCREEN_ID	The ID of the virtual screen in which the physical screen will be placed.
SIZE	The size of the physical screen
ORIGIN	The origin of the physical screen within the virtual screen.
PHYS_SCREEN_ID	The ID of the physical screen created.

7.3.2 Get Physical Screen Characteristics

This operation returns information about the specified physical screen.

```
procedure GET_PHYSICAL_SCREEN_CHAR(
        PHYS_SCREEN_ID: PSB_INDEX;
        var ORIGIN: POINT;
        var SIZE: EXTENT);
```

where:

PHYS_SCREEN_ID	The unique ID of the physical screen to be examined.
ORIGIN	The origin of the physical on the virtual screen.
SIZE	The size of the physical screen

7.3.3 Set Physical Screen Characteristics

This operation modifies the current characteristics of a physical screen

```
procedure SET_PHYSICAL_SCREEN_CHAR(
        PHYS_SCREEN_ID: PSB_INDEX;
        ORIGIN: POINT;
        SIZE: EXTENT);
```

where:

PHYS_SCREEN_ID	The unique ID of the physical screen to be modified.
ORIGIN	The origin of the physical screen rectangle within the virtual screen.

SIZE	The height and width of the viewport rectangle.

7.3.4 Move Physical Screen In Virtual Screen

The operation moves a physical screen with in the virtual screen

```
Procedure MOVE_PHYSICAL_SCREEN_ON_VIRTUAL_SCREEN(
        PHYS_SCREEN_ID: PSB_INDEX,
        ORIGIN: POINT);
```

where:

PHYS_SCREEN_ID	The unique ID of the physical screen to be moved.
ORIGIN	The origin of the physical screen rectangle within the virtual screen.

7.3.5 Deassign Physical Screen

This operation detaches a physical screen from it assign virtual screen.

```
procedure DELETE_PHYSICAL_SCREEN(
        PHYS_SCREEN_ID:PSB_INDEX):
        FSTATUS;
```

where:

PHYS_SCREEN_ID	The unique ID of the physical screen to be deleted.

7.3.6 Assign Physical Keyboard To Pasteboard

This operation assigns a physical keyboard to a pasteboard. If a virtual keyboard is attached to the pasteboard, data from the physical keyboard are placed in the virtual keyboards buffer.

```
procedure ASSIGN_PHYS_KEYBOARD_TO_PASTEBOARD(
        PASTEBOARD_ID: PCB_INDEX;
        PHYS_KEYBOARD_DEV: DEVICE_SPEC);
        FSTATUS;
```

where:

PASTEBOARD_ID The unique ID of the pasteboard

PHYS_KEYBOARD_ID The device specification of the physical keyboard.

7.3.7 Assign Physical Positioner To Pasteboard

This operation assigns a physical graphics input device to a pasteboard.

```
procedure ATTACH_PHYS_POSITIONER_TO_PASTEBOARD(
        PASTEBOARD_ID: PCB_INDEX;
        PHYS_POSITIONER_DEV: DEVICE_SPEC);
        FSTATUS;
```

where:

PASTEBOARD_ID The unique ID of the pasteboard

PHYS_POSITIONER_DEV The device specification of the physical positioner.

7.4 POINTER OPERATIONS

7.4.1 Create Virtual Pointer

This operation creates a new virtual pointer.

```
function CREATE_VIRTUAL_POINTER(
        var POINTER_ID: PTB_INDEX;
        FSTATUS;
```

where:

POINTER_ID	The ID of the pointer created.

7.4.2 Assign Physical Positioner To Pointer

This operation defines physical graphics input device to be a specified pointer.

```
procedure ASSIGN_POSITIONER_TO_POINTER(
        POSITIONER_ID: PGB_INDEX;
        POINTER_ID: PTB_INDEX;)
```

where:

POINTER_ID	The ID of the pointer.
POSITIONER_ID	The ID of the positioner to be used as the pointer.

7.4.3 Delete Pointer

This operation destroys a virtual pointer.

```
function DELETE_POINTER(
        var POINTER_ID: PTB_INDEX;
        FSTATUS;
```

where:

POINTER_ID	The ID of the pointer to be destroyed

CHAPTER 8

VIRTUAL DISPLAY TEXT MANAGEMENT OPERATIONS

This chapter describes the primitives for text input and output to the virtual display. Text output is controlled by the current virtual display defaults for font, writing and background color, text line size, and baseline offset. Whenever a text string is written, the current virtual display defaults are used. Modification of defaults only affects future text output requests.

General output requests simply place characters within the virtual display. Characters are specified as elements of a defined font. No control characters are interpreted, and ASCII control characters may have printing symbols defined within the font.

Simple formatting is provided within a line or field using the simple Write String primitive. A more complex operation is also available if the user requires explicit control over the placement of each character.

8.1 VIRTUAL DISPLAY INPUT

8.1.1 Virtual Display State Poll

This operation returns the current input and output state of the specified virtual display. The information returned indicates the current text and graphics output positions, as well as the position of the pointer if currently inside of the virtual display.

```
procedure DISPLAY_STATE_POLL(
        DISPLAY_ID: VDB_INDEX;
        var ACTIVE: BOOLEAN;
        var POINTER_GRAPHICS: POINT;
        var POINTER_TEXT: TEXT_PTR;
        var CUR_GRAPHICS: POINT;
        var CUR_TEXT: TEXT_PTR);
```

where:

DISPLAY_ID	The unique ID of the virtual display to be examined.
ACTIVE	Boolean indicating whether or not the virtual display is active, that is, whether the pointer is currently within a visible portion of this virtual display through a screen viewport.
POINTER_GRAPHICS	If ACTIVE is true, then this contains the current position of the pointer in the virtual display coordinate system, otherwise undefined.
POINTER_TEXT	If ACTIVE is true, then this contains the current text character position, if any, of the pointer.
CUR_GRAPHICS	The current output position for graphics operations on the virtual display.
CUR_TEXT	The current output position for text operations on the virtual display.

8.1.2 Insert Line

This operation inserts an empty line in the virtual display, at the specified line number. If the scrolling direction is specified to be "up", then the lines before the inserted line are renumbered and the original line 1 is deleted. If the scrolling direction is specified to be "down", then the lines after the inserted line are renumbered.

```
procedure INSERT_LINES(
        DISPLAY_ID: VDB_INDEX;
        LINE: INTEGER;
        SCROLL_DIR: DIRECTION
        LINE_HEIGHT: INTEGER);
```

where:

DISPLAY_ID	The ID of the virtual display in which to insert the line.
LINE	The number of the line to be inserted
SCROLL_DIR	The direction scroll the text before or after the inserted line.
LINE_HEIGHT	The height of the inserted line.

8.1.3 Delete Line

This operation deletes a line from the virtual display. All lines below the deleted lines are scrolled up.

```
procedure DELETE_LINE(
        DISPLAY_ID: VDB_INDEX
        LINE_NUMBER);
```

where:

DISPLAY_ID	The ID of the virtual display being modified.
LINE_NUMBER	This line number and the current line number determine the set of lines to be deleted.

8.1.4 Insert Field

This operation defines a new field within a line. The field has a specific horizontal starting and ending position within the line. The field number is specified by the position in the line. The line number is specified by the line number in the current text position. The new field must not overlap an existing field.

```
procedure INSERT_FIELD(
        DISPLAY_ID: VDB_INDEX;
        START_X: INTEGER;
        FIELD_SIZE: INTEGER);
```

where:

DISPLAY_ID	The ID of the virtual display .
START_X	The starting X position of the field within the line.
FIELD_SIZE	The physical length of the field.

8.1.5 Delete Field

This operation destroys the current field and its contents. Deletion of a field does not cause any of the remaining fields in the line to move.

```
procedure DELETE_FIELD(
        DISPLAY_ID: VDB_INDEX;
        FIELD_POSITION: INTEGER):
```

where:

DISPLAY_ID	The ID of the virtual display.
FIELD_POSITION	Position of the field to be deleted

8.1.6 Insert Sub-Field

This operation defines a new sub-field within a field. The sub-field has a specific vertical starting and physical height The sub-field number is specified by the position in the field. A sub-field must not overlap an existing sub-field.

```
procedure INSERT_SUBFIELD(
        DISPLAY_ID: VDB_INDEX;
        START_Y: INTEGER;
        HEIGHT: INTEGER;
        FORMAT: TEXT_FMT);
```

where:

DISPLAY_ID	The ID of the virtual display .
START_Y	The starting Y position of the sub-field within the field.
HEIGHT	The physical height of the sub-field
FORMAT	Specification of special formatting instructions. The string can be formatted within a field. Supported formats are CENTER, LEFT JUSTIFY, RIGHT JUSTIFY.

8.1.7 Delete Sub-Field

This operation destroys the current sub-field and its contents. Deletion of a sub-field does not cause any of the remaining sub-fields in the field to move.

```
procedure DELETE_SUBFIELD(
        DISPLAY_ID: VDB_INDEX;
        SUB_FIELD_POS: INTEGER):
```

where:

DISPLAY_ID	The ID of the virtual display.
SUB_FIELD_POSITION	The Position of the sub-field to be deleted.

8.1.8 Insert Text String

This operation outputs a text string to the specified position in the virtual display. The text is output with current character attributes. When inserting at character position k within a sub-field, if character k already exists, the string is inserted before k. Characters at and after position k are moved to the right. If character k does not exist, spaces are inserted as fill characters until k-1 characters do exist; then the inserted string is appended. After the string is inserted, the entire field is reformated according to the field's text format.

```
procedure INSERT_TEXT(
        DISPLAY_ID: VDB_INDEX;
        INSERT_POSITION: TEXT_PTR
        STRING_SIZE: INTEGER;
        STRING: BUFFER);
```

where:

DISPLAY_ID	The unique ID of the virtual display being written.
INSERT_POSTION	The line,field,sub-field and character position.

STRING_SIZE	The number of symbols to be written to the virtual display.
STRING	An array of character codes specifying the symbols to be output.

8.1.9 Write Text String

This operation outputs a text string to the specified position in the virtual display. The text is output with current character attributes. Writing text is a character for character replace operation. After the string is written, the entire field is reformated according to the text format.

```
procedure WRITE_TEXT(
        DISPLAY_ID: VDB_INDEX;
        WRITE_POSITION: TEXT_PTR;
        STRING_SIZE: INTEGER;
        STRING: BUFFER);
```

where:

DISPLAY_ID	The unique ID of the virtual display being written.
WRITE_POSTION	The line,field,sub-field and character position at which the text string is to be written.
STRING_SIZE	The number of symbols to be written to the virtual display.
STRING	An array of character codes specifying the symbols to be output.

8.1.10 Delete Character

This operation deletes the character at the specified text position. After deletion, the entire field is reformated according to the field's text format.

```
procedure DELETE_CHAR(
        DISPLAY_ID: VDB_INDEX;
        DELETE_POSITION: TEXT_PTR):
```

where:

	DISPLAY_ID The ID of the virtual display being manipulated.
DELETE_POSTION	The line,field,sub-field and character position of the character to be deleted.

8.2 VIRTUAL DEVICE INPUT

8.2.1 Read Virtual Keyboard

This operation reads the next character for a virtual keyboard.

```
function READ_VIRTUAL_KEYBOARD(
        VIRT_KEYBOARD_ID: VKB_INDEX;
        var POINTER_POS: POINTER_INDEX;
        var ENTRY_TIME: UNIVERSAL_TIME;
        var CHARACTER: CHARACTER_INDEX;)
        FSTATUS;
```

where:

VIRT_KEYBOARD_ID Id of the virtual keyboard to be read.

POINTER_POS: Position of the pointer at the time the character was entered by the user

ENTRY_TIME: Time at which the character was entered.

CHARACTER: The character code of the entered character.

CHAPTER 9

VIRTUAL DISPLAY GRAPHICS OPERATIONS

[The current VDS graphics management facilities are undefined. It is expected the the propose ANSI VDI (Virtual Device Interface) will be used as the basis for the SDA graphics management facilities.]

APPENDIX A

CONFORMANCE LEVELS

A.1 INTRODUCTION

A.2 CONFORMANCE LEVELS

A.2.1 System Conformance

Any system claiming to implement the SDA conforming to level n:

1. shall implement ALL functions of levels Ø to n,
2. may implement some Level n+1 or higher functions, and
3. may implement any function not defined in any level.

Level n is a strict superset of level n-1. These constraints reduce the variability between systems to manageable proportions as seen by application programs.

A.2.2 Application Program Conformance

Any application software claiming to conform to level n:

1. may use any function in levels Ø to n,
2. shall NOT use any functions in levels n+1 or higher, and
3. shall NOT used any functions not defined in any level

A.3 OPTIONS

For a given conformance level, the architectural specifications combines some additional functions together to form a minimum number of options for that level. The system implementor may choose to include or omit options in a given implementation, or may choose to permit the application developer to use such

option.

For the purpose of conformance:

1. Systems claiming to to implement an option shall implement ALL functions in that option, but need not implement any lower level options unless specifically required by the specification of the option.

2. Applications claiming to support options, shall use any functions in those options, but shall use NO functions in other option. Furthermore, APPLICATIONS THAT SUPPORT OPTIONS SHALL OPERATE IN A SENSIBLE WAY IF THAT OPTION IS NOT PRESENT IN THE SYSTEM.

A.4 UNDEFINED OPERATIONS

Occasionally there are operations whose effect cannot be guaranteed to be the same in all implementations. Such operations shall be identified in BOTH the architectural and system specification as UNDEFINED [1]. Applications conforming to the SDA specification shall NOT use UNDEFINED operations. Every attempt will be made to minimize the number of UNDEFINED operations in the specifications since applications that inadvertently issues an UNDEFINED operation may NOT operate the same on all systems.

[1] See the VAX System Reference manual for this precedent.

APPENDIX B

COMPATIBILITY WITH OTHER ARCHITECTURES

B.1 INTRODUCTION

This appendix compares the SDA architectures with other display architectures currently under development with DIGITAL.

B.1.1 The Terminal Interface Architecture

The following is a unedited comparision of the TIA and SDA written by the TIA architect.

****

ABSTRACT: This paper describes the structure of the System Display Architecture being developed for graphics workstations, and the Terminal Interface Architecture being developed for video, printer and computing terminal devices. It describes the relationship of the components of the two architectures, and indicates how implementations of the architectures would map together to provide a consistent applications interface.

****

1.Ø TIA STRUCTURE

The Terminal Interface Architecture is designed to provide a consistent internal and external interface to terminal devices by "virtualizing" the device concept. This interface is referred to as a Virtual Terminal.

1.1 Virtual Terminals

A physical terminal may support multiple Virtual Terminals simultaneously or in sequence. Each Virtual Terminal has a data store, which presented as one or more Virtual Output Devices. Virtual Output Devices typically look like existing terminal products, such as a VT100, VT125, LA120, etc. Data may be written into the Virtual Output Device by an application, and operations on the data in the store may be performed according to the Service Class associated with the Virtual Output Device. A Virtual Terminal also has one or more Virtual Input Devices, which are typically modelled to appear as existing input devices, such as the VT100 keyboard. Each Virtual Input Device is associated at any point in time with a specific Virtual Output Device in the Virtual Terminal.

1.2 Windows And Viewports

It is a function of the Virtual Terminal to map data in the data store to the physical display surface. Because it may be desirable for performance reasons to store more data in the terminal than is displayed at any one time, Windows are created into the data store to indicate portions of the data which are to be displayed. A Window represents a rectangular area of a Virtual Output Device which may be mapped to the physical display. Each Virtual Output Device may have one or more Windows associated with it.

A Viewport represents a rectangular portion of the physical display into which the windowed data is to be mapped. The boundaries of a Window and its associated Viewport map one to one. Future implementations of this architecture may provide the ability to pan, zoom, scale, rotate, or mirror data by the use of Windows and Viewports.

2.0 SDA STRUCTURE

The Systems Display Architecture is a layered architecture, designed to provide a consistent applications interface to high performance display devices in a closely coupled system. Many of the concepts employed are similar to those used by TIA, although their implementation differs somewhat.

2.1 Virtual Devices

In SDA, applications typically address the display through virtual terminal emulators, or Virtual Displays. These Virtual Displays are identical in concept to TIA's Virtual Output Devices. They simulate the functions of known display devices, such as

VT100, VT125, Tektronix 4014, etc. An application may create an unlimited number of Virtual Displays.

2.2 Pasteboards

SDA introduces the concept of Pasteboards to allow the creation of composite images using multiple Virtual Displays. Multiple Virtual Displays may be positioned within a Pasteboard in some fixed relationship to one another. One example of the use of a Pasteboard would be to represent a page of a document which contained text as well as graphics data. The text could be written into a VT100 Virtual Display, the graphics could be written into a 4014 Virtual Display, and the two images could be juxtaposed in the pasteboard to build the composite page image.

2.3 Windows And Viewports

SDA also employs Windows and Viewports to map portions of the composite image to the display. The fundamental difference between TIA and SDA is that in SDA the Windows define portions of the Pasteboard, not portions of the Virtual Display (which would be the equivalent of TIA's Windows into Virtual Output Devices). Thus operations on Windows by applications have a different result in TIA and SDA, a fact that will be dealt with in a later section of this report.

2.4 Virtual Screen

SDA also introduces the concept of a Virtual Screen. The Virtual Screen may be larger than the actual physical display surface. Thus Viewports (which get mapped into the Virtual Screen) may not always be visible on the Physical Screen. This concept is implied but not explicitly expressed in TIA.

3.0 DIFFERENCES AND MAPPINGS

As indicated previously, the fundamental difference between TIA and SDA lies in the use of Pasteboards and Windows. TIA does not include the Pasteboard concept. In TIA, composite images are created directly on the physical display by the use of Windows and Viewports. There are several reasons why this simpler structure is employed:

1. It is less memory intensive, and thus more likely to be implementable in low cost terminal devices.

2. It provides for buffering at a level (Virtual Output

Device) which is more suitable to the way in which terminals are designed, and provides better performance on low speed communications lines, which is the assumed default for terminal operations.

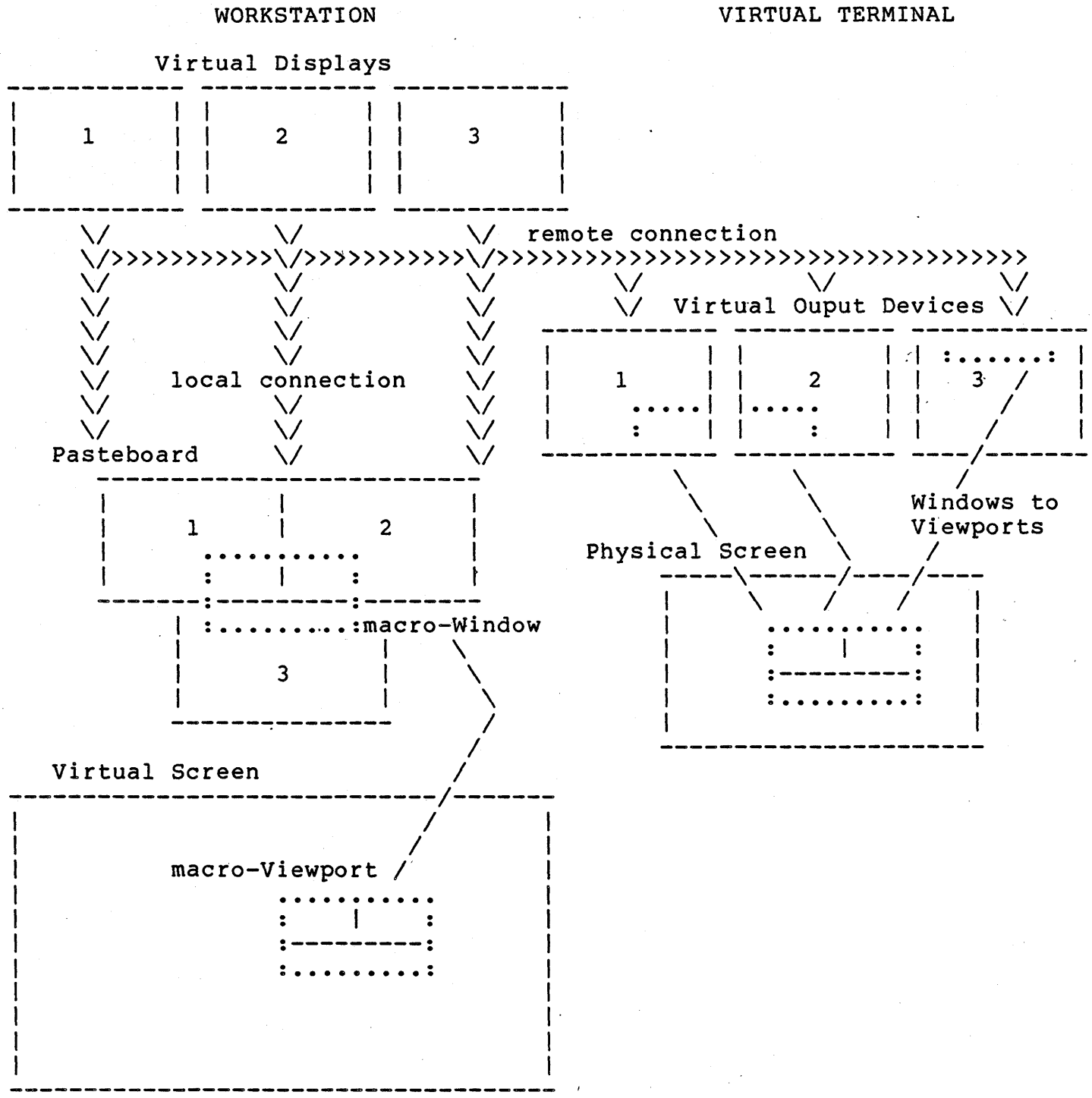
3. It does not require a sophisticated human interfacing process to be resident in the terminal to affect the manipulation of Windows and Viewports for the human operator, which is assumed in the high performance workstation environment.

It is not the intention of TIA that applications should use the Virtual Terminal interface directly, but that some level of translation should be provided by software services (such as run-time libraries or system services). Thus compatibility between TIA and SDA can be maintained at the applications level by providing a mapping of SDA's Pasteboard Windows (hereafter referred to as "macro-Windows") to TIA's Virtual Output Device Windows. This can be accomplished in the following manner:

1. When an application writes data into a Virtual Display, the data is transmitted to a Virtual Terminal and stored in a corresponding Virtual Output Device. Thus a copy of all data to be manipulated by the application is maintained in a store local to the terminal.

2. The application juxtaposes Virtual Displays in the Pasteboard to create the desired composite image. The Virtual Terminal has no knowledge of this positional relationship.

3. A macro-Window is created into the Pasteboard, which includes data from more than one Virtual Display. This macro-Window would be mapped into a macro-Viewport on the Virtual Screen. In the case of a terminal, the macro-Window is "decomposed" into a set of Windows, which are then mapped into Viewports on the physical display, juxtaposed so as to provide the same visual effect as the macro-Viewport. Moving of the macro-Window in the Pasteboard requires corresponding movement of all Windows in the macro-Window set to maintain the desired composite image. (See attached drawing)



SDA Mapping to TIA

Macro-Window/Viewport mapped to multiple Windows and Viewport

B.1.2 The Terminal Software Architecture

[tbs]