VAX LISP/VMS Graphics Programming Guide

Order Number: AA-GH76A-TE

May 1986

This document contains information required by a LISP language programmer to write programs that use the VAX LISP interface to VAXstation graphics.

Operating System and Version:	VAX/VMS Version 4.2
Software Version:	VAX LISP/VMS Version 2.0

digital equipment corporation
maynard, massachusetts

First Printing May 1986

A postage-paid READER'S COMMENTS form is included on the last page of this document. Your comments will assist us in preparing future documentation.

CONTENTS

TABLES

PREFACE

Manual Objectives

The *VAX LISP/VMS Graphics Programming Guide* describes the VAX LISP/VMS graphics system. This graphics system provides an interface to, and is intended for use on, the VAXstation family of workstations.

Intended Audience

This manual is designed for programmers who are already familiar with VAX LISP and who need to use VAX LISP's programming interface to VAXstation graphics. You should also be familiar with the user interface to the VAXstation, as described in the *MicroVMS Workstation User's Guide*.

Some sections of this manual require detailed understanding of the operating system or of VAX LISP's interaction with it. In such sections, you are directed to the appropriate manual(s) for additional information.

Structure of This Document

An outline of the organization and chapter content of this manual follows:

PART I: GUIDE TO GRAPHICS PROGRAMMING

Part I presents the VAX LISP graphics system in tutorial fashion, by subject area. You may want to read through these chapters once, initially, and then refer to them afterward as necessary.

- Chapter 1, "Overview," introduces the graphics system and provides a demonstration of its capabilities.
- Chapter 2, "Virtual Displays, Windows, and Transformations," explains the mechanisms by which graphic information is stored and displayed on the screen.

- Chapter 3, "Graphic Output Operations," describes the various operations that display lines and text, as well as ways of changing the appearance of the output.

- Chapter 4, "Screen Images and Bitmaps," shows how you can read images from the screen into an array, modify the array, and write it back to the screen.

- Chapter 5, "Pointer Operations," explains how to use the pointing device (mouse or tablet) as an input device.

- Chapter 6, "Keyboard Input," describes virtual keyboards, through which keystrokes from the physical keyboard can be captured by a window.

- Chapter 7, "Window Output Streams," shows how you can perform text output to a window through a VAX LISP stream.

PART II: GRAPHICS SYSTEM COMPONENTS

Part II contains definitions of the functions, macros, and data types specific to the VAX LISP graphics system. The descriptions are in alphabetical order. Each description explains the function's or macro's use and shows its format, applicable arguments, and return value.

Associated Documents

The following documents are relevant to using VAX LISP graphics:

- The *VAX LISP/VMS User's Guide* provides general information about using VAX LISP, and serves as a guide to generally-helpful VMS documentation.

- *COMMON LISP: The Language* provides a definition of the COMMON LISP language.

- The *VAX LISP/VMS System Access Programming Guide* describes how a VAX LISP programmer can use the programming interface to the VMS operating system. The chapter entitled "Interrupt Functions" is especially important for users of VAX LISP graphics.

- The *VAX LISP/VMS Editor Programming Guide* explains how to extend the capabilities of the VAX LISP Editor and includes information on workstation-specific Editor functionality.

PREFACE

Conventions Used in This Document

The following conventions are used in this manual:

Convention	**Meaning**
( )	Parentheses used in examples of LISP code indicate the beginning and end of a LISP form. For example: `(SETQ NAME LISP)`
UPPERCASE	DCL commands and qualifiers, and defined LISP functions, macros, variables, and constants are printed in uppercase characters; however, you can enter them in uppercase, lowercase, or a combination of uppercase and lowercase characters.
lowercase italics	Lowercase italics in function and macro descriptions and in text indicate arguments that you supply; however, you can enter them in lowercase, uppercase, or a combination of lowercase and uppercase characters.
...	In LISP examples, a horizontal ellipsis indicates code not pertinent to the example and not shown.
. . .	A vertical ellipsis indicates that all the information that the system would display in response to the particular function call is not shown; or, that all the information a user is to enter is not shown.
{ }	In function and macro format specifications, braces enclose elements that are considered to be one unit of code. For example: {*keyword value*}
{ }*	In function and macro format specifications, braces followed by an asterisk enclose elements that are considered to be one unit of code, which can be repeated zero or more times. For example: {*keyword value*}*
&OPTIONAL	In function and macro format specifications, the word &OPTIONAL indicates that the arguments after it are defined to be optional. For example: UIS:SET-BUTTON-ACTION *display window action* &OPTIONAL *x1 y1 x2 y2* Do not specify &OPTIONAL when you invoke a function or macro whose definition includes &OPTIONAL.

Convention	Meaning
&KEY	In function and macro format specifications, the word &KEY indicates that keyword arguments are accepted. For example: `UIS:MOVE-VEIWPORT window &KEY :GENERAL-PLACEMENT :CENTER :ABSOLUTE-POSITION-X :ABSOLUTE-POSITION-Y` Do not specify &KEY when you invoke the function or macro whose definition includes &KEY.
<RET>	A symbol with a 1- to 3-character abbreviation indicates that you press a key on the terminal. For example: <RET> or <ESC> In examples, carriage returns are implied at the end of each line. However, the <RET> symbol is used in some examples to emphasize carriage returns.
CTRL/x	CTRL/x indicates a control key sequence where you must hold down the key labeled CTRL while you press another key. For example: CTRL/C or CTRL/Y

PART I

GUIDE TO GRAPHICS PROGRAMMING

CHAPTER 1

SYSTEM OVERVIEW

This chapter provides an overview of the VAX LISP/VMS graphics system. After a brief introduction to the system, the major sections of the chapter demonstrate how the parts of the system operate and relate to each other. This demonstration is given by means of a continuing example that you can reproduce on your VAXstation. You can start with the examples in Section 1.2, then continue through the remainder of the chapter, using the display and window objects you created.

1.1 INTRODUCTION

The VAX LISP/VMS graphics system is a collection of functions and other objects that provide an interface to the bitmap graphics capability of the VAXstation II family of workstations. You can use this system to do the following from LISP code:

- Create a virtual display that can collect graphic information
- Create one or more windows into the virtual display in order to make the graphic information appear on the screen
- Draw lines and text in the virtual display
- Respond to the movement of a pointing device, such as a mouse, and to pointer buttons
- Get input from windows by means of virtual keyboards
- Use standard LISP output functions to send output to windows through window output streams

These capabilities are available only when you are working on a VAXstation supported by VAX LISP/VMS, such as the VAXstation II. They are not supported on the VT100 or VT2xx families of terminals.

1.1.1 Relationship to MicroVMS Workstation Graphics Software

The VAX LISP/VMS graphics system is based on Version 2.0 of the workstation graphics software provided with MicroVMS. Most of the functions in the VAX LISP/VMS graphics system are directly equivalent to routines provided with Version 2.0 of the MicroVMS workstation graphics software. This equivalence is noted in the function descriptions in Part II of this manual.

VAX LISP/VMS has added a number of features to the basic MicroVMS workstation graphics software. These features are intended to make graphics easy to use in LISP code and to provide some higher-level capabilities. VAX LISP/VMS provides the following:

- LISP object types: DISPLAY, WINDOW, TRANSFORMATION, and KEYBOARD
- Window output streams, allowing text output to windows using normal LISP I/O functions
- A facility for operating on bitmaps, using the BITBLT function and BITBLT objects

The MicroVMS workstation graphics software is described in the *MicroVMS Workstation Graphics Programming Guide*. It may occasionally be useful to refer to this manual for more detail on a subject. However, the *VAX LISP/VMS Graphics Programming Guide* gives a complete description and definition of the LISP graphics system, using LISP examples to demonstrate the software.

1.1.2 Programming Considerations

With the exception of the BITBLT-related objects, all objects described in this manual are located in a package called UIS. You can gain access to these objects in one of two ways:

- Preface the name of each object with the package specifier UIS:

  ```
  (UIS:PLOT ...)
  ```

- Use the USE-PACKAGE function to make the objects available to you without the preface:

  ```
  (USE-PACKAGE "UIS")
  (PLOT...)
  ```

Although the second method is more convenient, DIGITAL recommends the first method in order to avoid real and potential conflicts with other symbols. This is particularly true if you are writing a program that uses both the graphics system and the VAX LISP Editor, since some symbol names appear in both packages.

In this chapter, all examples use the explicit preface UIS with graphics system objects. In the remaining chapters, the preface is not used in the interest of readability. The object descriptions in Part II use the explicit preface to show which package the object is in.

1.2 VIRTUAL DISPLAYS AND WINDOWS

The fundamental unit in which graphic information is collected and displayed is the virtual display. Each virtual display has the following characteristics:

- It has a coordinate system, called the world coordinate system, which is appropriate for the graphic information to be written to it.
- It has a width and height that determine the size of its windows on the screen.
- It can collect graphic information by means of its display list.
- It has a number of attributes, collected in one or more attribute blocks, that determine how its graphical information appears on the screen.

1.2.1 Creating a Virtual Display

You create a virtual display object by means of the CREATE-DISPLAY function. This function creates and returns a display having the coordinate system, width, and height that you request.

If you would like to follow along with the examples in this chapter, create a virtual display now by means of the following forms:

```
(SETF *DEMO-DISPLAY*
      (UIS:CREATE-DISPLAY
        -10.0 -10.0               ; Lower left corner coordinates
        10.0 10.0                 ; Upper right corner coordinates
        15.0 15.0))               ; Default viewport size in cm

(UIS:ENABLE-DISPLAY-LIST *DEMO-DISPLAY*)
```

You now have an object named *DEMO-DISPLAY* whose value is a virtual display. The virtual display has been made capable of recording graphic information by means of the ENABLE-DISPLAY-LIST function.

However, nothing has appeared on your screen yet. A virtual display by itself does not display anything; you must create windows into the display to make that happen.

You have given your virtual display a coordinate system defined by two points at opposite corners of a square. The points have the coordinates -10.0,-10.0 and 10.0,10.0. This square establishes the display's world coordinate system. You have also established a default width and height of 15 centimeters for the display.

1.2.2 Creating Windows

Now, create a window into the virtual display:

```
(SETF *DEMO-WINDOW-1*
      (UIS:CREATE-WINDOW *DEMO-DISPLAY*))
```

A window appears on your screen. It resembles a VT100 terminal emulator window, except that it is square and has no title or keyboard icon. The area inside the border is 15 centimeters square. This window maps into the entire virtual display; that is, it completely fills the coordinate system that you specified when you created the display.

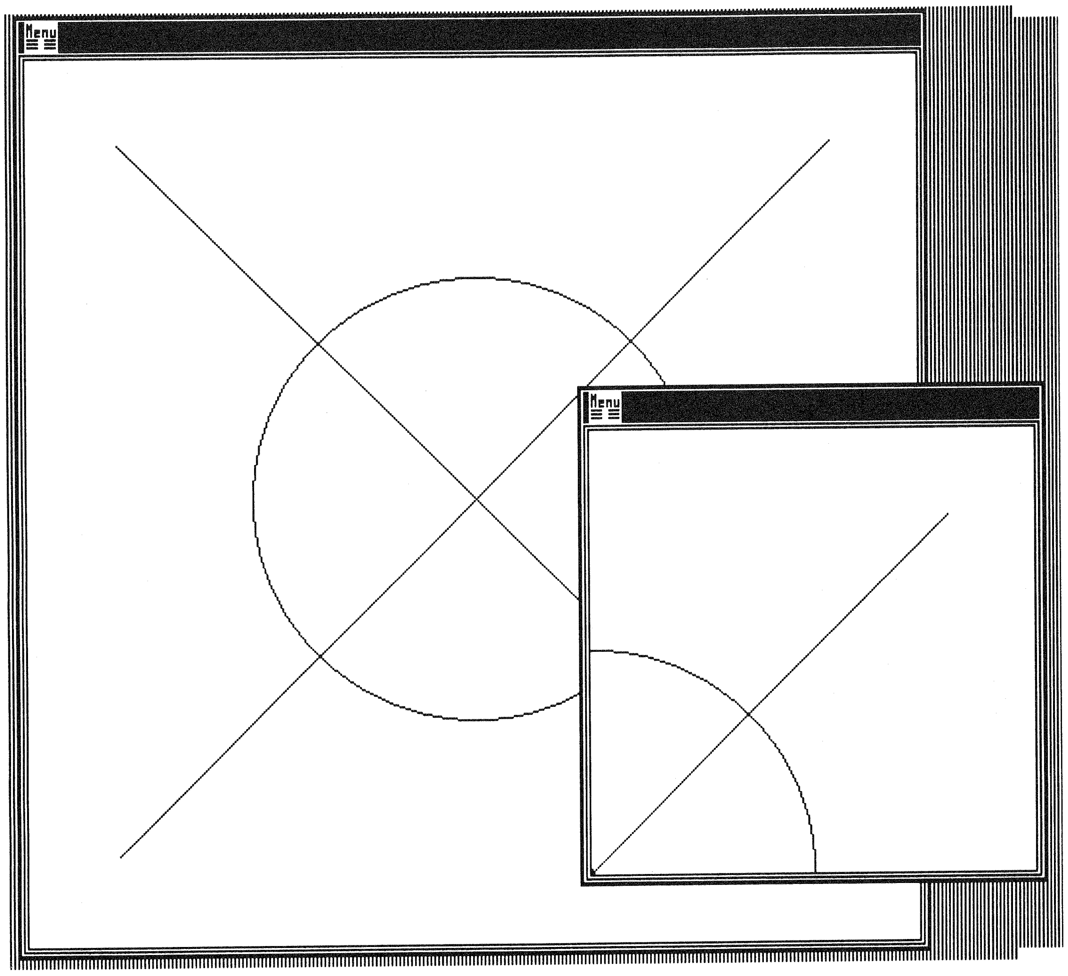
Now create a second window, one that maps into only a portion of the display:

```
(SETF *DEMO-WINDOW-2*
      (UIS:CREATE-WINDOW
         *DEMO-DISPLAY*          ; Display to map into
         0.0 0.0 10.0 10.0))     ; Corners of rectangle in display
```

A second window appears; note that its sides are only half the length of the first window's sides. The four optional arguments specify two points, 0.0,0.0 and 10.0,10.0, that define the rectangle into which this window maps. These points specify the upper-right quadrant of the virtual display. To make this relationship apparent, draw two lines and a circle in the display, as follows:

```
(UIS:PLOT *DEMO-DISPLAY* 0 -8.0 -8.0 8.0 8.0)
(UIS:PLOT *DEMO-DISPLAY* 0 8.0 -8.0 -8.0 8.0)
(UIS:CIRCLE *DEMO-DISPLAY* 0 0.0 0.0 5.0)
```

The two windows now look like this:



You can see that the smaller window reproduces the upper-right quadrant of the larger window. You could create other windows into other parts of the display.

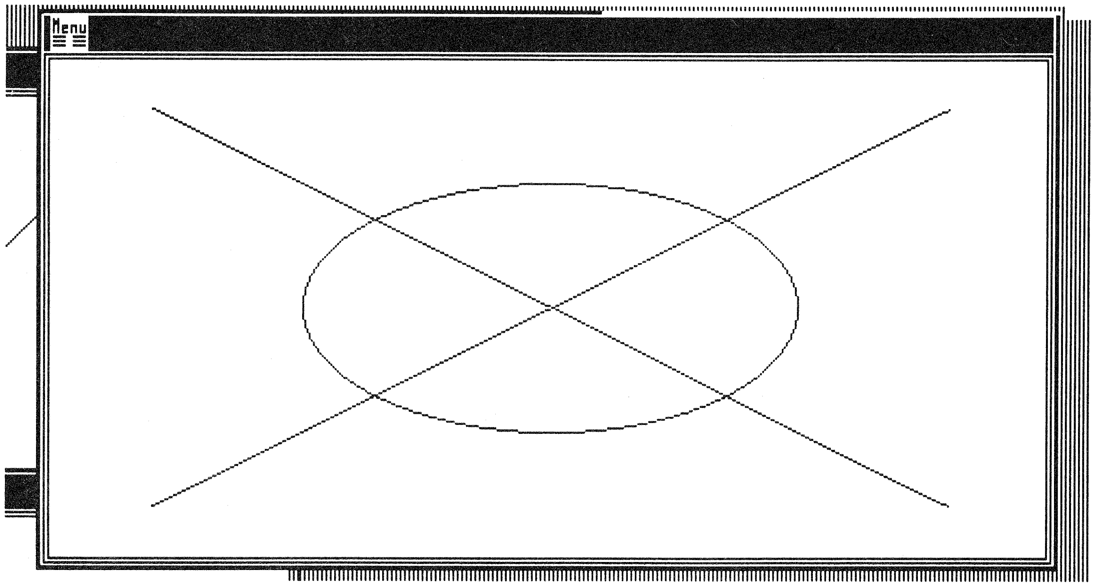
1.2.3 Windows and Viewports

Both windows you have created so far depend on the default width and height of 15 centimeters that you specified when you created the virtual display. *DEMO-WINDOW-1*, which maps into the entire display, is 15 centimeters square; *DEMO-WINDOW-2*, which maps into one-quarter of the display, is 7.5 centimeters square. You can also specify the dimensions explicitly when you create the window, overriding the default dimensions. Create a third window:

```
(SETF *DEMO-WINDOW-3*
      (UIS:CREATE-WINDOW
      *DEMO-DISPLAY*
      NIL NIL NIL NIL              ; Coordinate arguments
      :VIEWPORT-WIDTH 20.0         ; Width in centimeters
      :VIEWPORT-HEIGHT 10.0))      ; Height in centimeters
```

(Note that the four optional coordinate arguments to CREATE-WINDOW are now given as NIL. This requests that the window be mapped to the entire virtual display. COMMON LISP requires that you supply all optional arguments before any keyword-argument pairs you wish to use.)

This window looks like this:



You have created a window that maps into the entire virtual display but does not use the default dimensions of the display. As a result, the picture is distorted: stretched lengthwise, compressed in height.

This exercise brings up an important distinction, that between the window and the viewport. The window is a rectangle in a virtual display. The viewport is the corresponding rectangle on the screen. Anything that is drawn in the virtual display in the area covered by the window appears in that window's viewport. However, unless the window and viewport have the same aspect ratio (that is, the ratio of height to width), the contents of the viewport will be distorted. This is the case with *DEMO-WINDOW-3*.

Figure 1-1 illustrates the relationship among virtual displays, windows, and viewports.

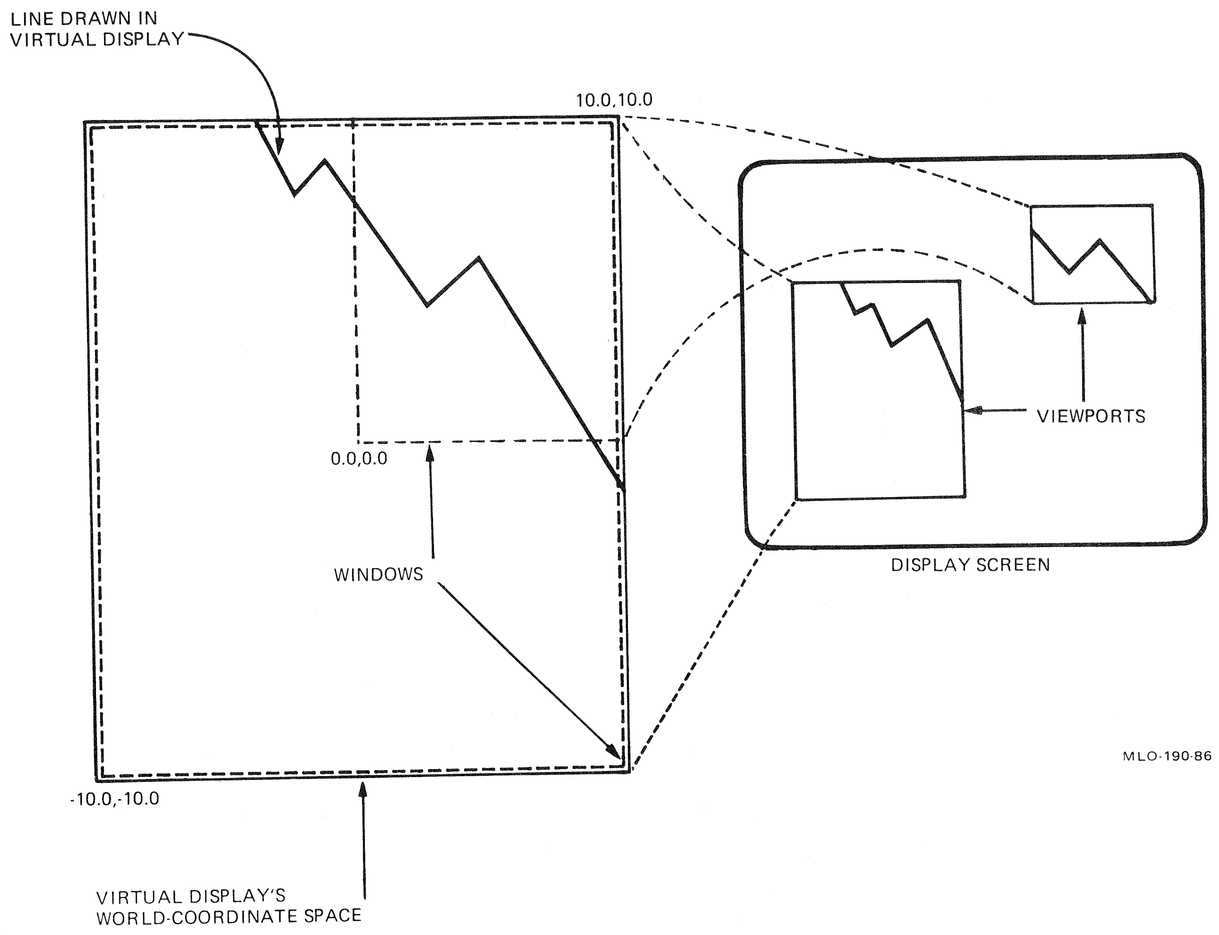

LINE DRAWN IN VIRTUAL DISPLAY
10.0,10.0
0.0,0.0
WINDOWS
-10.0,-10.0
VIRTUAL DISPLAY'S WORLD-COORDINATE SPACE
VIEWPORTS
DISPLAY SCREEN
MLO-190-86

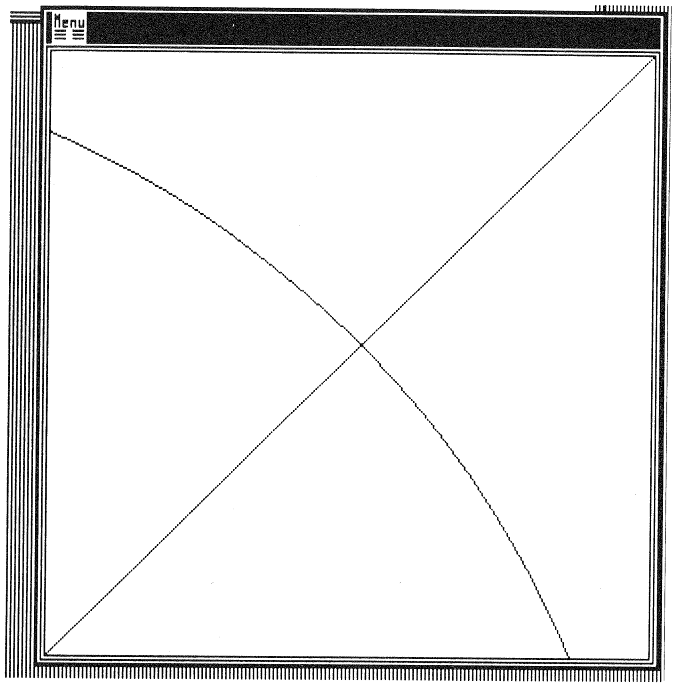
Figure 1-1: Virtual Displays, Windows, and Viewports

When you create a window, you also create its associated viewport. You always refer to a viewport by means of its window. Some functions affect windows and other functions affect viewports; both types of functions take window objects as arguments.

You can create a window that maps into a small part of a virtual display and an associated viewport that is relatively large. This has the effect of magnifying the image in the window. For example:

```
(SETF *DEMO-WINDOW-4*
      (UIS:CREATE-WINDOW
      *DEMO-DISPLAY*
      2.0 2.0 5.0 5.0            ; Small window in display
      :VIEWPORT-WIDTH 10.0       ; Large viewport on screen
      :VIEWPORT-HEIGHT 10.0))
```

The new window looks like this:



This image is "magnified" only in relation to the images in the other windows. It is important to realize that the world coordinate system of the virtual display does not imply any particular physical coordinate system. It is only the default size you specified when you created the virtual display that implies a physical size, and this default can be overridden when you create a window.

Separating the world coordinates of the virtual display from the physical size of the window allows you to set up a world coordinate system that is convenient for the data you are presenting. For example, to graph production of something between 1980 and 1988, you could create this virtual display:

```
     (UIS:CREATE-DISPLAY 1978.0 -100.0 1988.0 1000.0 30.0 15.0)
```

This display would allow you to plot raw data without first having to transform it to some physical coordinate system. (The extra space on the left and bottom edges is for labels.) The last two arguments specify the default size of a window mapped into this display in centimeters.

This concludes the demonstration portion of Section 1.2. If you wish to stop here, delete the virtual display and its associated windows as follows:

```
     (UIS:DELETE-DISPLAY *DEMO-DISPLAY*)
```

Otherwise, leave the display and *DEMO-WINDOW-1* in place and continue with the demonstrations in Section 1.3. You can get rid of the other windows with the DELETE-WINDOW function:

```
     (UIS:DELETE-WINDOW *DEMO-WINDOW-2*)
```

1.2.4 Other Operations

In addition to what you have seen in this section, you can manipulate displays, windows, and viewports in the following ways:

- Keyword arguments to CREATE-WINDOW allow you to control the appearance and location of a viewport.
- You can move a window around in a virtual display, and change the size of the window, with the MOVE-WINDOW function.
- You can move a viewport around the screen with the MOVE-VIEWPORT function. The POP-VIEWPORT and PUSH-VIEWPORT functions allow you to arrange viewports in front of or behind one another.
- You can establish an action to be taken if the user of your program attempts to delete, resize, or move a window.
- You can create a transformation into a virtual display. A transformation allows you to superimpose a second coordinate system on a virtual display.

All of these subjects are covered in Chapter 2 of this manual.

1.3 GRAPHICS OPERATIONS

Once you have created a virtual display and one or more associated windows, you use various line-drawing and text-writing functions to create images on the screen. These functions are directed into the virtual display, and the virtual display "remembers" what has been done by means of its display list. An image appears on the screen when its position in the virtual display falls under one or more of the windows that are mapped into the display.

You can create and manipulate screen images in several ways. This section introduces two: line-drawing functions and text-writing functions. The section also introduces the use of attributes and attribute blocks to modify the appearance of screen images.

The demonstrations in this section assume the existence of the virtual display *DEMO-DISPLAY*, created in Section 1.2, and its associated window *DEMO-WINDOW-1*. If you wish to try the examples in this section, create these objects if they do not already exist.

1.3.1 Drawing Lines and Circles

The PLOT function draws a point, a line, or a series of connected lines. You have already seen how PLOT can draw a single line:

```
(UIS:PLOT *DEMO-DISPLAY* 0 -8.0 -8.0 8.0 8.0)
```

The first argument to PLOT is the virtual display. This argument directs the output of PLOT to a particular display and causes the rest of its arguments to be used in the context of that display.

The second argument to PLOT is the attribute block. It is discussed later in this section.

The remaining arguments to PLOT are coordinate pairs that define points in the virtual display. In the example, the first point is at -8.0,-8.0; the second at 8.0,8.0. In this case, PLOT draws a line between those two points. If only one point had been supplied, PLOT would draw just that point.

If more than two points are supplied, PLOT draws connected lines between the points. Try the following:

```
(UIS:PLOT *DEMO-DISPLAY* 0
          -4.0 -4.0
          4.0 -4.0
          4.0 4.0
          -4.0 4.0)
          -4.0 -4.0)
```

This function call plots a complete square.

Note that the coordinates are specified as pairs of floating-point numbers. Coordinates in a virtual display's world coordinate system are always floating-point numbers.

You have also seen how to draw a circle:

```
(UIS:CIRCLE *DEMO-DISPLAY* 0 0.0 0.0 5.0)
```

As with PLOT (and all other drawing and text functions that operate in virtual displays), the first two arguments are the display and attribute block. The next two arguments specify the coordinates of the center of the circle, 0.0,0.0 in this case. The final argument specifies the radius of the circle in the units of the display's world coordinate system.

You can also draw an arc with CIRCLE by using two optional arguments to specify the beginning and end of the arc. Try the following:

```
(UIS:CIRCLE *DEMO-DISPLAY* 0
            0.0 0.0 7.0
            (/ PI 2) PI)
```

This draws an arc consisting of a quarter-circle, as shown here:

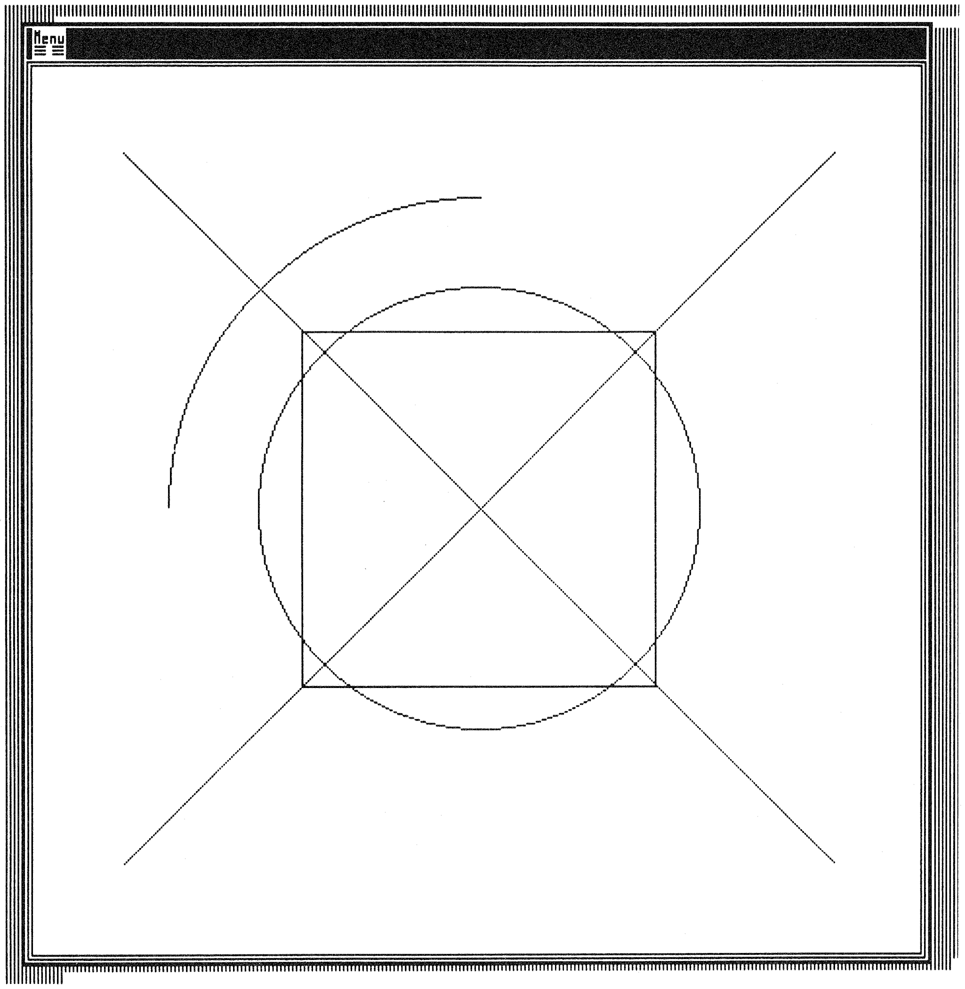


Figure 1-2 illustrates the relationship of CIRCLE's various arguments to the circle that is drawn.

The ELLIPSE function draws an ellipse. Its operation and arguments are analogous to CIRCLE, except that it has two radius arguments -- one each for the horizontal and vertical axes.

1.3.2 Writing Text

The TEXT function writes a specified string of text into the virtual display. Try the following:

```
(UIS:TEXT *DEMO-DISPLAY* 0 "This is text" 0.0 0.0)
```

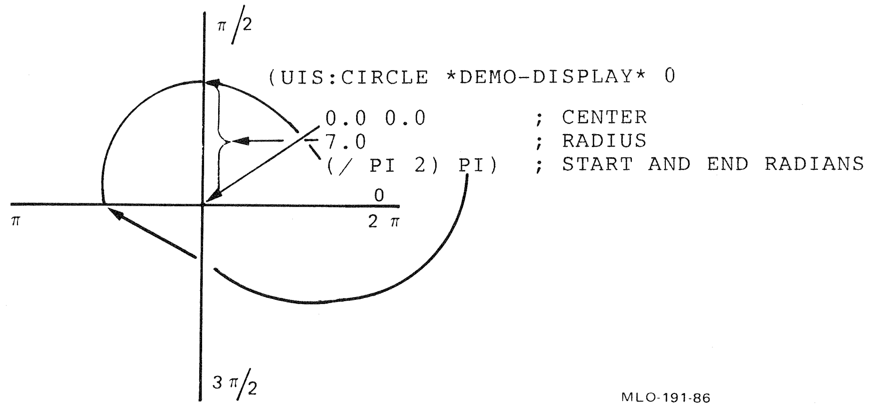

π/2
(UIS:CIRCLE *DEMO-DISPLAY* 0
0.0 0.0 ; CENTER
7.0 ; RADIUS
(/ PI 2) PI) ; START AND END RADIANS
0
2 π
π
3 π/2
MLO-191-86

Figure 1-2: Drawing a Circle

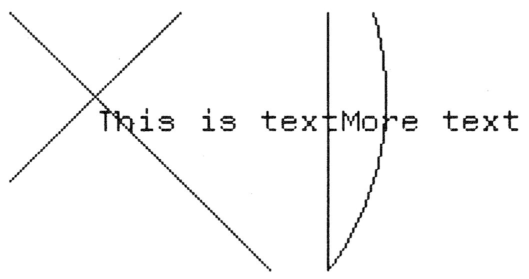
The string of text appears at the middle of the virtual display.

The first two arguments are again the virtual display and attribute block. The third argument is the text string to be displayed; it can be any LISP character string.

The last two arguments are coordinates for the point in the virtual display at which the text should start. However, these two arguments are optional. There is a default starting position for text, which is where the last text operation left off. To illustrate this, try the following:

```
(UIS:TEXT *DEMO-DISPLAY* 0 "More text")
```

The new text string appears right at the end of the previous one:



The NEW-TEXT-LINE function returns the text position to the left margin for text operations, and moves it down by the height of a line of text. Thus, NEW-TEXT-LINE is analogous to the carriage return on a typewriter. Try these functions:

```
(UIS:NEW-TEXT-LINE *DEMO-DISPLAY* 0)
(UIS:TEXT *DEMO-DISPLAY* 0 "New line of text")
```

Note that the new line of text appears at the left edge of the window. This is the default left margin. You can change this margin by modifying an attribute block, as described in the next section.

1.3.3 Using Attributes

Each of the functions in this section has taken an attribute block as its second argument. Until this point, the attribute block argument has been 0. Almost all functions that can produce something visible require an attribute block argument. An attribute block is a collection of attributes, each of which specifies the appearance of one aspect of window output. Attributes control the following characteristics, among others:

- The appearance of lines: solid, dotted, dashed, and so on
- The way in which arcs are treated: not closed, closed with a chord, or made into a pie segment
- Whether or not line figures are filled, and if so, what they are filled with
- The font in which text is written, and the left margin for text operations

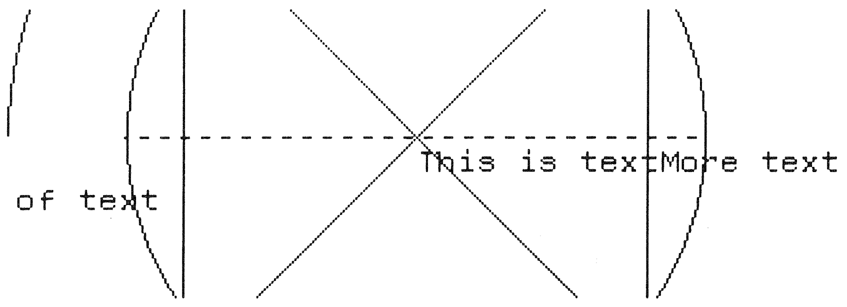
Every function that requires an attribute block uses the values of some attributes from the specified block to influence the appearance of the output. No single function uses all the attributes in a block. Drawing functions use some attributes and text functions use others, while some attributes are common to both drawing and text.

Attribute block 0 contains a collection of default attribute values that are appropriate for many operations. To use a different value for an attribute, you must first modify an attribute block so that it has the appropriate value for that attribute. You then supply the modified attribute block as an argument to the drawing or text function.

To modify an attribute block, use the SET-ATTRIBUTE function. This function takes an attribute block and copies its values to an attribute block, changing the value of one attribute in the process. The following example modifies attribute block 1 so that it causes lines to be drawn dashed instead of solid. Try modifying this attribute block, and then draw a line using it:

```
(UIS:SET-ATTRIBUTE *DEMO-DISPLAY* 0 1 :LINE-STYLE :DASHED)
(UIS:PLOT *DEMO-DISPLAY* 1 -5.0 0.0 5.0 0.0)
```

The PLOT function call draws a dashed horizontal line:



The SET-ATTRIBUTE function call in the example copies values from attribute block 0 to attribute block 1. The only difference between attribute blocks 0 and 1 is the value of the :LINE-STYLE attribute, which is now :DASHED instead of :SOLID.

Note that the SET-ATTRIBUTE function requires a virtual display argument. This is because attribute blocks are associated with virtual displays. Each virtual display has 256 attribute blocks associated with it. Attribute block 0 is the same for all virtual displays, but attribute block 1 may be different from one display to the next.

You can use SET-ATTRIBUTE to change one value in an attribute block that has already been modified, simply by specifying the same attribute block for input and output. The following example further modifies attribute block 1:

```
(UIS:SET-ATTRIBUTE *DEMO-DISPLAY* 1 1 :ARC-TYPE :CHORD)
```

Now, in addition to specifying a dashed line style, attribute block 1 indicates that arcs should be closed by drawing a line between their endpoints.

You can modify any attribute block except attribute block 0. Attribute block 0 can only be read.

One frequently-changed attribute is the font in which text is written. A font consists of a collection of printing characters in a particular type style and size. Fonts are contained in a directory with the logical name SYS$FONT; each file in this directory represents a single font.

For a detailed discussion of attributes, see Section 3.3. The discussions of various graphic operations in Chapter 3 contain information on individual attributes.

1.3.4 Other Operations

In addition to what you have seen in this section, the following graphic operations are available to you:

- The PLOT-ARRAY function plots lines using entries from two vectors as the X and Y coordinates of each point.
- A number of functions allow you to measure strings of text in a specified font and to precisely position text on the screen.
- The READ-IMAGE-PIXEL and IMAGE functions store a screen image in memory and write an image from memory to a virtual display, respectively. The BITBLT family of functions and objects perform various operations on screen images stored in memory.
- The ERASE function removes graphic objects from a virtual display.
- The BEGIN-SEGMENT and END-SEGMENT functions allow the use of temporary attribute blocks over specified groups of graphic operations.
- Device coordinate counterparts to most of the graphic functions allow you to operate at the pixel level, offering potential gains in speed and accuracy.

All of these subjects are treated in greater detail in Chapters 3 and 4.

1.4 POINTER OPERATIONS

Workstations supported by the VAX LISP graphics system come equipped with a pointing device, typically a mouse. The pointing device serves two functions: it directs a pointer cursor around the screen, and it has one or more buttons that allow you to make requests of the workstation. Working together, the pointer and its buttons allow you to select items from menus, move windows around the screen, and so on.

A number of functions allow your programs to make use of the pointing device in various ways. You can:

- Determine the position of the pointer cursor in the coordinate system of a virtual display
- Move the pointer cursor around the screen under program control
- Determine the state of the pointing device buttons

- Specify a function to be invoked when the pointer cursor moves within or exits a specified area of a display
- Specify a function to be invoked when a button on the pointing device is pressed or released

For information and examples on pointer operations, see Chapter 5.

1.5 KEYBOARD INPUT FROM WINDOWS

You can capture keystrokes directed at a particular window by means of a virtual keyboard. A virtual keyboard is a virtual input device that you create with the CREATE-KB function. Various functions let you associate a virtual keyboard with one or more windows. When the user, or your program, makes a particular virtual keyboard active, keystrokes at the physical keyboard are transmitted through the virtual keyboard to your program. Your program can use these keystrokes in one of two ways:

- Synchronously, by means of the READ-KB-CHAR function. This function returns each character from a particular virtual keyboard in turn.
- Asynchronously, by means of the SET-KB-ACTION function. This function establishes an action, such as an interrupt function, to execute each time a character is typed through a particular virtual keyboard.

Chapter 6 describes the use of virtual keyboards.

1.6 WINDOW OUTPUT STREAMS

You can establish an output stream to a window. Output sent to that stream by any of the standard COMMON LISP output functions appears in the window. This facility includes the following features:

- Choice of horizontal text wrapping or truncation
- Choice of vertical text scrolling or wrapping
- Control over the part of the window in which text appears
- Control over the size and style of text

Chapter 7 describes window output streams.

CHAPTER 2

VIRTUAL DISPLAYS, WINDOWS, AND TRANSFORMATIONS

This chapter describes the various objects and concepts involved in the display of graphic information on a workstation screen. The actual graphic operations are described in Chapter 3; this chapter defines the framework within which those operations occur. You need an understanding of the material presented in this chapter before you can use the information in subsequent chapters.

This chapter consists of the following:

- An overview of how graphic information appears on the screen
- A description of the coordinate systems to which you have access
- A description of virtual displays, which accept and store graphic information
- A description of windows and viewports, which allow graphic information stored in virtual displays to appear on the screen
- A description of transformations, which allow you to superimpose alternate coordinate systems on a virtual display

If you are completely unfamiliar with these topics, you may wish to read Section 1.2, which provides an elementary introduction to them.

2.1 HOW AND WHEN GRAPHIC INFORMATION IS DISPLAYED

In order to display graphic information on the workstation screen, you must do three things:

1. Create a virtual display.
2. Create a window into the virtual display.

3. Draw lines or write text into the virtual display, within the area covered by the window.

The virtual display is the fundamental unit of the VAX LISP graphics system. It is a two-dimensional space that can store graphic information. All graphic operations are performed with respect to some virtual display. Thus the first step in any graphics program must be to create at least one virtual display.

When you create a window, you define a rectangle in the virtual display and a corresponding rectangle (the viewport) on the screen. Anything in the virtual display that falls within the bounds of the window rectangle appears in the viewport. You can create many windows of various sizes that map into the same virtual display. Windows can overlap one another.

When you use a function that draws a line or writes text, you cause that information to be stored in the virtual display. (This assumes that the virtual display's display list has been enabled -- see Section 3.1. The display list is disabled by default.) Conceptually, that information is stored at a certain location in the two-dimensional space of the display. If a window happens to include that location, the information will also appear on the screen. If a window is later created that includes that location, the information will appear in the associated viewport. If two windows include that location, the information will appear in both corresponding viewports.

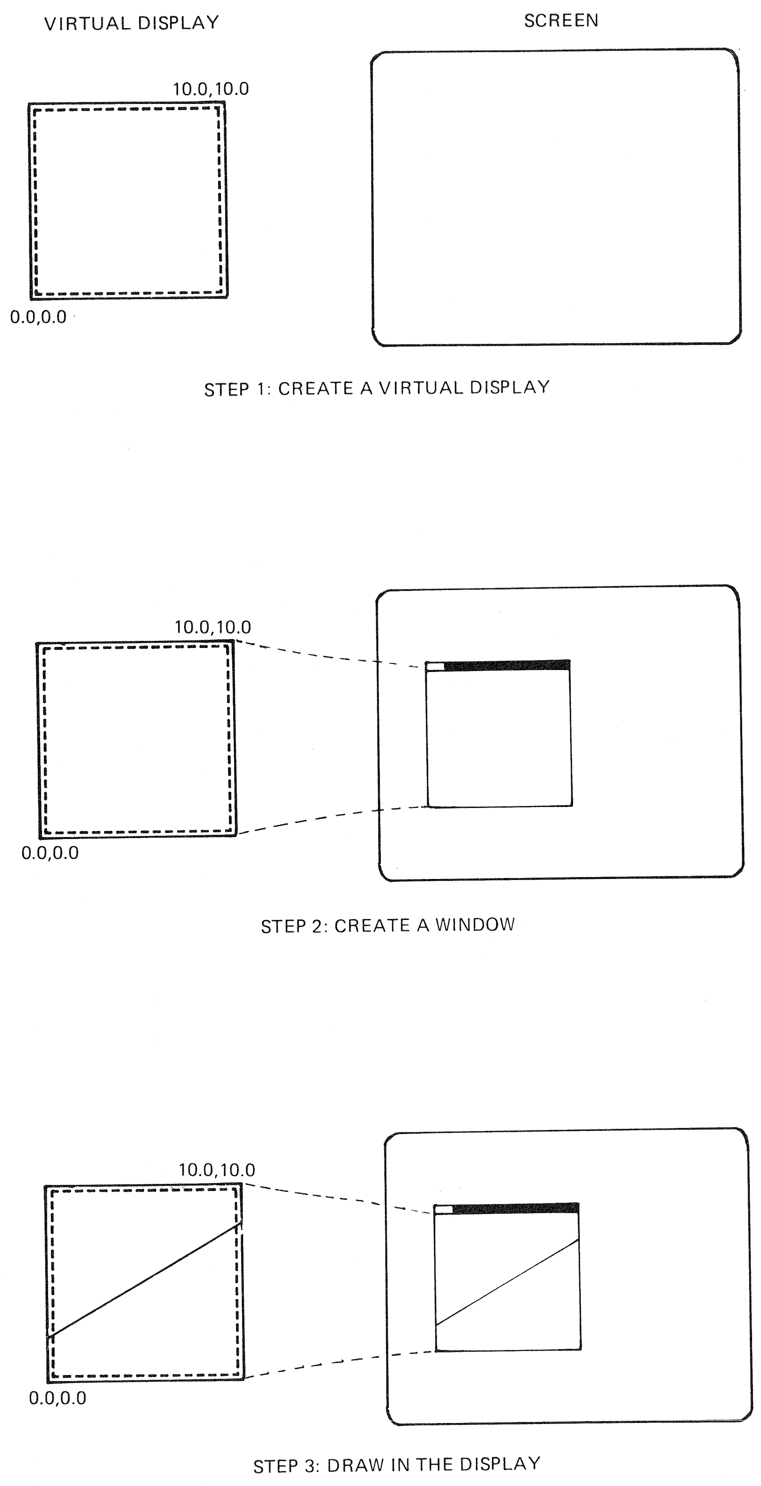
Figure 2-1 shows how graphic information is displayed.

2.2 COORDINATE SYSTEMS

The VAX LISP graphics system employs three coordinate systems for three different purposes:

- World coordinates specify locations in a virtual display
- Device coordinates specify locations in a viewport
- Screen coordinates specify locations on the display screen

Each of these coordinate systems is a two-dimensional Cartesian system. In a Cartesian coordinate system, the location of a point is expressed as a pair of numbers. The first number in the pair, the X value, specifies the horizontal displacement of the point from an origin point. The second number, the Y value, specifies the vertical displacement. The coordinates of the origin are expressed as 0.0,0.0 (for world and screen coordinates) or 0,0 (for device coordinates).



```
(SETF DISPLAY (CREATE-DISPLAY
                    0.0 0.0
                    10.0 10.0
                    5.0 5.0))

(ENABLE-DISPLAY-LIST DISPLAY)
```

```
(SETF WINDOW (CREATE-WINDOW
                    DISPLAY))
```

```
(PLOT DISPLAY 0
      0.0 2.0 10.0 8.0)
```

Figure 2-1: Displaying Information on the Screen

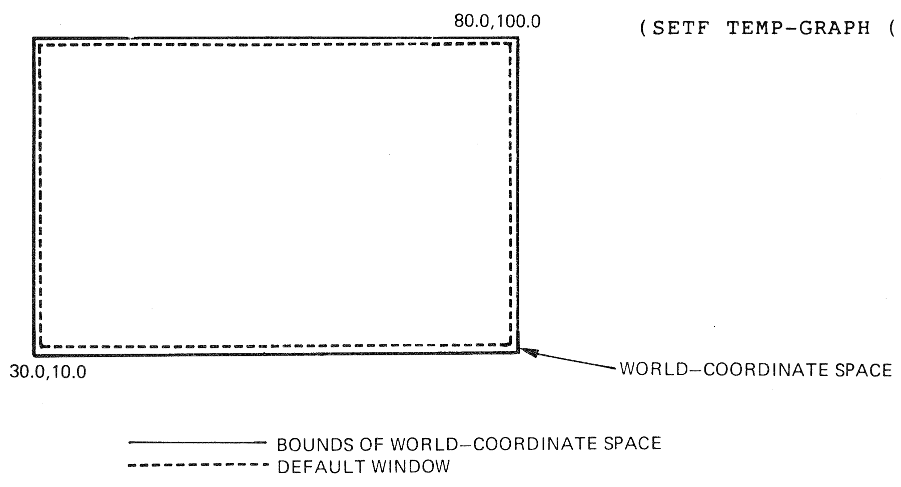
2.2.1 World Coordinates

Each virtual display that you create has its own world coordinate system. The characteristics of a world coordinate system are the following:

- You can set up a world coordinate system so that it is appropriate for the data you are trying to present.
- The world coordinate system establishes bounds outside of which your data points should not fall.
- You specify world coordinate values as pairs of floating-point numbers.

Conceptually, a virtual display's world coordinate system is a two-dimensional space in which points are specified by pairs of floating-point coordinates. When you create a virtual display, you specify a rectangle in the world coordinate space that you intend to contain all of your graphic information. You select the bounds of this rectangle to be appropriate for the data you wish to display.

For example, suppose you wish to graph chemical activity, measured by the amount of gas released in cubic centimeters per second, against temperature measured in degrees Celsius. The amount of gas ranges from 10 to 100 cubic centimeters per second, while the temperature ranges from 30 to 80 degrees. When you create your virtual display, you specify a rectangle as shown in Figure 2-2.



```
(SETF TEMP-GRAPH (CREATE-DISPLAY
                       30.0 10.0
                       80.0 100.0
                       27.0 15.0))
```

Figure 2-2: Example of a World Coordinate System

The rectangle indicates where a default window will be located in the virtual display and how large it will appear on the screen. You may also request that graphic information lying outside the rectangle be made invisible, or clipped.

Every virtual display has its own world coordinate space. Thus, graphic information written in one virtual display can never appear in another virtual display, even though both virtual displays may map the same portion of their world coordinate spaces to the screen.

2.2.2 Device Coordinates

A display device, such as a display screen or a laser printer, forms an image by selectively activating thousands of tiny points in a grid. These points are called picture elements, or pixels for short. Figure 2-3 shows how individual pixels make up an image. When the pixels are small enough, the human eye blends them into a smooth form.

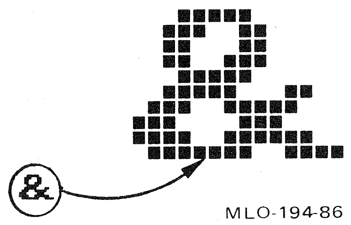


Figure 2-3: Making an Image from Pixels

The VAX LISP graphics system provides direct access to the display's pixels through the device coordinate system. Each viewport displayed on the screen has its own device coordinate system. A device-coordinate pair specifies the location of a pixel in a particular viewport. A function that gives location information in the device coordinate system draws directly into the viewport, not into a virtual display. Figure 2-4 illustrates this.

The origin of the device coordinate system is always at the lower left corner of the viewport. The upper bounds of the system for a given viewport are determined by the number of pixels in the viewport's horizontal and vertical dimensions. Points that lie outside the bounds of the system are never displayed.

Since pixels are indivisible units, device coordinates are always expressed as integers.

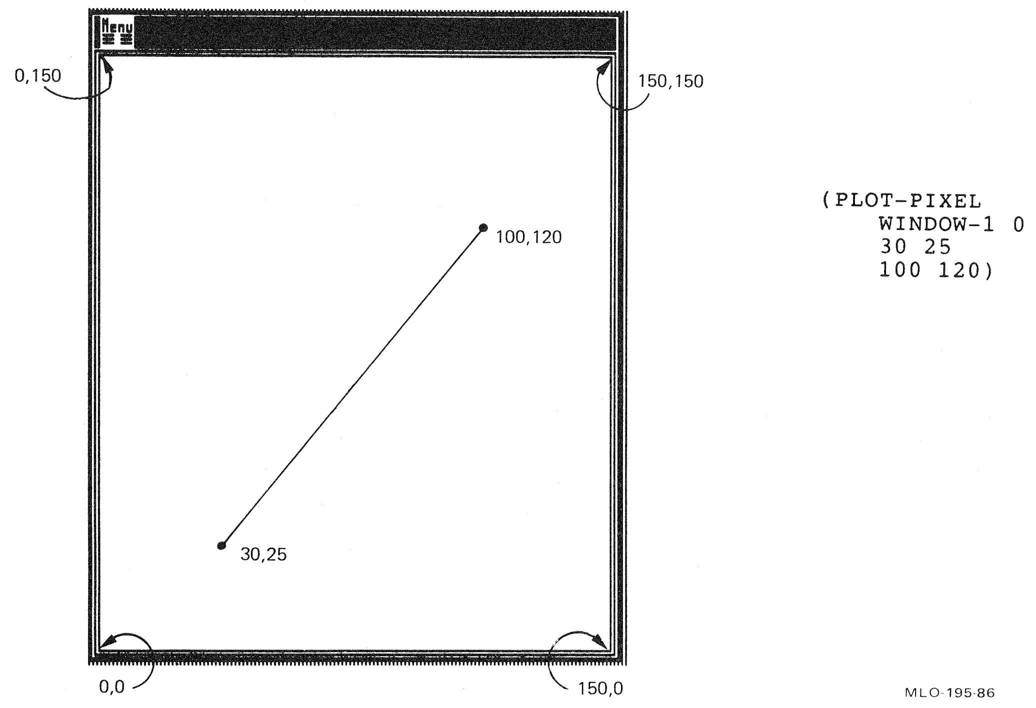


Figure 2-4: The Device Coordinate System

All functions that draw lines or write text into a virtual display have counterparts that operate in a viewport. These functions have the suffix "-PIXEL" added to their names. They have the following characteristics:

- They take a window as their first argument instead of a virtual display. The output occurs in the viewport associated with the window.
- They take device-coordinate location arguments (integers) instead of world-coordinate arguments (floating-point numbers).
- They do not store graphic information in a virtual display.
- They can be faster than their world-coordinate counterparts.

In some situations, it is appropriate to use device coordinates instead of world coordinates. Chapter 3 covers this subject in detail.

Since device coordinates specify pixel locations, the size and shape of an image specified in the device coordinate system depends on the size and shape of a display device's pixels. Thus, the same function may present a different appearance on different devices. A common problem arises when pixels are not "square," that is, when they are not the same in width and height. Figure 2-5 illustrates the difference between square and nonsquare pixels.

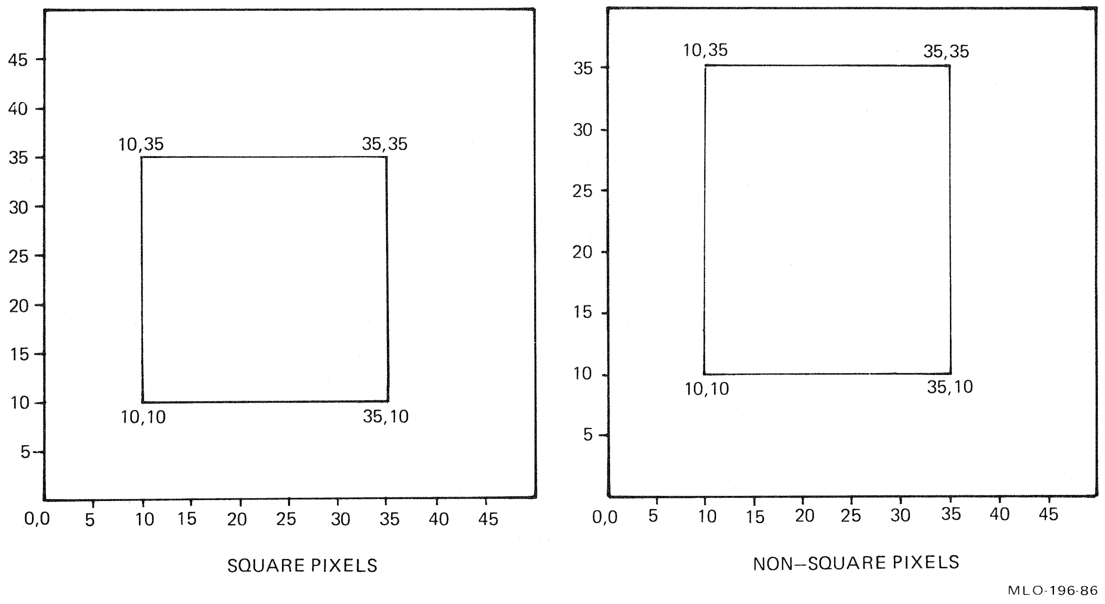


Figure 2-5: Square and Nonsquare Pixels

The size of a pixel is expressed in terms of the display's horizontal and vertical resolution -- that is, the number of pixels per centimeter, measured horizontally and vertically. For example, if a display has a resolution of 40 pixels/cm both horizontally and vertically, that display has square pixels. The following function will draw a 1-centimeter square on that display:

```
(PLOT-PIXEL *DEMO-WINDOW-1* 0
            40 40 80 40 80 80 40 80 40 40)
```

Another display device may have a resolution of 40 pixels/cm horizontally but only 35 pixels/cm vertically. On that device, the same function would plot a rectangle 1.0 centimeter wide and 1.14 centimeters high.

You can use the GET-DISPLAY-SIZE function to determine the horizontal and vertical resolution of a display device. This function returns six values; the third and fourth values are the horizontal and vertical resolutions, respectively. They are expressed as pixels/centimeter.

2.2.3 Screen Coordinates

A screen coordinate system is defined for the entire display device. The units of the screen coordinate system are centimeters, and the origin of the system is at the lower-left corner of the display device. The upper bounds of the system are determined by the size of the screen. You specify locations on the screen with pairs of floating-point numbers.

You can use the GET-DISPLAY-SIZE function to determine the size of a display device, and thus the bounds to its coordinate system. GET-DISPLAY-SIZE returns six values. The first two are the width and height of the display screen in centimeters, and the last two are the width and height in pixels.

Since VAX LISP graphics functions cannot draw directly on the display screen, the usefulness of screen coordinates is limited to positioning viewports on the screen. Section 2.4 contains examples of the use of screen coordinates.

2.3 CREATING AND MAINTAINING VIRTUAL DISPLAYS

This section explains how to create a virtual display, how to gain access to it, how to delete it, and how to specify its properties. Other sections of this manual describe other aspects of a virtual display:

- Section 2.2.1 describes the coordinate system used within a virtual display.
- Section 3.1 describes the virtual display's display list, by means of which it stores graphic information. You must enable the display list if you want the virtual display to store information.
- Section 3.3 describes the virtual display's attribute blocks, by means of which it modifies the appearance of graphic operations.
- Section 3.4 describes the virtual display's color map, which controls the display color of objects in the virtual display.

2.3.1 Creating and Accessing a Virtual Display

The CREATE-DISPLAY function creates and returns a virtual display object. You should retain the virtual display, for example by assigning it to a symbol:

```
     (SETF *DISPLAY-1* (CREATE-DISPLAY
                          0.0 0.0 50.0 200.0 20.0 10.0))
```

You can then use the symbol as an argument to the numerous functions that require a virtual display.

The DISPLAYP function returns T if the value of its argument is a virtual display object, and NIL otherwise.

Each virtual display has an identification number (the vd_id) that is assigned by the MicroVMS workstation graphics software. You can use the UIS-ID function to return this number, should you need it for use with CALL-OUT.

If you do not retain the value returned by CREATE-DISPLAY, you have no means of access to the virtual display. You will be unable to create windows into it, to draw in it, or to delete it. (Should you "lose" a virtual display, the LIST-ALL-DISPLAYS function returns a list of all the displays that you have created and have not yet deleted.)

Once you have created a virtual display, you must enable its display list if you want to store information in the display. Section 3.1 describes the display list.

2.3.2 Deleting a Virtual Display

Virtual displays consume system resources. If you have not explicitly deleted a virtual display, the garbage collector cannot free the dynamic memory that it occupies. Therefore, you should take care to delete virtual displays when you are done with them. The DELETE-DISPLAY function deletes a virtual display:

```
     (DELETE-DISPLAY *DISPLAY-1*)
```

The DELETE-DISPLAY function also deletes any windows and transformations associated with the display that it deletes.

2.3.3 Arguments to CREATE-DISPLAY

CREATE-DISPLAY takes six arguments. The first four arguments specify two world-coordinate points that define the lower-left and upper-right corners of a rectangle. This rectangle serves as a reference when creating windows into the virtual display. It also limits the coordinates of the information you can write into the display.

CREATE-DISPLAY's last two arguments specify default screen dimensions for windows created into the display. They establish the width and height, in centimeters, of a window that corresponds to the rectangle defined by the first four arguments. Windows that occupy only a portion of the rectangle will be proportionally smaller.

Figure 2-6 illustrates how CREATE-DISPLAY's arguments relate to the world coordinate system.

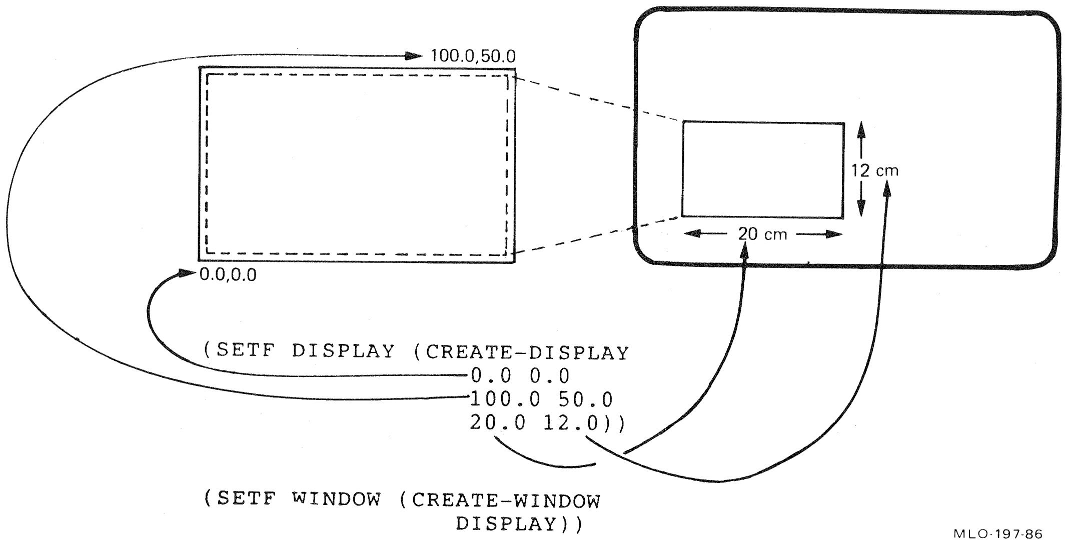


Figure 2-6: Arguments to the CREATE-DISPLAY Function

2.4 CREATING AND MANIPULATING WINDOWS AND VIEWPORTS

This section explains how to create and delete a window into a virtual display, how to control the appearance of the viewport associated with the window, and what you can do with windows and viewports after they are created.

2.4.1 Creating and Accessing Windows

The CREATE-WINDOW function creates a window into a virtual display, and a viewport associated with the window on the screen. CREATE-WINDOW returns the window object it created. If you wish to manipulate the window or viewport after it has been created, or to delete the window, you must retain the returned window object. For example:

```
(SETF *WINDOW-1* (CREATE-WINDOW *DISPLAY-1*))
```

The window object returned by CREATE-WINDOW is your means of access to both the window and its associated viewport. The following functions provide information about windows:

- The WINDOWP function returns T if its argument is a window object and NIL otherwise.
- The WINDOW-DISPLAY function returns the display into which its argument, a window, is mapped.
- The DISPLAY-WINDOWS function returns a list of the windows mapped into a specified display.
- The UIS-ID function returns the MicroVMS workstation graphics software window ID (wd_id) for the window.

If you use the CREATE-WINDOW function without any optional arguments (as in the example just given), the function creates and returns a window whose bounds are the world-coordinate rectangle you specified with CREATE-DISPLAY (see Section 2.3.1). This window is the virtual display's default window.

The width and height of the viewport associated with a default window are specified by the last two arguments to the CREATE-DISPLAY function call that created the virtual display. The viewport also has the following default characteristics:

- It has a border.
- It has a banner but no title.
- Its placement on the screen is determined entirely by the graphics software.
- It is immediately visible on the screen.

Section 2.4.1.2 explains how to specify these aspects of a viewport's appearance when creating a window.

2.4.1.1 **Controlling Window Coordinates** - Four optional arguments to the CREATE-WINDOW function allow you to create a window that maps into a world-coordinate rectangle other than the default rectangle established with CREATE-DISPLAY. The four optional arguments are floating-point numbers that specify coordinates of the lower-left and upper-right corners of the desired rectangle. Figure 2-7 illustrates the use of these arguments.

Note that if you want to use keyword arguments with CREATE-WINDOW, you must supply the four optional arguments. You can use values of NIL to request the default window dimensions.

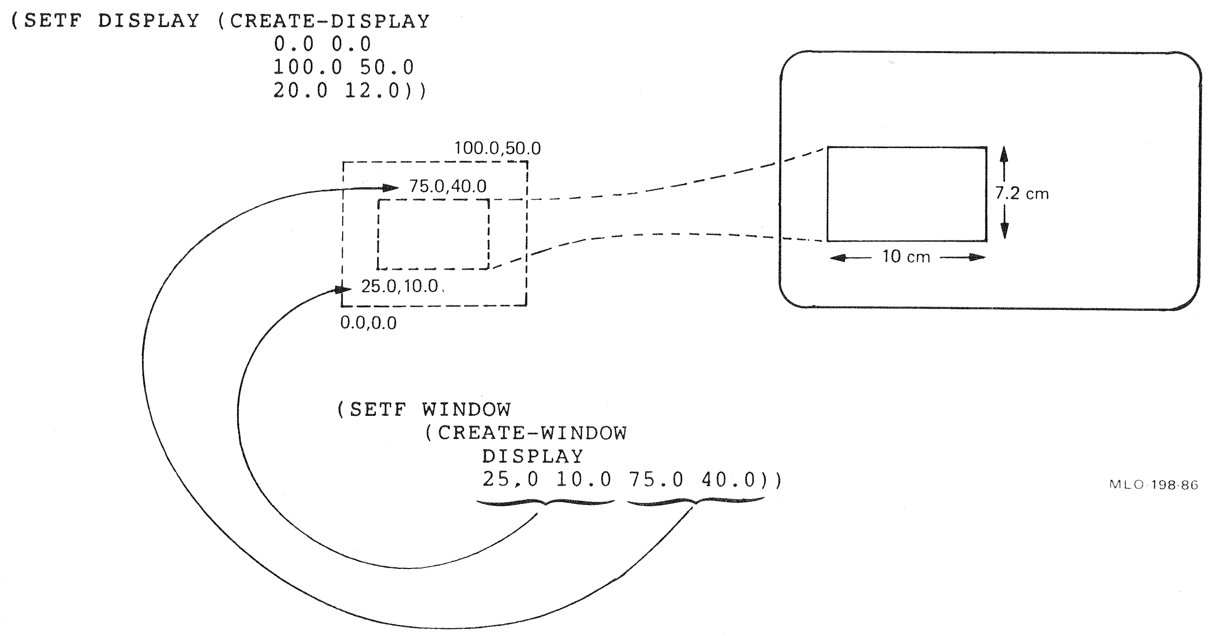


Figure 2-7: Specifying Window Coordinates with CREATE-WINDOW

Unless you also specify :VIEWPORT-WIDTH or :VIEWPORT-HEIGHT (see Section 2.4.1.2), the screen dimensions of a viewport associated with a nondefault-sized window are determined by the virtual display's default screen dimensions. When you use the CREATE-DISPLAY function, you establish a scale between the units of your world coordinate system and the physical dimensions of the display device. For example, if your default window is 100 world-coordinate units wide and your default viewport is 10 centimeters wide, the default horizontal scale is 10 units/centimeter. The vertical scale may be the same or different, depending on the arguments to CREATE-DISPLAY. When you create a window without specifying the viewport size, the graphics system preserves this scale. For example, if you use optional arguments to create a window that is 50 units wide, the corresponding viewport will be 5 centimeters wide.

2.4.1.2 **Controlling Viewport Characteristics** - A number of keyword arguments to the CREATE-WINDOW function allow you to control the size, appearance, and screen placement of a viewport. Figure 2-8 illustrates a typical viewport and identifies its various components.

The :VIEWPORT-WIDTH and :VIEWPORT-HEIGHT arguments allow you to specify, in centimeters, the width and height of the viewport's picture area. The picture area is the portion of the viewport that is mapped to the window in the virtual display.

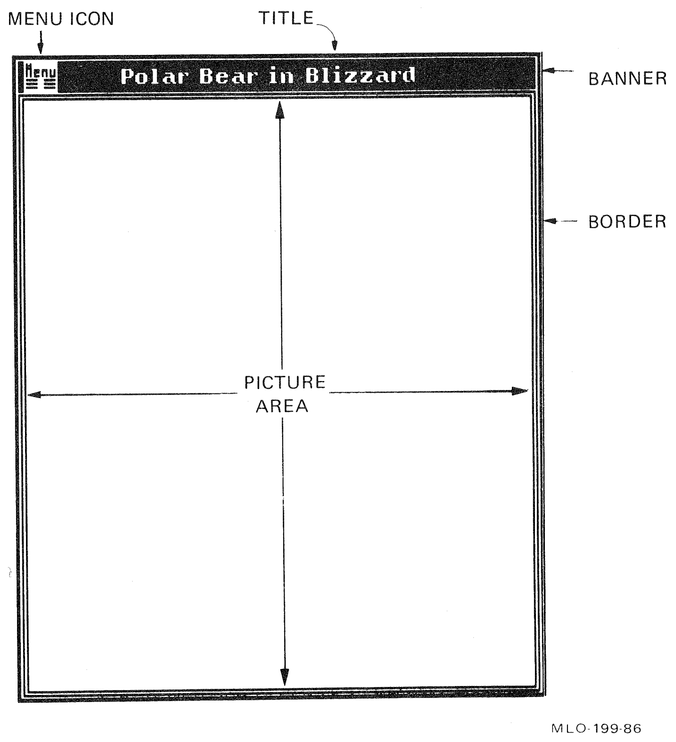


Figure 2-8: Viewport Components

You can supply a title for the viewport with the :BANNER-TITLE argument. The :NOBANNER and :NOBORDER keywords, when supplied with values of T, cause the banner and border to be removed from the viewport.

You can control the placement of a window on the screen in one of two ways:

- Absolutely, by specifying the viewport's screen coordinates
- Generally, by specifying the area of the screen in which to place the viewport

Use the :ABSOLUTE-POSITION-X and :ABSOLUTE-POSITION-Y keywords to position the viewport at an exact location on the screen. These give the location, in centimeters, of the viewport's lower-left corner. (If you also supply the :CENTER keyword with a non-NIL value, the viewport will be centered on the specified location rather than having its lower-left corner at that location.)

Use the :GENERAL-PLACEMENT keyword to specify a general screen location for the viewport. The value you supply with :GENERAL-PLACEMENT can be :TOP, :BOTTOM, :LEFT, :RIGHT, or a list of two of these, such as '(:TOP :RIGHT). The graphics system will attempt to place the viewport in the area you request, although factors such as the proximity of other viewports may prevent it from doing so.

You can request that a viewport be created off-screen by using the :INVISIBLE keyword with a non-NIL value. The window and viewport are created but the viewport does not appear. You can later use the MOVE-VIEWPORT function to move the viewport onto the screen (see Section 2.4.4).

2.4.2 Deleting Windows

The DELETE-WINDOW function deletes a window and causes the window's associated viewport to disappear from the screen. The function does not delete the virtual display into which the window is mapped, nor does it affect any information that is stored in the virtual display.

If there are any window output streams associated with the window, those streams are closed.

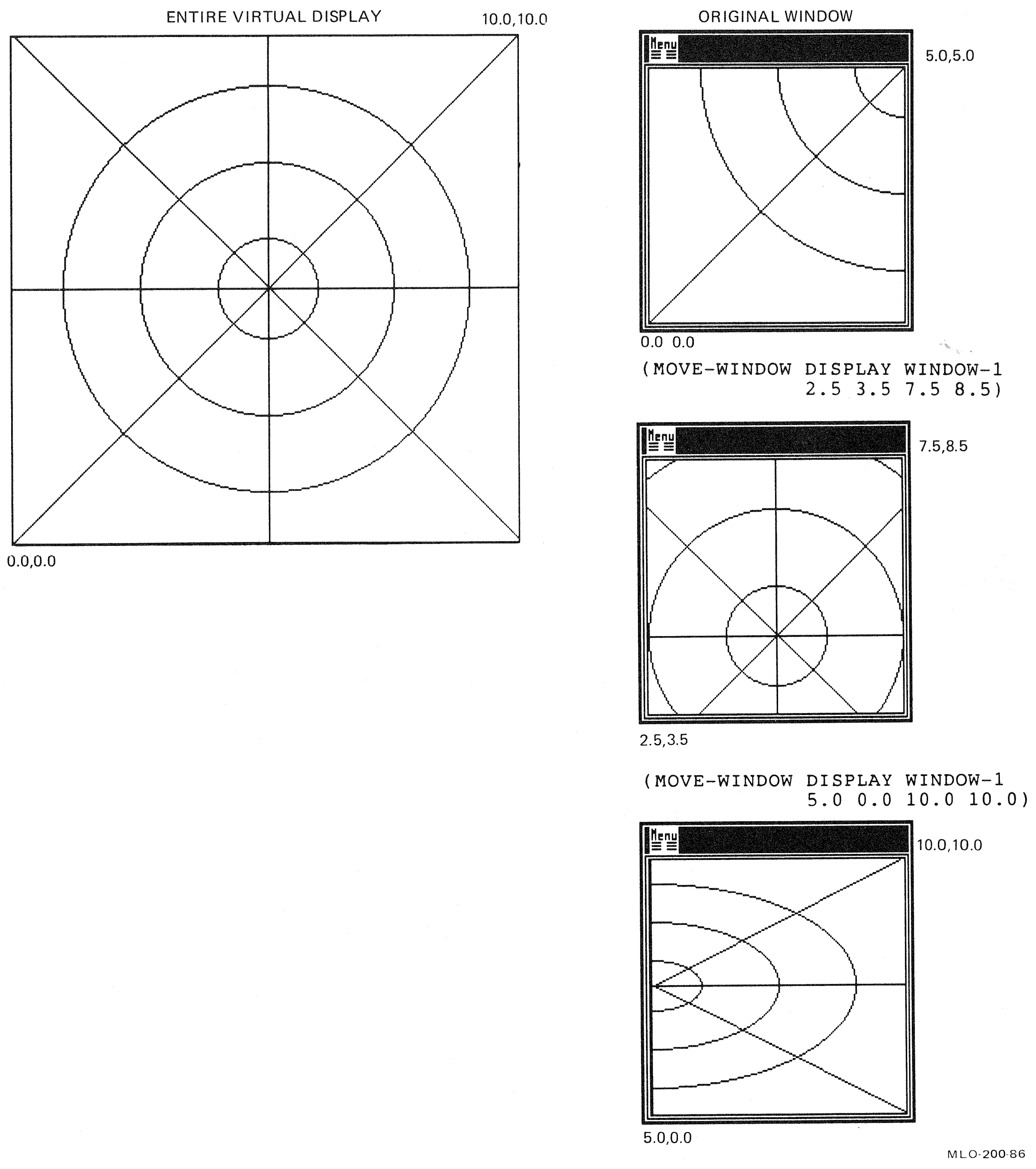
If you have not explicitly deleted a window, the garbage collector cannot free the dynamic memory that it occupies.

2.4.3 Moving and Resizing Windows

Two functions allow you to alter a window's coordinates in a virtual display and change its size in the display. The first function, MOVE-WINDOW, changes the location and/or size of the window without affecting the size of the viewport. The second function, RESIZE-WINDOW, can affect both the window and the viewport. Both functions can change the scale of objects displayed in the viewport.

The MOVE-WINDOW function shifts a window to a new rectangle in the virtual display. Any information in the display at that new location appears in the window's associated viewport. The dimensions (in world coordinate units) of the new rectangle may be the same as those of the window's previous rectangle. In this case the information displayed in the viewport is drawn to the same scale as it was previously. If the new rectangle's dimensions are different, then the scale of the displayed information will also be different. Figure 2-9 illustrates two calls to MOVE-WINDOW, one without and one with rescaling.

The RESIZE-WINDOW function allows you to change the size of the window by specifying a new size for its associated viewport. It also allows a change of the location of the window in the display and of the viewport on the screen. See Section 2.4.6.2 for more information about the RESIZE-WINDOW function.



```
(MOVE-WINDOW DISPLAY WINDOW-1
             2.5 3.5 7.5 8.5)
```

```
(MOVE-WINDOW DISPLAY WINDOW-1
             5.0 0.0 10.0 10.0)
```

Figure 2-9: Moving a Window

2.4.4 Moving Viewports

The MOVE-VIEWPORT function allows you to move a viewport and its contents around the screen. The arguments to MOVE-VIEWPORT are the window associated with the viewport and several keyword arguments to control placement. These keyword arguments are the same as the arguments to CREATE-WINDOW:

- :ABSOLUTE-POSITION-X and :ABSOLUTE-POSITION-Y, to specify an absolute position for the viewport, and :CENTER, to specify that the viewport be centered on that location
- :GENERAL-PLACEMENT, to request a general position for the viewport
- :INVISIBLE with a non-NIL argument, to request that the viewport be moved off the screen.

When you use MOVE-VIEWPORT to move a viewport on the screen, all the graphic information in the viewport moves with it. This includes information stored in the virtual display associated with the viewport's window, and any information that was written directly into the viewport. For example, if you used the PLOT-PIXEL function to draw lines in the viewport, those lines would be preserved, even though they are not stored in the virtual display.

2.4.5 Determining and Controlling Viewport Occlusion

When you create a window, its associated viewport hides, or occludes, any other viewports that it overlaps. The contents of the occluded viewports are not lost but they cannot be seen. Four functions allow you to determine if a particular viewport (or location in a viewport) is occluded, and to control which viewports are occluded and which are exposed.

The GET-VISIBILITY function allows you to determine if all or a specified part of a window is visible on the screen. Depending on the arguments you supply, GET-VISIBILITY checks to see if a point, a rectangle, or the window is entirely visible. The coordinates of the point and rectangle are supplied using the world coordinate system. The GET-VISIBILITY-PIXEL function performs the same operation except that it allows you to specify a point or rectangle in the device coordinate system of the viewport.

Two functions, POP-VIEWPORT and PUSH-VIEWPORT, allow you to control viewport visibility. Each takes a single argument, a window. POP-VIEWPORT brings the viewport associated with the window to the front so that it occludes any viewports that it overlaps. PUSH-VIEWPORT pushes the viewport to the rear so that it is occluded by any viewports that it overlaps. Neither of these functions changes the position of the viewport on the screen. They only change the visibility relationship between a viewport and other viewports that it overlaps.

2.4.6 Handling User Actions

The default VAX LISP graphics viewport is similar to the VAXstation's VT100 viewport in that the user can modify its size and location. For example, a user can use the pointing device to move a viewport that you create around the screen. If the viewport has a menu icon, the user can use the Window Options menu to push or pop the viewport or change its size.

The VAX LISP graphics system provides several functions that allow you to control and respond to user actions such as these. The VAX LISP graphics system also provides a default response to these actions. You can choose to use the default response, and you can reinstate the default response later should you require your own response for a while.

A user of your program can use the pointer to manipulate a viewport at any point in program execution. For this reason, user actions with regard to viewports must be considered asynchronous activities; that is, you cannot predict when they will happen. You therefore must specify interrupt functions to respond to these actions. The *VAX LISP/VMS System Access Programming Guide* contains information on interrupt functions.

2.4.6.1 **Responding to Viewport Movement** - A user can move the viewport on the display screen by moving the pointer to the viewport border, pressing a button, and moving the outline of the viewport to a new location. By default, the graphics system does nothing to prevent this action and makes no response to it. However, in some applications, movement of one viewport might ruin an important spatial relationship between viewports.

The SET-MOVE-INFO-ACTION function allows you to specify a response to movement of any particular viewport. You specify this response in the form of an interrupt function that is called each time the viewport is moved. This function can take the steps necessary to correct any problems that the viewport movement may have created.

Example 2-1 illustrates a simple use of SET-MOVE-INFO-ACTION. Using the function defined in Example 2-1, you could create a window:

```
(SETF IMMOVABLE-WINDOW
      (CREATE-IMMOVABLE-WINDOW DISPLAY 4.0 4.0))
```

The viewport corresponding to IMMOVABLE-WINDOW would have its lower-left corner at 4.0 centimeters from the bottom and left edges of the screen. If the user attempted to move the viewport, the interrupt function would simply return it to the original position on the screen.

Example 2-1: The Immovable Viewport

```
;; Creates and returns a window object, with the viewport at position
;; X,Y on the screen.  The user cannot move the viewport from that
;; location.

(DEFUN CREATE-IMMOVABLE-WINDOW (DISPLAY X Y)
  (LET* ((WIN (CREATE-WINDOW
                DISPLAY NIL NIL NIL NIL
                :ABSOLUTE-POSITION-X X
                :ABSOLUTE-POSITION-Y Y))
         (IIF-ID (INSTATE-INTERRUPT-FUNCTION
                    #'ABS-MOVE-VIEWPORT
                    :ARGUMENTS (LIST WIN X Y))))
    (SET-MOVE-INFO-ACTION WIN IIF-ID)
    (VALUES WIN IIF-ID)))

(DEFUN ABS-MOVE-VIEWPORT (WINDOW X Y) ; Move viewport to absolute locatio
  (MOVE-VIEWPORT WINDOW
                :ABSOLUTE-POSITION-X X
                :ABSOLUTE-POSITION-Y Y))
```

An informational function, GET-VIEWPORT-POSITION, returns the screen coordinates of a viewport. An interrupt function that you specify with SET-MOVE-INFO-ACTION can use GET-VIEWPORT-POSITION to determine where the viewport has been moved to.

2.4.6.2 **Responding to Viewport Resizing** - The user can choose the "Change the size" entry from the Window Options menu to change the size of a viewport. When the user selects this option, dots appear at the edges of the viewport's picture area at each corner and at the midpoint of each edge. The user then selects one of the dots with the pointer and moves it. Selecting a dot on an edge allows that edge to be moved while the opposite edge remains anchored. Selecting a dot on a corner allows that corner to be moved while the opposite corner remains anchored.

By default, the VAX LISP graphics system responds to viewport resizing by altering the window into the virtual display in a corresponding fashion. For example, if the user selects the top edge of the viewport and moves it up, the top edge of the window will be moved up in the virtual display by an appropriate amount. The bottom edge of the window will remain at the same place in the virtual display. Thus, the scale of what is shown in the viewport does not change, but the viewport now shows a greater amount of information.

You can use the SET-RESIZE-ACTION function to specify a different response to window resizing or to prevent it from happening. SET-RESIZE-ACTION accepts a virtual display (or transformation), a window, and an action as its arguments. The action can be one of the following:

- :DEFAULT or NIL, which requests the default response described above.
- :DISALLOW, which prevents the window from being resized by the user. The Window Options menu for a viewport on which resizing has been disallowed will show the "Change the size" option in grey, indicating that the user cannot change the viewport size.
- An interrupt function identifier (*iif-id*), specifying an interrupt function to be executed when the user attempts to resize the window.

If you specify an interrupt function to execute when a window is resized, the interrupt function receives eight arguments from the graphics system. These are: the new screen coordinates of the lower-left corner of the viewport; the new width and height (in centimeters) of the viewport; and the new location (in world or transformation coordinates) of the lower-left and upper-right corners of the window. The window location is given in the form *x1 y1 x2 y2*. The coordinate system (world or transformation) depends on whether you supplied a virtual display or a transformation as the first argument to SET-RESIZE-ACTION.

If you have specified an interrupt function as the response to window resizing, the graphics system does not automatically resize the viewport and window. If you want the viewport and window resized, you can use the RESIZE-WINDOW function in your interrupt function to perform that action. For example, you could define your interrupt function as follows:

```
(DEFUN RESIZE-ACTION (NEW-SCREEN-X NEW-SCREEN-Y
                      NEW-WIDTH NEW-HEIGHT
                      X1 Y1 X2 Y2
                      WINDOW)
    .
    .        ; Application-specific code
    .
  (RESIZE-WINDOW NIL WINDOW NEW-SCREEN-X NEW-SCREEN-Y
                 NEW-WIDTH NEW-HEIGHT X1 Y1 X2 Y2))
```

In this example, the interrupt function RESIZE-ACTION is invoked when a particular window is resized. The function performs some processing in response to the resizing, then calls RESIZE-WINDOW to actually change the size of the viewport and window. Note that the eight arguments passed to the interrupt function by the graphics system correspond to the third through tenth arguments to RESIZE-WINDOW.

The function RESIZE-ACTION in the example above might be used as follows:

```
        .
        .
    (LET ((RESIZE-IIF (INSTATE-INTERRUPT-FUNCTION
                         #'RESIZE-ACTION
                         :ARGUMENTS (LIST WINDOW))))
      (SET-RESIZE-ACTION DISPLAY WINDOW RESIZE-IIF))
```

2.4.6.3 **Responding to Viewport Deletion** - The Window Options menu for a window that you create has a "Delete" option. For a window you create from LISP, this option is shown in grey, indicating to the user that the window cannot be deleted. You may choose to allow the user to delete the window, and you can specify what action is to be taken should that occur.

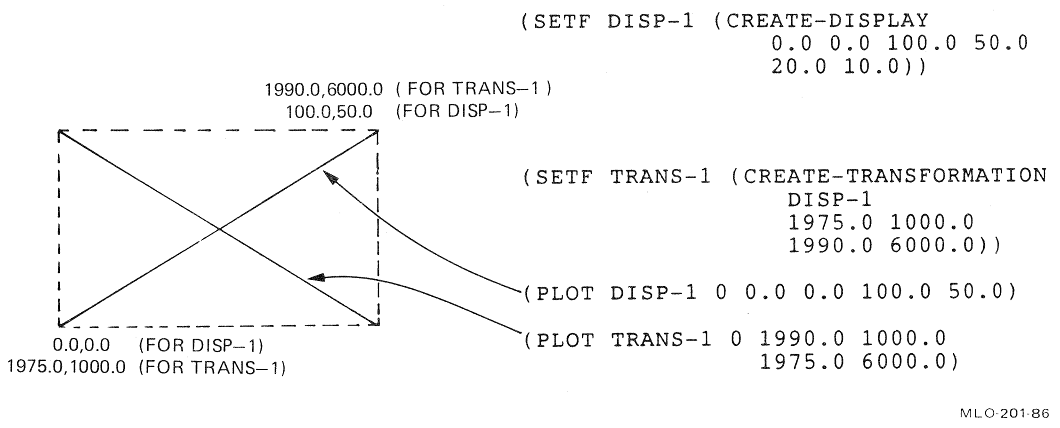
The SET-CLOSE-ACTION function establishes whether or not a particular window can be deleted by the user, and what happens if the user attempts to delete it. The SET-CLOSE-ACTION function takes two arguments, a window and an action. The action can be one of four things:

- :DISALLOW, in which case the user is not allowed to delete the window. This is the default.
- :DELETE, in which case the user can delete the window. The graphics system will call DELETE-WINDOW on the window when the user deletes it.
- :DELETE-DISPLAY, in which case the graphics system will call the DELETE-DISPLAY function on the window's virtual display when the user deletes the window.
- An interrupt function identifier (*iif-id*), specifying an interrupt function to be executed when the user deletes the window. If you supply an *iif-id*, your interrupt function is responsible for taking appropriate action; the graphics system takes no action.

2.5 TRANSFORMATIONS INTO VIRTUAL DISPLAYS

Sometimes it is useful to have more than one coordinate system available in a virtual display. For example, you might want to create a composite graph showing more than one type of information, such as horsepower and torque. To allow you to have multiple coordinate systems in a virtual display, the VAX LISP graphics system provides transformations.

A transformation is a LISP object that you make with the CREATE-TRANSFORMATION function. You specify a virtual display, bounds for a new coordinate system in the display, and, optionally, a rectangle in the existing coordinate system to which you want the new coordinate system mapped. You can then use the resulting transformation object in place of the *display* argument for any function that requires a virtual display. Figure 2-10 illustrates this concept.



```
(SETF DISP-1 (CREATE-DISPLAY
                0.0 0.0 100.0 50.0
                20.0 10.0))
```

```
(SETF TRANS-1 (CREATE-TRANSFORMATION
                 DISP-1
                 1975.0 1000.0
                 1990.0 6000.0))
```

```
(PLOT DISP-1 0 0.0 0.0 100.0 50.0)
```

```
(PLOT TRANS-1 0 1990.0 1000.0
                1975.0 6000.0)
```

Figure 2-10: Transformations

2.5.1 Creating and Accessing Transformations

The CREATE-TRANSFORMATION function creates and returns a transformation object. You can assign the value returned by CREATE-TRANSFORMATION to a symbol:

```
(SETF *TRANS-1* (CREATE-TRANSFORMATION *DISPLAY-1*
                     0.0 0.0 100.0 350.0
                     10.0 0.0 40.0 200.0))
```

You can then use the symbol as an argument to the numerous functions that require a virtual display.

The TRANSFORMATIONP function returns T if the value of its argument is a transformation object, and NIL otherwise.

Each transformation has an identification number assigned by the MicroVMS workstation graphics software. This number is called the transformation ID or tr_id. You can use the UIS-ID function to return this number, should you need it for use with CALL-OUT.

You must retain the value returned by CREATE-TRANSFORMATION; otherwise, you have no means of access to the transformation. You will be unable to draw in it or to delete it.

2.5.2 Deleting a Transformation

Transformations consume system resources. Therefore, you should take care to delete transformations when you are done with them. The DELETE-TRANSFORMATION function deletes a transformation:

```
     (DELETE-TRANSFORMATION *TRANS-1*)
```

The DELETE-TRANSFORMATION function does not delete the display into which a transformation was mapped, nor does it delete or alter any graphic information or windows that were created using the transformation in place of the *display* argument.

CHAPTER 3

GRAPHICS OUTPUT OPERATIONS

This chapter describes the various ways you can cause lines, text, and other graphic information to appear in windows. The major sections in this chapter and the subjects they cover are as follows:

- Section 3.1 describes the display list, the means by which graphic information is stored in a virtual display.
- Section 3.2 describes two families of graphic operations: those you specify in the world coordinate system of a virtual display, and those you specify in the device coordinate system of a viewport.
- Section 3.3 describes the use of attributes and attribute blocks, by means of which you can alter the appearance of graphic information.
- Section 3.4 describes color maps and the use of color.
- Section 3.5 describes functions that draw lines and filled shapes.
- Section 3.6 describes functions that write and position text.
- Section 3.7 describes segments, which allow you to define attribute blocks that are local to a group of operations.
- Section 3.8 describes how to move graphic information in a virtual display and erase it from the display.

3.1 THE DISPLAY LIST

Each virtual display has a display list. When it is enabled, the display list records the graphic operations that have been performed in that display. The VAX LISP graphics software refers to the display list when it needs to create or refresh a window, and for certain operations that affect the contents of a virtual display, such as erasing or moving material within the display.

When you first create a virtual display, its display list is disabled; that is, the display list does not record graphic operations. If you wish to use the display list, you must use the ENABLE-DISPLAY-LIST function to turn the display list on. The display list will then record graphic operations performed in the virtual display until you use the DISABLE-DISPLAY-LIST function to turn the display list off again.

The display list can record only those operations that are performed in a virtual display or in a transformation mapped into a virtual display. Device-coordinate operations -- those whose names end in "-PIXEL" -- work directly in a viewport. These functions never add to the display list.

There are advantages and disadvantages to using a display list. The chief advantage is that operations recorded in the display list can be re-executed (played back) when necessary. Conversely, information present in a viewport but not recorded in the display list will disappear when the display list is re-executed. The display list is completely or partially re-executed by the following functions and user operations:

- CREATE-WINDOW
- MOVE-WINDOW
- RESIZE-WINDOW
- "Change the size" choice on the Window Options menu

Consequently, information that is not recorded in the display list may not appear on the screen following one of these functions or operations. For example, if you draw lines into the virtual display with the display list disabled, and later create a window over those lines, the lines will not appear in the corresponding viewport.

The functions MOVE-AREA and ERASE alter the display list, although they do not re-execute it.

The chief disadvantage of the display list is overhead. It takes time to update the display list and virtual memory to contain it. It also takes time to delete the display list. A virtual display whose display list contains a large amount of information will take appreciably longer to delete than a virtual display with no display list.

A reasonable strategy with regard to the display list is to enable it when you are performing output operations that must survive the functions listed previously, and leave it disabled otherwise. Disabling is especially useful for applications such as animation, where the output operations are plentiful, time-critical, and temporary. It may also be possible in some applications to use device-coordinate operations (that do not update the display list) when temporary graphic information is required. Section 3.2 contains more information on device-coordinate operations.

Information in a display list is encoded in a device-independent fashion; that is, executing the display list will produce the same appearance on any display device supported by the graphics software. However, you do not have any direct access to the display list; your only means of access is through the functions described in this manual.

A display list is automatically deleted when you delete its virtual display.

3.2 WORLD-COORDINATE AND DEVICE-COORDINATE OPERATIONS

The VAX LISP graphics system allows you to specify graphic operations in either of two coordinate systems: the world coordinate system and the device coordinate system. (Sections 2.2.1 and 2.2.2 describe these systems.) For each function that accepts world coordinates to specify an output location, there is a corresponding function that accepts device coordinates. The latter functions have the same name as the former with the suffix "-PIXEL" added; for example, PLOT and PLOT-PIXEL.

Other important distinctions between world-coordinate and device-coordinate operatons are the following:

- **Arguments.** All world-coordinate output functions take a display object as their first argument; device-coordinate operations take a window object.

- **Coordinate data type.** World-coordinate output functions take floating-point arguments to specify coordinates; device-coordinate functions take integers.

- **Display list.** World-coordinate operations place information on a virtual display's display list (if it is enabled; see Section 3.1). This information appears on the screen when you create a window that includes the area in which you placed the information.

  Device-coordinate operations, on the other hand, work directly in a viewport by affecting the viewport's pixels. They never affect the display list.

- **Longevity.** Information created with world-coordinate operations and recorded on the display list survives as long as the virtual display. The only way to eliminate it is with the ERASE function.

Lines and text created with device-coordinate operations can survive only as long as the viewport survives, and may be erased if the corresponding window is moved or resized.

- **Device independence.** World-coordinate operations are inherently device-independent; that is, information stored using world-coordinate operations will appear the same on any supported display device. Device-coordinate operations are inherently device-dependent, since the size and shape of pixels influence the appearance of the output. (See Section 2.2.2 for more information on this distinction.)

- **Speed.** World-coordinate operations must transform their coordinate arguments through several intermediate steps to device-dependent coordinates that the display device can use. Some of these steps use relatively slow floating-point arithmetic. World-coordinate operations must also take time to update the display list if it is enabled.

  Device-coordinate operations need only perform a single translation of their coordinate arguments, using integer arithmetic, before they can be displayed; and they never incur the overhead of updating the display list. Consequently, device-coordinate operations are faster than their world-coordinate counterparts.

- **Accuracy.** The floating-point numbers used in world coordinates are subject to a slight amount of round-off error. Furthermore, floating-point coordinates must ultimately be used to address a pixel, which, by its nature, has integer coordinates. These factors can result in the lines that are finally drawn in the viewport being off by a pixel from the anticipated location.

  Device-coordinate operations, since they use integer coordinates to address pixels directly, perform with complete accuracy up to the resolution limits of the display device.

These comparisons suggest the following guidelines for using world-coordinate operations and device-coordinate operations. Use world-coordinate operations when:

- You need the device independence of the world coordinate system.

- You need the permanence of information recorded in a display list.

- You do not know when or where windows will be present in the virtual display.

- You do not have any way to correlate the world coordinates of your virtual display to the device coordinates of a viewport.
- Speed is not of prime importance.
- Accuracy to the pixel level is not of prime importance.

Use device-coordinate operations when:

- You want to display output in one particular window, rather than in any window mapped into a display.
- The lines and text you wish to output are temporary and need not survive if the window is moved or resized.
- You are not using the world coordinate system.
- Speed is important, as in animation.
- Accuracy to the pixel level is important.

You can intermix world-coordinate and device-coordinate operations freely in your programs, and there are some situations where that is appropriate.

For example, consider a graphics editor program. Typically, such a program allows the user to create a line, rectangle, or circle, and then manipulate it, using the pointer, until satisfied with its size, shape, and location. The user then presses a pointer button to indicate that this particular figure is finished. In writing such a program, you might use device-coordinate operations for all manipulation of the figure (consisting of alternately drawing elements and then erasing them), and then use world-coordinate operations to draw the finished figure and record it on the display list. This scheme would allow quick and efficient figure manipulation, yet still record the final figure permanently.

3.3 ATTRIBUTES AND ATTRIBUTE BLOCKS

All functions that can draw lines or write text take an attribute block specifier as their second argument. The attribute block specifier, an integer from 0 through 255, tells the graphics system which attribute block to use with a particular operation.

An attribute block is a collection of attributes. Each attribute controls some aspect of the visual results of an operation. There is, for example, an attribute that specifies the width of a line; another that specifies the font in which text should be written; a third that specifies whether the results of an operation should be clipped at a certain rectangle.

This section is divided in the following way:

- Section 3.3.1 describes how to modify and use attribute blocks.
- Section 3.3.2 describes the three general categories of attributes, and describes attributes that are common to all output operations in detail.

Sections 3.5 and 3.6 contain descriptions of attributes that relate exclusively to line drawing and text operations, respectively. Section 3.7 describes segments, by means of which you can create temporary attribute blocks.

3.3.1 Modifying and Using Attribute Blocks

If you want to alter the appearance of a graphic output operation -- for example, if you wish to draw a dashed line instead of a solid line -- you must first modify an attribute block so that it contains the desired attribute value, and then give that attribute block as the second argument to the function that draws the line. Once you have modified the attribute block, you can reuse it for any other operation in that virtual display.

The VAX LISP graphics system provides a default attribute block, numbered 0, when you create a virtual display. This default attribute block is the same for all virtual displays. You can read from this attribute block, but you cannot alter it.

To modify an attribute block, you use the SET-ATTRIBUTE function. This function takes as arguments a virtual display, an input attribute block, an output attribute block that is to be modified, an attribute specifier, and a new value for the specified attribute. The action of this function is to modify the output attribute block so that it is a copy of the input attribute block, differing only in the value of the attribute you specify. Thus, the output attribute block inherits all the values of the input attribute block, with the exception of the attribute you specify. You can change the value of only one attribute with each call to SET-ATTRIBUTE.

When you create a virtual display, all 256 of the display's attribute blocks are initialized to the values contained in attribute block 0, the default attribute block. Once the virtual display has been created, you can modifiy the contents of any of the attribute blocks execpt attribute block 0. It is convenient to use attribute block 0 as the basis for most of the attribute blocks you create, because you can be sure of its contents.

The following example illustrates how you would modify an attribute block so that it causes lines to be dashed. Figure 3-1 illustrates the process schematically.

```
     (SET-ATTRIBUTE *DEMO-DISPLAY*
                    0              ; Input attribute block
                    1              ; Output attribute block
                    :LINE-STYLE :DASHED)  ; Attribute and value
```

Following the execution of this function, any line-drawing function given attribute block 1 as its second argument draws dashed lines instead of solid lines.

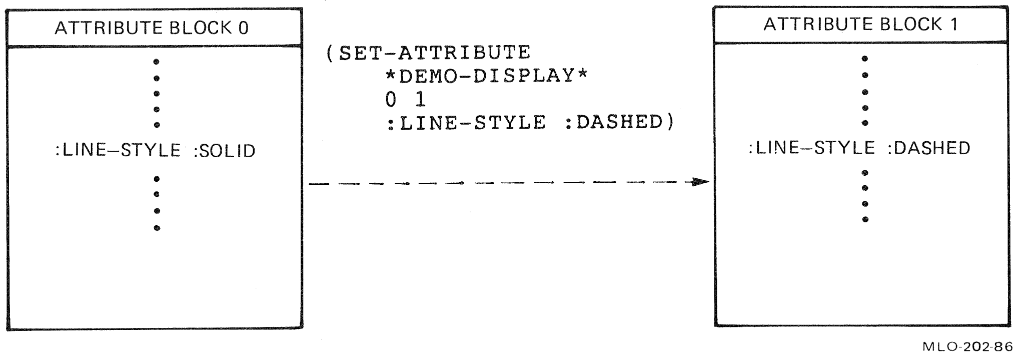


Figure 3-1: Modifying an Attribute Block

Since you can only change one attribute value per invocation of SET-ATTRIBUTE, you must make repeated calls to SET-ATTRIBUTE if you want to create an attribute block that differs from attribute block 0 in more than one attribute value. The following example modifies an attribute block to specify dashed lines, a new line width, and a method of closing arcs:

```
     (SET-ATTRIBUTE *DEMO-DISPLAY* 0 2 :LINE-STYLE :DASHED)
     (SET-ATTRIBUTE *DEMO-DISPLAY* 2 2 :LINE-WIDTH 2.0)
     (SET-ATTRIBUTE *DEMO-DISPLAY* 2 2 :ARC-TYPE :CHORD)
```

The first call to SET-ATTRIBUTE uses attribute block 0 as input and modifies attribute block 2. The last two calls to SET-ATTRIBUTE use the same block, 2, for both input and output. The result of the three calls to SET-ATTRIBUTE is an attribute block that differs from attribute block 0 in three values.

Section 3.3.2 lists the default attributes contained in attribute block 0.

Two attributes with the suffix -PIXEL only have an effect during a device-coordinate operation. You set them by using the SET-ATTRIBUTE function, but at least one window must be mapped into the display you specify with SET-ATTRIBUTE. The values that you establish for the

-PIXEL attributes affect operations in any window that into the virtual display, including windows created after the attribute is set.

You can retrieve the value of an attribute in a particular attribute block with the GET-ATTRIBUTE function. The GET-ATTRIBUTE function takes a virtual display as its first argument, an attribute block as its second, and an attribute as its third. It returns the value of the specified attribute. If you want the value of one of the -PIXEL attributes, there must be a window mapped into the display.

The GET-ATTRIBUTE-LIST function returns attribute values from an attribute block in the form of a property list.

3.3.2 Attributes

Attributes fall into three general categories:

- Those that only affect line drawing. These are described in Section 3.5.3.
- Those that only affect text operations. These are described in Section 3.6.3.
- Those that affect both line and text operations, or the virtual display as a whole. These are described in this section.

Table 3-1 lists all the attributes. The subsections that follow describe the attributes that affect both line and text operations.

3.3.2.1 **:BACKGROUND-INDEX** - The :BACKGROUND-INDEX attribute controls the color that is used for background parts of a graphic object. Examples of background parts are the spaces between the dots in a dotted line, the area surrounding and contained by letters in a text string, and the parts of a fill pattern that are not normally visible against the background. The value of the :BACKGROUND-INDEX attribute is an integer that is an index into a color map (see Section 3.4). On a monochrome (non-grey-scale) workstation, the value must be 0 or 1, with 0 the default.

The effect of the :BACKGROUND-INDEX attribute depends on the setting of the :WRITING-MODE attribute. With the default writing mode (:OVERLAY), background parts of graphic objects are not written to the screen; therefore, the color specified by the :BACKGROUND-INDEX attribute does not appear. With other writing modes (such as :REPLACE), the background parts are written to the screen. See Section 3.3.2.5 for information on writing modes.

Table 3-1: Attributes

Name	Possible Values	Default	Cat.	Description
:ARC-TYPE	:CHORD :OPEN :PIE	:OPEN	Line	Specifies whether an arc is left open, closed by a chord, or made into a pie segment
:BACKGROUND-INDEX	0 or 1 (for monochrome system)	0	Both	Designates an entry in the display's color map to use for background parts of output
:CHARACTER-SPACING	List of two single floats	(0.0 0.0)	Text	Specifies extra space between text characters and text lines, respectively, as proportions of the character width and height
:CLIP	NIL or list of four single floats	NIL	Both	Specifies whether output is limited to a portion of a display and, if it is, gives a world-coordinate rectangle to clip to
:CLIP-PIXEL	NIL or list of four fixnums	NIL	Both	Specifies whether output is limited to a portion of a window and, if it is, gives a device-coordinate rectangle to clip to
:FILL-PATTERN	NIL or a keyword shown by SHOW-FILL-PATTERNS function	NIL	Line	Specifies a pattern with which to fill drawn figures; the :FONT attribute for this block must specify the font file UIS$FILL_PATTERNS
:FONT	Pathname, character string specifying a font file or logical name pointing to a font file, or keyword-value list	NIL	Text	Specifies the font to use for text operations
:LEFT-MARGIN	Single float	X coordinate of left edge of virtual display	Text	Specifies the left margin for text operations, in world coordinates
:LEFT-MARGIN-PIXEL	Fixnum	0	Text	Specifies the left margin for text operations in windows, in device coordinates
:LINE-STYLE	:DASHED :DASHED-DOTTED :DOTTED :SOLID bit vector	:SOLID	Line	Specifies how lines are drawn
:LINE-WIDTH	Single float or list in form (n :WORLD-COORDINATES)	1.0	Line	Specifies the width of a line as a multiple of the default line width, or in world-coordinate units
:WRITING-INDEX	0 or 1 (for monochrome system)	1	Both	Designates an entry in the display's color map to use for writing (foreground) parts of output

(Continued on next page)

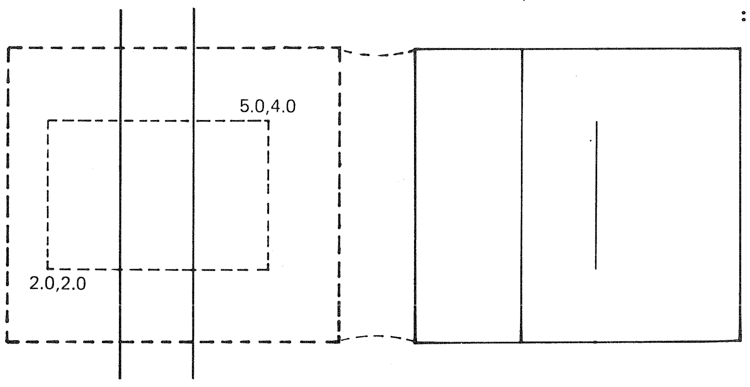
Table 3-1 (cont.)

Name	Possible Values	Default	Cat.	Description
:WRITING-MODE	:COMPLEMENT :ERASE :ERASE-NEGATE :OVERLAY :OVERLAY-NEGATE :REPLACE :REPLACE-NEGATE :TRANSPARENT :COPY	:OVERLAY	Both	Specifies how new graphical output interacts with background and existing output

3.3.2.2 **:CLIP** - The :CLIP attribute controls whether or not graphic output is truncated at the boundaries of a rectangle defined in the virtual display. By default, output is clipped only by the edges of a viewport, not in the virtual display.

You can create an attribute block whose :CLIP value is a list of four floating-point numbers which define two opposite corners of a world-coordinate rectangle. Any graphic operations that use this attribute block restrict the output to the inside of this rectangle. Note, however, that this clipping rectangle only affects operations that use this particular attribute block. Simply defining a clipping rectangle does not cause the results of previous operations to be clipped, nor does it affect operations that use an attribute block with no defined clipping rectangle.

Figure 3-2 illustrates the use of the :CLIP attribute.



```
(SET-ATTRIBUTE *DISPLAY* 0 2
               :CLIP '(2.0 2.0 5.0 4.0))

(PLOT *DISPLAY* 0
      3.0 0.5 3.0 5.5)   ; NOT CLIPPED

(PLOT *DISPLAY* 2
      4.0 0.5 4.0 5.5)   ; CLIPPED
```

MLO-203-86

Figure 3-2: Clipping

3.3.2.3 **:CLIP-PIXEL** - The :CLIP-PIXEL attribute controls whether or not graphic output is clipped (that is, truncated) at the boundaries of a rectangle defined in the viewport. By default, output is clipped only by the edges of a viewport.

You can create an attribute block whose :CLIP-PIXEL value is a list of four integers which define two opposite corners of a device-coordinate rectangle. Graphic operations that use this attribute block will clip the output at the edges of this rectangle. Note, however, that this clipping rectangle only affects operations that use this particular attribute block. Simply defining a clipping rectangle does not cause the results of previous operations to be clipped, nor does it affect operations that use an attribute block with no defined clipping rectangle.

3.3.2.4 **:WRITING-INDEX** - The :WRITING-INDEX attribute controls the color that is used for the writing parts of a graphic object. Examples of writing parts are lines, the actual letters in a text string, and the parts of a fill pattern that are normally visible against the background.

The value of the :WRITING-INDEX attribute is an integer that is an index into a color map (see Section 3.4). On a monochrome (non-grey-scale) workstation, the value must be 0 or 1, with 1 the default.

3.3.2.5 **:WRITING-MODE** - The :WRITING-MODE attribute specifies how lines, fill patterns, text, and bitmap images are displayed on the screen if they overlap with information already on the screen. This section describes the attribute and its values in general terms. Later sections describe its specific effects on lines, fill patterns, text, and bitmap arrays.

Every graphic object has a writing part, the part that you normally see. It also has a background part that you normally do not see, but that is part of the object nonetheless. For example:

- The writing part of a dotted line consists of the dots. The background part is the space between the dots.
- The writing part of a string of text consists of the actual text characters. The background part consists of the area inside and around the characters, called the character cell box.
- The writing part of a fill pattern consists of the lines or dots that you normally see. The background part consists of the spaces between the lines or dots.

- The writing part of a screen image represented as a bitmap array consists of those pixels for which the corresponding element in the array is nonzero. The background part consists of those pixels for which the corresponding array element is zero. (See Chapter 4 for information about screen images and bitmap arrays.)

A graphic object, consisting of writing parts and background parts, is placed on the workstation screen by some function. Before the operation, any particular point of the screen can contain either the writing color or the background color. The writing color is the color in which lines and text are normally drawn; the background color is the color of an empty viewport. For a monochrome display screen with a light background, the writing color is black and the background color is white.

The graphic operation deposits the writing color and/or the background color on the screen to represent the object. (You can change the writing and background color by changing the values of the :WRITING-INDEX and :BACKGROUND-INDEX attributes, respectively.) The value of the :WRITING-MODE attribute for the operation dictates what happens to pixels that correspond to writing and background parts of the output. The following list describes each of the nine values of the :WRITING-MODE attribute:

- :OVERLAY - Writes the writing color onto the screen to represent the writing parts. Background parts are not affected; that is, the existing background shows through in background parts. This is the default.

- :OVERLAY-NEGATE - Writes the background color onto the screen to represent the writing parts. Background areas are not affected.

- :COMPLEMENT - In writing parts of the object, writes the writing color into areas currently occupied by the background color, and the background color into areas currently occupied by the writing color. Background parts are not affected.

- :REPLACE - Writes the writing color onto the screen to represent writing parts, and writes the background color into background parts of the output.

- :REPLACE-NEGATE - Writes the background color onto the screen to represent writing parts, and writes the writing color into background parts of the output.

- :ERASE - Writes the background color onto the screen in both writing and background parts of the output.

- :ERASE-NEGATE - Writes the writing color onto the screen in both writing and background parts of the output.

- :TRANSPARENT - Has no effect on the screen. Operations that use this attribute do, however, update the display list and the text position.

- :COPY - Copies the bitmap representation of the object directly to the screen without regard to writing color or background color. This writing mode is primarily useful for creating screen images from bitmap arrays (see Chapter 4).

Table 3-2 shows how screen pixels are affected by output operations using each of the writing modes. The table shows the result when a pixel of either the writing color or the background color is overlaid by either the writing part or the backaround part of the output.

Table 3-2: Writing Modes

Output pixel	W	W	B	B
Screen pixel	B	W	W	B
Mode				
:OVERLAY	W	W	W	B
:OVERLAY-NEGATE	B	B	W	B
:REPLACE	W	W	B	B
:REPLACE-NEGATE	B	B	W	W
:ERASE	B	B	B	B
:ERASE-NEGATE	W	W	W	W
:COMPLEMENT	W	B	W	B
:TRANSPARENT	B	W	W	B
:COPY	N/A	N/A	N/A	N/A

Note: W writing B background

Figure 3-3 illustrates the effect of the various writing modes for line, fill pattern, and text output.

3.4 COLOR

A workstation screen can simultaneously display some number of colors. For a monochrome system, the number is two: black and white. Color systems may be able to display 16, 256, or more colors simultaneously.

You specify color in an output operation by means of two attributes, :WRITING-INDEX and :BACKGROUND-INDEX. The values for these attributes are integers that are indexes into a color map. The color map is a table that contains the description of a color (or an equivalent grey-scale intensity level) in each entry. The number of entries is equal to the number of colors that the system can simultaneously display. Figure 3-4 shows how an output operation works through an attribute block and a color map to produce an image on the screen.

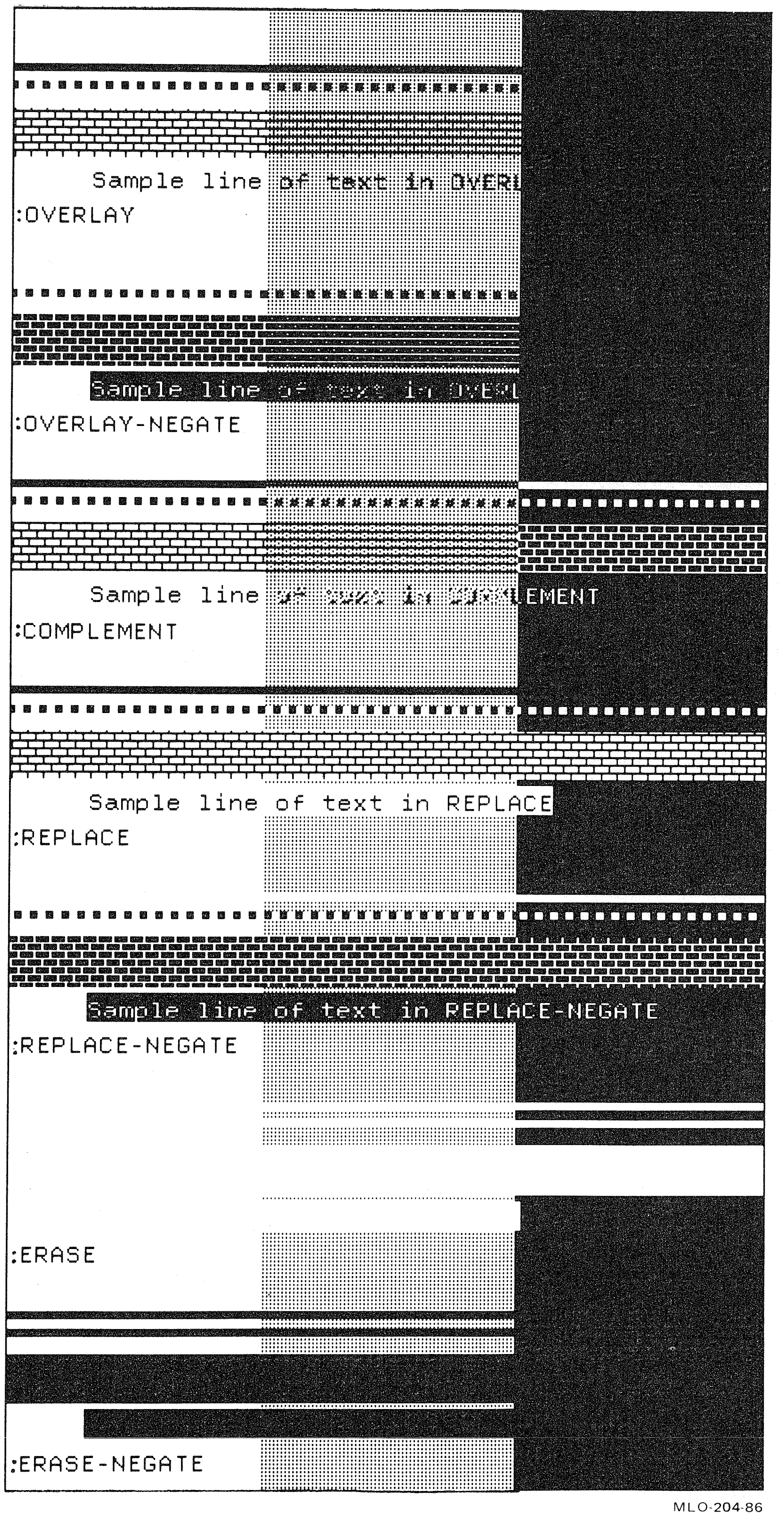

Sample line of text in OVERL
:OVERLAY
Sample line of text in OVERL
:OVERLAY-NEGATE
Sample line of text in COMPLEMENT
:COMPLEMENT
Sample line of text in REPLACE
:REPLACE
Sample line of text in REPLACE-NEGATE
:REPLACE-NEGATE
:ERASE
:ERASE-NEGATE
MLO-204-86

Figure 3-3: Writing Modes

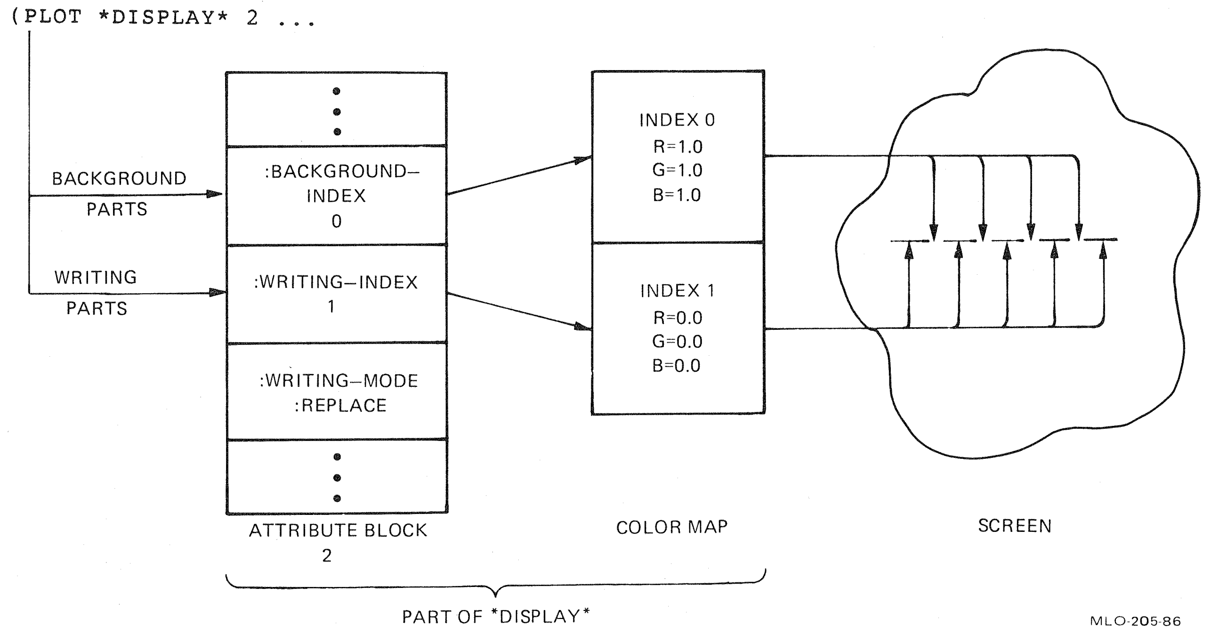


Figure 3-4: Writing Through the Color Map

A workstation system comes with a standard color map built in. Two functions, GET-WS-COLOR and GET-WS-INTENSITY, return information about this color map. Each function takes as arguments a display and an index into the color map. GET-WS-COLOR returns three values corresponding to the red, green, and blue components of the color in that entry. GET-WS-INTENSITY returns a single floating-point number that is the equivalent grey-scale intensity for the entry.

Both GET-WS-COLOR and GET-WS-INTENSITY take an optional window argument. When you supply this argument, the return values reflect the actual realized color or intensity for the specific device on which the window was created. This difference reflects the fact that display devices cannot always reproduce exactly the color or grey-scale intensity that is specified in the color map.

For example, imagine that an entry in a color map has an intensity value of 0.7. If a window is created on a monochrome (non-grey-scale) monitor, an object written to the window through this entry cannot actually be displayed in the shade of grey requested. The display device makes its best approximation, which is 1.0, or white. Using GET-WS-INTENSITY with this window specified returns a value of 1.0, even though the value in the color map is 0.7.

Each virtual display has its own color map, which controls the colors of objects in that display. Initially, a display's color map is identical to the workstation standard color map. You can change entries in the display's color map by means of the SET-COLOR and SET-INTENSITY functions. Each takes a virtual display and an index into the color map as its first two arguments.

You specify a color with SET-COLOR by means of three additional arguments, one each for the red, green, and blue components of the color. These values are floating-point numbers between 0.0 and 1.0, inclusive.

SET-INTENSITY takes only one additional argument, the value of the equivalent grey-scale intensity. This is a floating-point number between 0.0 and 1.0. SET-INTENSITY sets the three colors in the color map entry to produce a grey-scale level corresponding to the value you specify.

When you change an entry in the color map, there is no immediate change on the screen. Objects drawn using that entry after the change has been made will show the effect of the new value. Also, anything that causes the display list to be re-executed (such as resizing the viewport) will change the appearance of objects drawn using an altered color map entry, whether they were drawn before or after the change was made to the color map.

The background and parts of the borders of a viewport are drawn using values stored in the color map. Therefore, alterations to entries 0 and 1 in a virtual display's color map can alter the appearance of viewports mapped to the display. The changes do not become apparent in existing windows unless the display list is re-executed.

You can get information about a display's color map with the GET-COLOR and GET-INTENSITY functions. These are equivalent to GET-WS-COLOR and GET-WS-INTENSITY, except that they return information about the display's color map rather than the standard color map. Both functions take an optional window argument. Without the window argument, they return values from the color map. With the window argument, they return the realized values from the device on which the window was created.

3.5 DRAWING LINES AND SHAPES

This section describes functions and related attributes that allow you to draw lines and filled shapes in virtual displays and viewports. The section is organized as follows:

- Section 3.5.1 describes functions that draw points, lines, and series of connected lines.
- Section 3.5.2 describes functions that draw circles, ellipses, and arcs.
- Section 3.5.3 describes attributes that are used with these functions.

3.5.1 Points and Lines

The PLOT family of functions draws single points, lines between two points, and two or more connected lines. There are four PLOT functions:

- PLOT draws a point, a line, or up to 124 connected lines in a virtual display.

- PLOT-PIXEL draws a point, a line, or up to 124 connected lines in a viewport.

- PLOT-ARRAY draws up to 65,534 connected lines in a virtual display, taking coordinates from two vectors of single-float numbers.

- PLOT-ARRAY-PIXEL draws up to 65,534 connected lines in a viewport, taking coordinates from two vectors of integers.

The basic difference between the PLOT functions and the PLOT-ARRAY functions is in the way you specify coordinates to them. The PLOT functions accept coordinates as pairs of arguments. The first two coordinate arguments are required; if you give just these arguments, the PLOT functions plot a single point at the specified location. You can supply up to 124 additional coordinate pairs. The following example illustrates the PLOT function:

```
(PLOT *DISPLAY* 0 1.0 1.0)           ; Plot a point
(PLOT *DISPLAY* 0 1.0 1.0 3.0 1.0)   ; Plot a horizontal line
(PLOT *DISPLAY* 0 1.0 1.0 3.0 1.0
      3.0 2.0)                       ; Plot two lines
(PLOT *DISPLAY* 0 1.0 1.0 3.0 1.0
      3.0 2.0 1.0 2.0 1.0 1.0)       ; Plot a rectangle
```

The PLOT-ARRAY functions accept coordinate arguments in the form of two specialized one-dimensional arrays. For PLOT-ARRAY, the elements must be of type SINGLE-FLOAT; for PLOT-ARRAY-PIXEL, the elements must be of type (SIGNED-BYTE 32). The elements of the first vector represent the X coordinates for each point; the elements of the second vector represent the Y coordinates. Thus, to duplicate the rectangle drawn by the final PLOT function in the preceding example, you could use this function:

```
(PLOT-ARRAY
  *DISPLAY* 0
  (MAKE-ARRAY 5 :ELEMENT-TYPE 'SINGLE-FLOAT
              :INITIAL-CONTENTS '(1.0 3.0 3.0 1.0 1.0))
  (MAKE-ARRAY 5 :ELEMENT-TYPE 'SINGLE-FLOAT
              :INITIAL-CONTENTS '(1.0 1.0 2.0 2.0 1.0)))
```

PLOT-ARRAY matches up elements from the first vector with elements from the second vector to form each coordinate pair in succession.

The PLOT functions and PLOT-ARRAY functions each have strengths and weaknesses; thus, PLOT may be appropriate in situations where PLOT-ARRAY is not, and vice versa. The following suggests when to use one or the other:

- PLOT is more convenient for casual line drawing and in situations where a program draws an indeterminate number of lines. For example, a recursively-defined function would use PLOT to draw one line at a time, since it cannot know in advance how many lines to draw or where to draw them.

- PLOT-ARRAY may be more convenient and efficient for drawing a series of similar figures, since the array arguments can be reused. You can define functions that operate on the vector elements to shift or resize the figures they represent.

- The limited number of lines that PLOT can draw with one call puts a limit on the complexity of a filled shape that you can create using PLOT.

Note that there is no concept of a "current position" when you use the PLOT functions. That is, you must explicitly specify the beginning location for each operation; the graphics system does not "remember" where the last point was plotted. (Text output does have a "current position.") Programs that need to retain the position of the last point plotted must do so explicitly.

Depending on the attributes you specify, lines drawn by the PLOT functions may be solid, dotted, dashed, or dashed-dotted. (See Section 3.5.3.3.) You can also specify that lines be heavier than normal. (See Section 3.5.3.4.)

If an attribute block specifies a fill pattern (see Section 3.5.3.2), PLOT does not draw lines at all. Instead, it fills in the shape that the lines would have defined had they been drawn. To draw a filled shape with an outline, call PLOT twice with identical coordinate arguments but different attribute blocks. One attribute block specifies a fill pattern and thus draws the filling; the other does not and thus draws the outline.

To fill in a shape drawn with the PLOT or PLOT-ARRAY functions, the graphics system first makes an imaginary line between the first point that was plotted and the last point that was plotted, unless they coincide. Then, any area(s) enclosed by the lines drawn by the function and the imaginary line between the beginning and end points is filled. Figure 3-5 illustrates three shapes that have been drawn with lines, then filled. All of these figures illustrate the imaginary line between the beginning and end points.

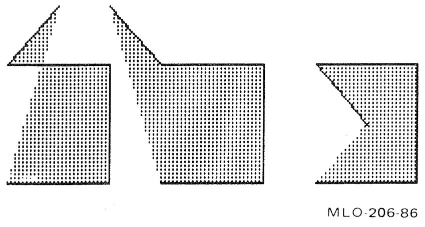


Figure 3-5: Filling Plotted Shapes

3.5.2 Circles, Ellipses, and Arcs

Four functions allow you to draw circles, ellipses, and arcs. You can use attributes with these functions to create pie segments and filled circles and wedges.

- CIRCLE draws a circle or a portion of a circle in a virtual display.
- CIRCLE-PIXEL draws a circle or a portion of a circle in a viewport.
- ELLIPSE draws an ellipse or a portion of an ellipse in a virtual display.
- ELLIPSE-PIXEL draws an ellipse or a portion of an ellipse in a viewport.

These functions are very similar. Each one takes the coordinates of a center position. The CIRCLE functions take a single radius argument, whereas the ELLIPSE functions take a horizontal radius and a vertical radius.

All four of the functions take optional arguments that specify starting and ending positions in radians. These arguments, if given, define an arc which starts at the starting position and proceeds counterclockwise to the ending position. If the starting position argument is NIL or omitted, it defaults to 0.0. If the ending position argument is NIL or omitted, it defaults to 2*PI. Figure 3-6 illustrates this angular coordinate system and shows several examples of arcs.

You can use the :ARC-TYPE attribute (see Section 3.5.3.1) to specify how an arc should be closed. If you specify :OPEN (the default), the arc is left open. You can also use :PIE to draw a pie segment, or :CHORD to draw a line between the endpoints of the arc.

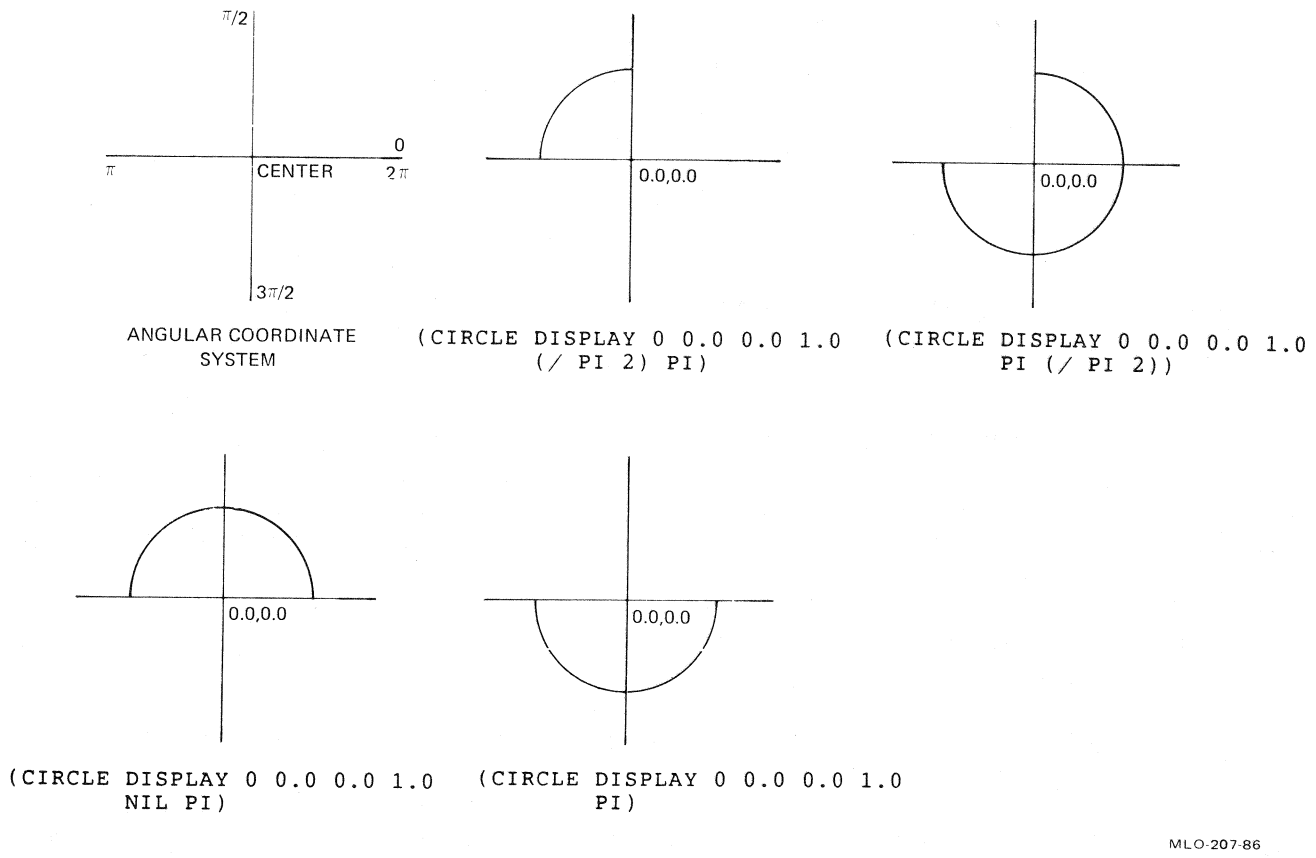

π/2
0
π
CENTER
2π
3π/2
ANGULAR COORDINATE SYSTEM
0.0,0.0
(CIRCLE DISPLAY 0 0.0 0.0 1.0 (/ PI 2) PI)
0.0,0.0
(CIRCLE DISPLAY 0 0.0 0.0 1.0 PI (/ PI 2))
0.0,0.0
(CIRCLE DISPLAY 0 0.0 0.0 1.0 NIL PI)
0.0,0.0
(CIRCLE DISPLAY 0 0.0 0.0 1.0 PI)
MLO-207-86

Figure 3-6: Drawing Arcs

If you have specified a fill pattern (see Section 3.5.3.2), the lines that make up the circle, ellipse, or arc are not drawn. The shape is filled in only if it is a complete circle or ellipse, or if the arc type is :PIE or :CHORD. This means that if you draw an arc with an attribute block that has the default arc type (:OPEN) and a fill pattern in effect, nothing appears on the screen.

3.5.3 Attributes Used with Line-Drawing Functions

The attributes described in this section are useful only with the functions that draw lines or create shapes. Their values are ignored by other output functions.

3.5.3.1 **:ARC-TYPE** - The :ARC-TYPE attribute specifies how a portion of a circle or an ellipse should be closed. There are three possible values:

- :OPEN, specifying that the arc not be closed at all. This is the default value.
- :CHORD, specifying that the endpoints of the arc be joined by a straight line.
- :PIE, specifying that straight lines be drawn from the endpoints of the arc to the center of the circle or ellipse.

An arc that is drawn with the :OPEN attribute cannot be filled. Arcs drawn with the :CHORD or :PIE attribute are filled in the area enclosed by the arc and its closure.

3.5.3.2 **:FILL-PATTERN** - The :FILL-PATTERN attribute specifies two things: first, that figures be filled, and second, the pattern with which to fill them. The values you can give for :FILL-PATTERN are either NIL (requesting the default behavior, which is that figures are not filled), or any of a number of keywords representing specific fill patterns. The SHOW-FILL-PATTERNS function displays all the fill patterns available and the keyword that corresponds to each one.

By default, figures are drawn with lines and are not filled. If you specify a fill pattern keyword for the :FILL-PATTERN attribute, you first of all request that figures not be drawn with lines. Instead, the shapes specified by output operations are filled with the pattern you specify.

To be filled, a figure must meet the following criteria:

- It must be created with a single function call. For example, a rectangle that you draw with a single call to PLOT can be filled, but an identical rectangle drawn with four separate calls to PLOT cannot be.
- If it is an arc, it must be drawn with an :ARC-TYPE attribute of :CHORD or :PIE. (Full circles and ellipses are filled regardless of the :ARC-TYPE value.)

Figures drawn with any of the PLOT functions are filled by making an imaginary line between the first and last points plotted, then filling any area(s) enclosed by the lines that the function created and the imaginary line.

To modify an attribute block to result in a fill pattern, you must specify both the :FONT and the :FILL-PATTERN attributes. Fill patterns are stored as character glyphs in a font file. The system logical name UIS$FILL_PATTERNS points to this file. Therefore, modifying an attribute block to specify a fill pattern is a two-step process:

1. Create an attribute block whose :FONT attribute is the string "UIS$FILL_PATTERNS".

2. Modify the same attribute block so that its :FILL-PATTERN attribute is a keyword associated with one of the patterns.

For example:

```
(SET-ATTRIBUTE *DISPLAY* 0 1 :FONT "UIS$FILL_PATTERNS")
(SET-ATTRIBUTE *DISPLAY* 1 1 :FILL-PATTERN :GRID4)
```

Figures drawn with attribute block 1 after execution of these two functions will be filled with a grid pattern. Note, however, that you can no longer use attribute block 1 to write text, because its :FONT attribute does not point to a font file that contains legible characters. (See Section 3.6.3.2 for information on the :FONT attribute.)

You can turn off filling and reset line drawing in an attribute block by specifying a value of NIL for the :FILL-PATTERN attribute.

3.5.3.3 **:LINE-STYLE** - The :LINE-STYLE attribute specifies the appearance of lines drawn on the screen. It can have one of four keyword values: :SOLID, :DOTTED, :DASHED, and :DASHED-DOTTED. :SOLID is the default. Figure 3-7 illustrates the appearance of each of the standard line styles.

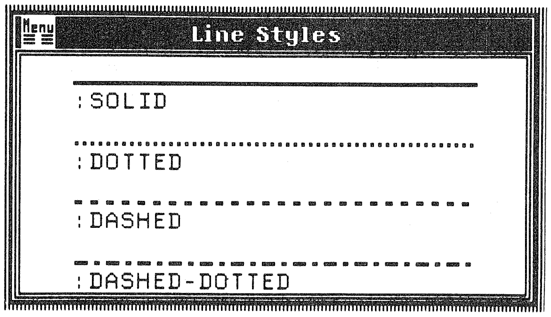


Figure 3-7: Line Styles

You can also supply the value of :LINE-STYLE as a bit vector, using bits with values of 1 and 0 to specify the writing color and background color, respectively. The bit vector must be 32 or fewer bits in length. If it is less than 32 bits long, SET-ATTRIBUTE fills it out to a length of 32 by replicating it. For example:

```
(SET-ATTRIBUTE *DISPLAY* 0 1 :LINE-STYLE #*11111100)
```

Since the length of this bit vector divides exactly into 32, it will be replicated exactly four times and will produce an even dashed line. If the length of the bit vector does not exactly divide into 32, it will be only partially replicated at the end and will not produce an even pattern.

3.5.3.4 **:LINE-WIDTH** - The :LINE-WIDTH attribute specifies the width (weight) of lines drawn on the screen. Its value is normally a floating-point number that expresses line width as a multiple of the default value, which is 1.0. For example, to draw a line three times heavier than normal, use an attribute block whose :LINE-WIDTH value is 3.0.

You can also supply a value for :LINE-WIDTH in the form '(*n* :WORLD-COORDINATES), where *n* is a floating-point number specifying the width of the line in world-coordinate units. Line width is subject to scaling when a line drawn with this attribute block is displayed in a viewport.

The GET-ATTRIBUTE function used with :LINE-WIDTH returns a floating-point number representing the line width, unless you have specified world coordinate line width. In this case, GET-ATTRIBUTE returns the list (*n* :WORLD-COORDINATES).

3.6 TEXT OPERATIONS

The VAX LISP graphics system provides functions that write text into a virtual display or viewport, and that allow precise control over the appearance and position of the text on the screen. This section describes these facilities. The section is divided as follows:

- Section 3.6.1 shows how to write text strings to particular locations in the display or viewport and describes the properties of text and text operations.
- Section 3.6.2 describes functions that allow more flexible positioning of text and measurement of text strings.
- Section 3.6.3 describes the various attributes that affect text.

3.6.1 Writing Text

Two functions, TEXT and TEXT-PIXEL, write text. TEXT writes text into a virtual display; TEXT-PIXEL writes text directly into a viewport. Optional arguments to both functions allow you to specify the position of the text in the display or viewport. Attribute values in the attribute block you specify with the TEXT functions affect the appearance of the text.

The first argument to TEXT is a virtual display. The TEXT function places the text in the specified virtual display and updates the display list, if the display list is enabled.

The first argument to TEXT-PIXEL is a window. The TEXT-PIXEL function places the text in the viewport associated with the window. As with the other device-coordinate functions, TEXT-PIXEL does not update the display list.

The second argument to both TEXT and TEXT-PIXEL is an attribute block. The values of the following attributes affect the appearance of the text:

- :FONT (see Section 3.6.3) determines the size and style of the text
- :WRITING-MODE (see Section 3.3.2.5) determines how the characters of the text are written on the screen
- :CHARACTER-SPACING (see Section 3.6.3) indicates whether extra space is added between characters and lines

A fourth attribute, :LEFT-MARGIN (and its device-coordinate counterpart, :LEFT-MARGIN-PIXEL), sets up a left margin for text operations. This attribute is discussed in Section 3.6.2.

The third argument to each of the TEXT functions is the text that you wish to write, in the form of a single character or a character string. The string can be a literal quoted string:

```
(TEXT *DISPLAY* 0 "This is your life!")
```

Or, you can use any form that evaluates to a character string:

```
(TEXT *DISP* 0 (FORMAT NIL "This is your life, ~A" NAME))
```

There are several ways to position text in the display or viewport. The TEXT functions take two optional arguments that specify the coordinates of the beginning position for the text. If you do not supply these arguments, the output begins at the end of the last text that was output. So, for example, the two function calls in the following example would output a single line of text:

```
(TEXT *DISP* 0 "Mary had a " 0.0 3.0)
(TEXT *DISP* 0 "little lamb.")
```

The first function call begins a line of text at (0.0,3.0) in world coordinate space. The second function call appends its text to the end of the line created by the first. Any subsequent call to TEXT will append text to this line, unless a different position is specified explicitly in the call to TEXT or the current text position is altered in some other way. Section 3.6.2 describes text positioning in more detail.

If you specify a position with TEXT, you are indicating where the upper left corner of the first character should be placed (the "aligned position"). There are other ways of specifying position; they are described in Section 3.6.2.

The appearance of the text on the screen depends on two attributes: the font and the writing mode. A font is a particular size and style of type. The VAX LISP graphics system provides a number of fonts; they differ from the default font in both size and style. Section 3.6.3.2 describes how to use the :FONT attribute.

Text of a given font always appears the same size on the screen, even if scaling has taken place between the window and the viewport. For example, if you have a default window and viewport displaying some lines and text, and another window and viewport that magnify a portion of the display, text will be the same size in both viewports, although the lines are magnified in the second. Figure 3-8 illustrates this.

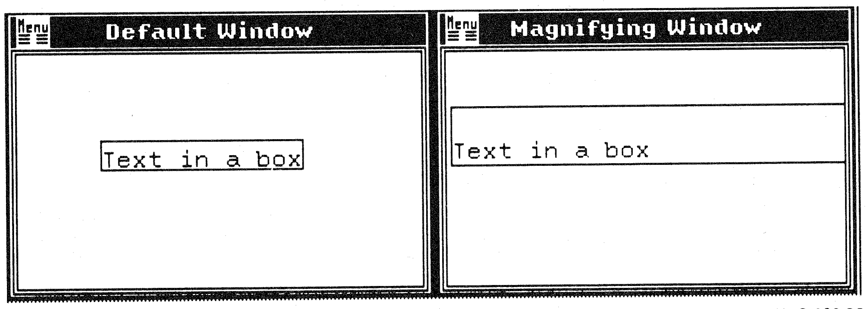


Figure 3-8: Text and Scaling

The writing mode attribute (described in Section 3.3.2.2) has four values that are particularly useful for text. They are :REPLACE, :REPLACE-NEGATE, :ERASE, and :ERASE-NEGATE. Each text character has a character cell around it; an imaginary rectangle that outlines it. For a line of text, the character cells form a long horizontal box. The :REPLACE writing mode causes the entire character cell box to be placed on the screen, not just the characters themselves. If the text is entirely over the background, there is no difference between the :REPLACE and :OVERLAY writing modes; either will result in writing-color letters on a featureless background. If, however, the text crosses a line or an area of fill, the background-color character-cell box provided by :REPLACE guarantees that the text will be visible. The :REPLACE-NEGATE writing mode is similar, except that it writes background-color text in a writing-color box.

The :ERASE and :ERASE-NEGATE writing modes are similar to :REPLACE and :REPLACE-NEGATE, except that they write only the character-cell box that the text would occupy; they do not write text. You can use them to erase a line of text that you previously wrote.

Figure 3-3 illustrates the effect of the various writing modes with text. Note that :OVERLAY can cause text to disappear against a writing-color background. The writing modes :OVERLAY, :OVERLAY-NEGATE,
and :COMPLEMENT do not work well over a textured background.

3.6.2 Positioning and Measuring Text

A number of VAX LISP graphics functions allow you to position text precisely within a virtual display or viewport. This section describes how to use them.

3.6.2.1 **The Text Position and Text Reference Points** - Each virtual display maintains a text position. This is the position in the virtual display at which the next text output operation will begin. A separate device-coordinate text position is also maintained for all windows into a virtual display.

To position text on the screen, you must know the various ways in which you can affect the text position. You must also know how the text position relates to the image of text on the screen; that is, which point in the text image corresponds to the text position.

Figure 3-9 illustrates a text image with the various reference points and lines pointed out. Each of these is discussed in detail in the following paragraphs.

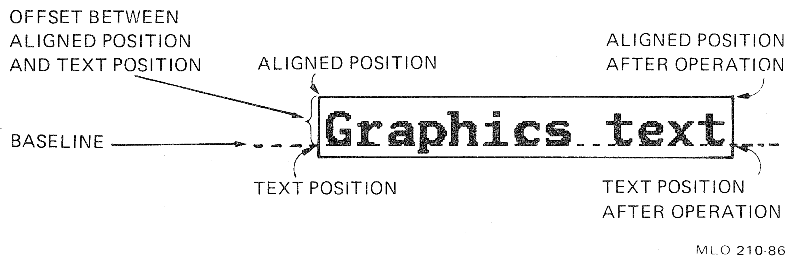

OFFSET BETWEEN ALIGNED POSITION AND TEXT POSITION
ALIGNED POSITION
ALIGNED POSITION AFTER OPERATION
BASELINE
Graphics text
TEXT POSITION
TEXT POSITION AFTER OPERATION
MLO-210-86

Figure 3-9: Text Reference Points

The text baseline is an imaginary line upon which the characters sit. Some letters, such as j and q, have descenders that dip below the baseline.

A call to TEXT that does not specify a starting position explicitly will place the text image so that the left end of the baseline coincides with the virtual display's text position. Following the execution of the function, the text position will be at the right end of the baseline. This means that you can use fonts of different sizes in successive calls to TEXT and know that they will all line up on a common baseline. Figure 3-10 illustrates this.

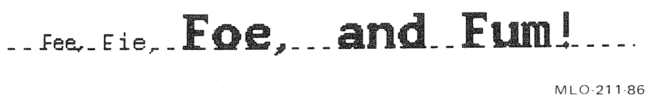


Figure 3-10: Using Different Fonts on the Same Line

When you give an explicit text position in a call to TEXT, you are specifying the aligned position at which the text should begin. The aligned position is at the upper left corner of the text image's character-cell box. Since the aligned position is offset vertically from the text position by a portion of the text's height, aligned position can only be determined with reference to a particular font. Therefore, an aligned position can only be specified with reference to an attribute block. The value of the attribute block's :FONT attribute is used to determine the aligned position.

3.6.2.2 **Changing the Text Position** - There are several ways to change the text position. Some of them are implicit, others are explicit.

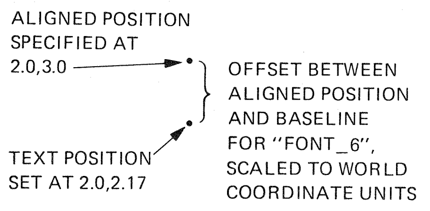
In all cases, it is important to note that a virtual display's text position is separate from the device-coordinate text position maintained for windows into the virtual display. A world-coordinate operation that affects the text position in a display does not affect the text position for device-coordinate operations on windows into that display. Similarly, a device-coordinate operation that affects the window text position does not affect the virtual display's text position. (However, changing the device-coordinate text position in one window does affect the device-coordinate text position in all other windows that map into that virtual display.)

A call to TEXT implicitly changes the virtual display's text position to the right end of the text image's baseline. A call to TEXT-PIXEL changes the device-coordinate text position in the same way. Thus, successive calls to TEXT or to TEXT-PIXEL will concatenate the output from left to right on a common baseline.

A call to the NEW-TEXT-LINE function changes the virtual display's text position to the left margin specified in the attribute block supplied in the function call. NEW-TEXT-LINE also moves the text position down from its previous setting by an amount that depends on the font specified in the attribute block. Thus, NEW-TEXT-LINE is like the RETURN key on a terminal. NEW-TEXT-LINE-PIXEL operates similarly in a viewport.

A call to the SET-POSITION function sets the virtual display's text position to an absolute world-coordinate location. The first argument to SET-POSITION is the display; the second and third arguments are the X and Y coordinates. The Y coordinate establishes the baseline for the next text operation, and the X coordinate establishes the left edge of the output. The SET-POSITION-PIXEL function operates similarly in a viewport.

The SET-ALIGNED-POSITION function also sets the virtual display's text position, but not directly. With SET-ALIGNED-POSITION, you specify where you want the aligned position (the upper left corner) of the next text output to fall. You must supply an attribute block with SET-ALIGNED-POSITION. The function calculates the text position from the aligned position you specify and the size of the font. Figure 3-11 illustrates this process.


ALIGNED POSITION SPECIFIED AT 2.0,3.0
OFFSET BETWEEN ALIGNED POSITION AND BASELINE FOR "FONT_6", SCALED TO WORLD COORDINATE UNITS
TEXT POSITION SET AT 2.0,2.17

```
(SET-ATTRIBUTE *DISPLAY*
    0 3 :FONT "FONT_6")

(SET-ALIGNED-POSITION
    *DISPLAY* 3 2.0 3.0)
```

MLO-212-86

Figure 3-11: Setting the Aligned Position

You must be sure to use the same font that you specified with SET-ALIGNED-POSITION in the call to TEXT that is to use the aligned position. If you use a different font, the aligned position of the output may be above or below the position you specified with SET-ALIGNED-POSITION.

The SET-ALIGNED-POSITION-PIXEL function is similar to SET-ALIGNED-POSITION except that it operates in a viewport instead of a virtual display.

All of the SET- functions mentioned in this section have corresponding GET- functions. These GET- functions return the location either of the text position or of the aligned text position with respect to a specified font.

3.6.2.3 **Measuring Text** - The MEASURE-TEXT function measures a string of text in a specified font, and returns the width and height of the text string's character-cell box in world-coordinate units. Since MEASURE-TEXT does not actually place any text on the screen, it allows you to preview the text and then place it according to its size. This capability is useful in the following situations:

- You can use MEASURE-TEXT to determine if a text string will fit between the text position and a right-hand limit that you have established. If the text string will not fit, you can call NEW-TEXT-LINE to start a new line.

- If you want to center an arbitrary line of text, you can use the width value returned by MEASURE-TEXT as shown in the following example:

  ```
  (DEFUN CENTER-TEXT (DISPLAY ATT-BLOCK TXT X Y)
    (TEXT DISPLAY ATT-BLOCK TXT
          (- X (/ (MEASURE-TEXT
                       DISPLAY ATT-BLOCK TXT)
                  2.0))
          Y))
  ```

  The function CENTER-TEXT accepts the same arguments as TEXT, but centers the text string on the specified X position instead of beginning it there. You could define a similar function to position the right edge of a text string at a specified point.

- If you want to write a string of text with a box around it, you can use the values returned by MEASURE-TEXT to determine the dimensions of the box. Example 3-1 shows one way to do this.

Example 3-1: Boxed Text

```
(DEFUN BOXED-TEXT (DISPLAY ATT-BLOCK TXT X Y)
  (MULTIPLE-VALUE-BIND (W H) (MEASURE-TEXT DISPLAY ATT-BLOCK TXT)
    (TEXT DISPLAY ATT-BLOCK TXT X Y)
    (RECTANGLE DISPLAY 0 X Y (+ X W) (- Y H))))

(DEFUN RECTANGLE (DISPLAY ATT-BLOCK X1 Y1 X2 Y2)
  (PLOT DISPLAY ATT-BLOCK X1 Y1 X2 Y1 X2 Y2 X1 Y2 X1 Y1))
```

The MEASURE-TEXT-PIXEL function is similar to MEASURE-TEXT, except that it returns the dimensions of the text string in terms of device coordinates instead of world coordinates.

3.6.3 Attributes that Affect Text

This section describes how to use the various attributes that affect text operations only. See Section 3.3.2 for a description of attributes that affect both text and graphic output.

3.6.3.1 **:CHARACTER-SPACING** - The :CHARACTER-SPACING attribute determines how much (if any) extra space should be left between characters in a string and between lines. You can add extra space between characters if you want to create an airy effect or fill a string of text out to a larger size.

The spacing between lines is used by the NEW-TEXT-LINE and NEW-TEXT-LINE-PIXEL functions. These functions add the interline space indicated by :CHARACTER-SPACING to the space indicated by the height of the font.

The value you specify for the :CHARACTER-SPACING attribute is a list of two floating-point numbers. The first number is the extra space to leave between characters. It is expressed as a proportion of the width of a character. The second number is the extra space to leave between lines, expressed as a proportion of the font height. The default for both is 0.0, indicating no extra space.

3.6.3.2 **:FONT** - The :FONT attribute determines the font in which text is written. A font is a collection of graphic characters in a particular size and style. The size of the font is its height in some physical unit, such as points (a point is equal to 1/72 of an inch) or centimeters. The style is the appearance of the font: for example, italic, bold, or roman.

In the VAX LISP graphics system, fonts are stored in files, one font to a file. The files are in a directory for which there is a logical name, SYS$FONT. Each file has a coded name that indicates the contents of the font, and the type .FNT. The file name is divided into several fields, with each field indicating one characteristic of the font. The SHOW-FONTS function displays all the available fonts and shows the value for each field that that differs from the default value.

You can specify a value for the :FONT attribute in one of two ways:

- You can use a pathname to the font file, or a character string that contains a file name specification or logical name pointing to the font file.

- You can also use a list of keyword-value pairs, where each keyword is one of the field-specification keywords displayed by the SHOW-FONTS function and the value is a character string containing the value of that field for the font you want. If you do not mention a field, SET-ATTRIBUTE fills in the value of that field from the default font specification.

As an example of the second method, consider the following example:

```
(SET-ATTRIBUTE *DISPLAY* 0 1
  :FONT '(:TYPE-FAMILY "TERMIN" :SPACING "M"
          :TYPE-SIZE "03C" :WEIGHT "P"))
```

This function sets up a :FONT attribute whose value differs from the value of the default :FONT attribute in four fields. The rest of the fields are taken from the default font values.

The values of each field for the default font (the font specified in attribute block 0) are as follows:

Field	**Value**
:TYPE-FAMILY	"TABER0"
:SPACING	"I"
:TYPE-SIZE	"03W"
:WEIGHT	"G"

One font file, specified by the system logical name UIS$FILL_PATTERNS, contains the patterns used to fill in figures. This font file is not useful for text output. (See Section 3.5.3.2 for information on using fill patterns.)

You can use the GET-FONT-SIZE function to find out the physical size of a font. GET-FONT-SIZE returns the width and height, in centimeters, of a specified text string in a specified font. Text characters are never scaled on the screen; therefore, the size information returned by GET-FONT-SIZE will always be valid, even if a viewport is scaled.

You specify the font to GET-FONT-SIZE the same way you specify it to SET-ATTRIBUTE. If you supply a keyword-value list, supply only those field specifiers whose values differ from the default font.

3.6.3.3 **:LEFT-MARGIN** - The :LEFT-MARGIN attribute sets up a margin to which the NEW-TEXT-LINE function sets the display's text position. You specify the :LEFT-MARGIN attribute as a floating-point number, which indicates an X value in world coordinate space. NEW-TEXT-LINE function calls that use this attribute block will cause the new line to start at that horizontal position.

The default value for :LEFT-MARGIN is the left edge of the default window you specified when you created the virtual display.

3.6.3.4 **:LEFT-MARGIN-PIXEL** - The :LEFT-MARGIN-PIXEL attribute sets up a margin to which the NEW-TEXT-LINE-PIXEL function sets the window text position. You specify the :LEFT-MARGIN-PIXEL attribute as an integer, which indicates a horizontal offset from the left edge of the viewport in device-coordinate units. NEW-TEXT-LINE-PIXEL function calls that use this attribute block will cause the new line to start at that horizontal position in the viewport.

The default value for :LEFT-MARGIN-PIXEL is 0, the left edge of the viewport.

The value of :LEFT-MARGIN-PIXEL is completely independent of the value of :LEFT-MARGIN. Changing the value of one has no effect on the value of the other. The NEW-TEXT-LINE-PIXEL function uses the value of :LEFT-MARGIN-PIXEL, and the NEW-TEXT-LINE function uses the value of :LEFT-MARGIN.

3.7 SEGMENTS

A segment is a grouping of graphic operations that is delimited by a call to the BEGIN-SEGMENT function at the beginning and a call to the END-SEGMENT function at the end.

Within a segment, any modifications that you make to attribute blocks are temporary. When you begin the segment, you have available all the attribute blocks that have been modified up to that point. Within the segment, you can use the SET-ATTRIBUTE function to modify these atttribute blocks. At the end of the segment, all the work you have done on attribute blocks is cancelled, and the state of the attribute blocks returns to what it was before you began the segment. Thus, you can use segments to ensure that previously-modified attribute blocks are not ruined. This can be useful in a portion of code that can be called anywhere in a program and that must set up one or more special attribute blocks.

Segments may be nested; that is, a segment can contain another segment. The inner segment inherits all the attribute blocks of the outer segment, including any that the outer segment may have modified.

For an example of the use of segments, consider a function called REVERSE-TEXT that takes a virtual display, an attribute block, a text string, and a coordinate pair. It is just like the TEXT function except that it attempts to output the text in the reverse of whatever the attribute block calls for. Example 3-2 shows how such a function might be defined.

Example 3-2: Reversing Text Using Segments

```
(DEFUN REVERSE-TEXT (D AB STR X Y)
  (BEGIN-SEGMENT D)
  (LET ((MODE                                       ; Get AB's writing mode
          (GET-ATTRIBUTE
              D AB :WRITING-MODE)))
    (SET-ATTRIBUTE                                  ; Modify block 1
        D AB 1 :WRITING-MODE                        ; Copy block AB except
        (CASE MODE                                  ;  for new writing mode
          (:OVERLAY :OVERLAY-NEGATE)
          (:OVERLAY-NEGATE :OVERLAY)
          (:REPLACE :REPLACE-NEGATE)
          (:REPLACE-NEGATE :REPLACE)
          (:ERASE :ERASE-NEGATE)
          (:ERASE-NEGATE :ERASE)
          (OTHERWISE MODE))))
  (TEXT D 1 STR X Y)                                ; Output reversed text
  (END-SEGMENT D))
```

The function REVERSE-TEXT in Example 3-2 uses the same attribute block that was passed to it, except that it attempts to use a writing mode that will reverse the text compared to the writing mode that is supplied. It must modify an attribute block to have the desired new value for :WRITING-MODE. However, it cannot modify an attribute block arbitrarily, because any attribute block might already be used by the program. Therefore, it begins a segment, then modifies attribute block 1 from the contents of the input attribute block, specifying a new writing mode. END-SEGMENT cancels the modification of attribute block 1 at the end of the function.

3.8 MOVING AND ERASING GRAPHIC INFORMATION

The VAX LISP graphics system provides a function that moves a portion of a virtual display to a different part of the virtual display, and another function that erases a portion of a virtual display. Two device-coordinate functions perform the same general operations in windows.

The MOVE-AREA function moves all the graphic objects within a specified rectangle from one location in a virtual display to another. A graphic object in this instance is any graphic output that was created with a single function. Thus, a circle is a graphic object, as is a line or a rectangle created with a single call to PLOT. A line of text is a graphic object only if it was created with a single call to TEXT.

If the display list is enabled, MOVE-AREA updates it to reflect the movement that has occurred. That is, the display-list representation of the moved graphic objects is modified to indicate the new location.

If there are any windows into a virtual display at the time you use MOVE-AREA, the viewports corresponding to those windows do not immediately reflect the updated display list. In a viewport, the effect of MOVE-AREA is to move the rectangle from one part of the viewport to another, with all its contents. Thus, if a line cuts through the rectangle, the portion of the line within the rectangle will be moved, although in the display list, none of the line has been moved. If you create a new window on the virtual display, you will see that the line is intact.

The ERASE function erases within a virtual display by deleting graphic objects from the display list. As with MOVE-AREA, the effect of ERASE on any windows into the virtual display does not immediately reflect the deletions in the display list. In a viewport, the rectangle specified with ERASE is simply blanked out. If the display list is re-executed at a later time, objects that were not deleted will be redisplayed.

The MOVE-AREA-PIXEL function moves an area from one part of a virtual display to another. It does not affect the display list. In a viewport, there is no concept of "graphic objects"; instead, the specified rectangle is simply picked up and moved with all its contents to another location. Fragments of drawings and text strings that are contained within the rectangle are relocated.

In the same way, the ERASE-PIXEL function blanks out the specified rectangle in a viewport.

CHAPTER 4

SCREEN IMAGES AND BITMAPS

This chapter explains how you can read screen images into an array, write arrays to the screen, and store, retrieve, and manipulate the array. The chapter is divided as follows:

- Section 4.1 explains bitmap arrays, which store representations of screen images in memory.
- Section 4.2 shows how you can read images from windows into bitmap arrays.
- Section 4.3 shows how to write bitmap arrays to the screen, and explains how various attributes can alter the appearance of the image.
- Section 4.4 explains how to create and read files that contain bitmap arrays.
- Section 4.5 presents functions that create, compare, and test bitmaps.
- Section 4.6 describes the BITBLT facility, by means of which you can alter bitmaps in various ways.

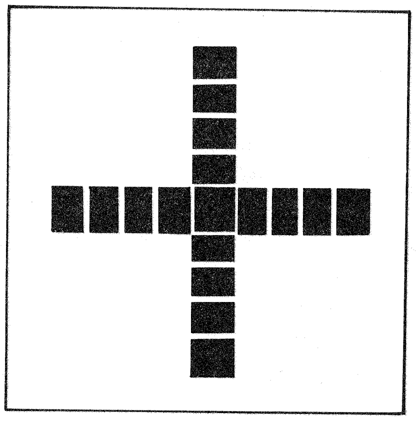
4.1 SCREEN IMAGES AND BITMAP ARRAYS

An image on the workstation screen is made up of individual pixels. In a monochrome system, each pixel is either illuminated or dark. In a color system, each pixel is one of a number of colors.

A bitmap is a representation in memory of a portion of the workstation screen. In the VAX LISP graphics system, a bitmap is represented by a specialized two-dimensional array of unsigned bytes. An array used in this way is called a bitmap array.

Each element of a bitmap array represents a pixel on the screen. The number of bits in each byte equals the number of bits used to represent each pixel on the workstation screen. In a monochrome system such as the VAXstation II, each pixel is represented by a single bit, whereas in a color system, each pixel is represented by several bits. In general, a workstation whose pixels are represented by *n* bits can simultaneously display 2***n* colors on the screen. For a monochrome system, where *n* is equal to 1, the screen can display two colors, black and white.

When VAX LISP graphics writes a bitmap array to a monochrome workstation screen, a bit with a value of 1 represents a pixel of the writing color, and a bit with a value of 0 represents a pixel of the background color. Figure 4-1 illustrates this relationship.



```
#2A((0 0 0 0 0 0 0 0 0 0 0)
    (0 0 0 0 0 1 0 0 0 0 0)
    (0 0 0 0 0 1 0 0 0 0 0)
    (0 0 0 0 0 1 0 0 0 0 0)
    (0 0 0 0 0 1 0 0 0 0 0)
    (0 1 1 1 1 1 1 1 1 1 0)
    (0 0 0 0 0 1 0 0 0 0 0)
    (0 0 0 0 0 1 0 0 0 0 0)
    (0 0 0 0 0 1 0 0 0 0 0)
    (0 0 0 0 0 1 0 0 0 0 0)
    (0 0 0 0 0 0 0 0 0 0 0))
```

MLO-213-86

Figure 4-1: Bitmaps and Screen Images

4.2 CREATING A BITMAP ARRAY FROM A SCREEN IMAGE

The READ-IMAGE-PIXEL function reads an image from a specified portion of a viewport. The function either returns a bitmap array or modifies an array that you supply to it. READ-IMAGE-PIXEL does not affect the image on the screen; it simply makes or modifies an array based on the pixels that make up the image.

It is important to note that READ-IMAGE-PIXEL uses the contents of a viewport and not the contents of a virtual display. A viewport can contain images from two sources: world-coordinate graphic operations in the virtual display on which it is based, and device-coordinate graphic operations in the viewport itself. READ-IMAGE-PIXEL does not discriminate between images from these two sources; anything that is visible in the viewport is included in the bitmap array.

If a viewport is partially off the workstation screen or is occluded by another viewport, READ-IMAGE-PIXEL still has access to all its contents.

The format of the READ-IMAGE-PIXEL function is:

```
UIS:READ-IMAGE-PIXEL window bitmap &OPTIONAL x1 y1 x2 y2
```

For *bitmap*, you can supply either a bitmap array or NIL. If you supply NIL, the function creates and returns a bitmap array. The size of the image read, and thus the size of the bitmap created, depends on the rectangle you specify with the four coordinate arguments.

If you supply a bitmap array for *bitmap*, the function modifies and returns that array. When you supply a bitmap, you should not supply the optional arguments *x2* and *y2*; the function ignores them if they are present. The size of the bitmap you supply determines the size of the image that is read.

Consider the following two forms:

```
(SETQ MAP (READ-IMAGE-PIXEL WINDOW-1 NIL 0 0 30 30))
(READ-IMAGE-PIXEL WINDOW-1 MAP 30 30)
```

The first form sets the value of MAP to be the 30x30 bitmap array created and returned by READ-IMAGE-PIXEL. The size of the array is specified by the four coordinate arguments in the call to READ-IMAGE-PIXEL. The array represents an area at the lower left corner of WINDOW-1.

The second READ-IMAGE-PIXEL call receives the same bitmap array as its second argument. It modifies this array to reflect the portion of WINDOW-1 with one corner at 30,30 and the other at 59,59. In this call, the dimensions of MAP determine the size of the image read, and the last two arguments determine the lower left corner of the image.

READ-IMAGE-PIXEL returns information about the image without regard to writing and background colors or a color map. In a monochrome system, this means that pixels that are dark are always represented as 0, and pixels that are illuminated are always represented as 1. This can cause the image to be complemented when you write the bitmap array back to the screen. Section 4.3 explains this phenomenon.

4.3 WRITING A BITMAP ARRAY TO THE SCREEN

The IMAGE and IMAGE-PIXEL functions write a bitmap array to the workstation screen. They differ in several respects:

- The IMAGE function specifies the location for the image in terms of world coordinates in a virtual display; the IMAGE-PIXEL function specifies image location in terms of device coordinates in a viewport.

- The IMAGE function causes the image to appear in any viewport corresponding to a window that is mapped over the image location in the virtual display. The IMAGE-PIXEL function causes the image to appear in only one viewport.

- The IMAGE function modifies the display list; the IMAGE-PIXEL function does not.

- If you specify a target area that is larger than the bitmap array, the IMAGE function scales up the image to fit the target area, whereas the IMAGE-PIXEL function does not scale up the image. Both functions will clip the image if the target area is smaller than the image.

The IMAGE function places a bitmap array into a specified world-coordinate rectangle in the virtual display. Depending on the size of the rectangle in relation to the size of the bitmap array, IMAGE may clip the image at the rectangle, or may scale up the image to fill out the rectangle:

- If the size of the image in the viewport is larger than the size of the world-coordinate rectangle as it appears in the viewport, IMAGE clips off the right and/or bottom edge(s) of the image.

- If the size of the rectangle is larger than the size of the image by at least an integer multiple in either dimension, IMAGE scales up the image on a per-pixel basis to fill out the rectangle as nearly as possible. The scaling may be horizontal, vertical, or both. Any room left over between the scaled image and the rectangle appears above and to the right of the image.

Figure 4-2 shows the results of using IMAGE to clip and to scale bitmap arrays.

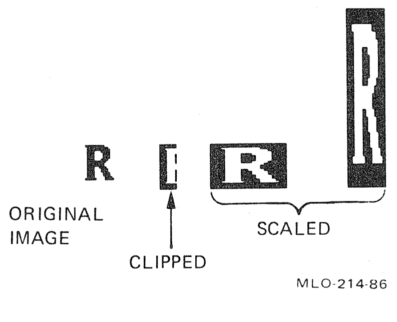


Figure 4-2: Writing Bitmap Arrays with IMAGE

IMAGE-PIXEL, in contrast to IMAGE, never scales or otherwise increases the size of a bitmap array. With IMAGE-PIXEL, you need supply only a single coordinate, the location for the lower-left corner of the image. IMAGE-PIXEL determines the size of the displayed image from the dimensions of the bitmap array that you supply. If you supply a second coordinate, and the rectangle so defined is smaller than the image, IMAGE-PIXEL clips the image on the right and/or bottom edge(s).

Both the IMAGE and IMAGE-PIXEL functions take an attribute block as their second argument. The functions use the :WRITING-INDEX and :BACKGROUND-INDEX attributes to determine how to represent each array element on the screen. A 0 is represented in the background color and a 1 is represented in the writing color.

If you are using windows with a light background and dark writing color, you will notice that images read from the window (with READ-IMAGE-PIXEL) and then written back to the window (with IMAGE or IMAGE-PIXEL) are complemented -- that is, the dark parts become light and the light parts become dark. This occurs because READ-IMAGE-PIXEL always returns a 0 for a dark pixel and a 1 for an illuminated pixel, without regard for writing or background colors. The IMAGE functions interpret the 0 as background (light) and the 1 as writing color (dark). Thus, what was background becomes writing color, and vice versa. This phenomenon does not occur in windows that have a dark background.

You can use an attribute block with the :WRITING-MODE attribute set to :OVERLAY-NEGATE to counteract the complementing of an image taken from a light-background window. Or, you can use the :COPY writing mode, which causes the image to be written to the screen without regard to writing color or background color. That is, with :COPY, a 1 always illuminates the pixel and a 0 always darkens it.

4.4 STORING BITMAP ARRAYS IN FILES

Two functions allow you to store a bitmap array in a file and to retrieve it from a file:

- DUMP-BITMAP creates a file containing a bitmap array.
- LOAD-BITMAP returns a bitmap array from a file created with DUMP-BITMAP.

Unlike most of the functions documented in this manual, these two functions are both in package LISP and are therefore available to you without a package specification.

4.5 CREATING, COMPARING, AND TESTING BITMAP ARRAYS

One way to create a bitmap is by using the READ-IMAGE-PIXEL function. Another way is by using the MAKE-BITMAP function. This function returns a bitmap array of the dimensions you specify. Optional arguments allow you to specify the number of bits used to represent each pixel, and the space in which the array is allocated. This function is in package LISP.

The COMPARE-BITMAPS function compares the two bitmap arrays given as its arguments. It returns T if the arrays are identical in both dimensions and contents. If it returns NIL, the additional return values provide more information about the difference. See the description of the function in Part II for more information. This function is also in package LISP.

The BITMAP-P function returns T if its argument is a valid bitmap array and NIL otherwise. It too is in package LISP.

4.6 ALTERING BITMAPS

This section describes the means by which you can alter a bitmap array. You may want to read an image into a bitmap array, alter it, and write it back to the same location; or you may wish to create a bitmap array from some nonimage source, modify it, and write it to the screen.

To alter a bitmap array, you use the BITBLT function, operating on BITBLT objects. (BITBLT stands for BIT Block Logical Transfer.) BITBLT objects contain complete specifications for a particular operation on a particular bitmap array. When you make a BITBLT object (with the MAKE-BITBLT function) you provide these specifications as keyword arguments. You then supply the BITBLT object as the single argument to the BITBLT function; this actually causes the operation to happen.

As a simple example, imagine that you want to overlay part of a viewport with a pattern of vertical bars. One way to do this is as follows:

1. Create a bitmap array from the portion of the viwport that you wish to overlay:

```
(SETF BARS-BITMAP (READ-IMAGE-PIXEL
                     WIN-1 NIL 50 50 120 150))
```

2. Create a BITBLT object that describes the operation you wish to perform:

```
(SETF BARS (MAKE-ARRAY '(1 4)
              :ELEMENT-TYPE 'BIT
              :INITIAL-CONTENTS '((1 1 0 0))))
(SETF BARS-BITBLT (MAKE-BITBLT
                    :SOURCE BARS-BITMAP
                    :DESTINATION BARS-BITMAP
                    :SRC-OP BOOLE-IOR
                    :TEXTURE BARS))
```

   This BITBLT object specifies that the BARS-BITMAP array should be combined with the array BARS, using the BOOLE-IOR function. The result of this operation is then used to replace the BARS-BITMAP array.

3. Pass the BITBLT object you created to the BITBLT function:

```
(BITBLT BARS-BITBLT)
```

   This has the effect of altering the bitmap array specified by BARS-BITBLT so that it has vertical bars running down it, but is otherwise the same as the original.

4. Write the altered bitmap array back to the screen:

```
(IMAGE-PIXEL WIN-1 0 BARS-BITMAP 50 50 120 150)
```

This example illustrates most of the features of the BITBLT facility, although much more control is available. With MAKE-BITBLT, you may specify:

- A source bitmap array. Keyword arguments allow you to use a specified rectangle of the source bitmap array.

- A texture bitmap array, which will be combined with the source bitmap array.

- A source operation (:SRC-OP), which specifies the result when elements of the source bitmap array are combined with corresponding elements of the texture bitmap array. The source operation can be any of the constants that you can supply to the BOOLE function.

- A destination bitmap array. In the example, the destination bitmap array was the same as the source bitmap array, but it could be a different array. Keyword arguments allow you to specify which part of the destination bitmap array should be altered.

- A destination operation (not shown in the example), which specifies how elements in the source-texture result array should be combined with corresponding elements in the destination bitmap array. These can be constants that you supply to the BOOLE function. The default operation, used in the example, is BOOLE-1, which replaces the destination bitmap array with the source-texture result array.

In general terms, a BITBLT operation proceeds as follows:

1. The texture bitmap is filled out (by horizontal replication) or trimmed on the right so that it is 32 elements wide. (It can be of any height.)

2. For each element in the destination bitmap (or specified portion thereof), the corresponding element in the source bitmap is combined with the corresponding element in the texture bitmap according to the source operation.

3. The result of combining the source and texture elements is combined with the destination element, according to the destination operation. This result replaces the destination element.

For more information and exact specifications of the BITBLT operation, see the description of the MAKE-BITBLT function in Part II of this manual.

CHAPTER 5

POINTER OPERATIONS

Every VAXstation is equipped with some sort of a pointer input device. The pointer may be a mouse, a graphics tablet, or some other device. All pointer input devices have in common the ability to move the pointer cursor around the screen, and all have one or more buttons to allow the user to request that some action take place.

Typical uses for the pointer are:

- Selecting an icon on the screen or an item from a menu
- Selecting portions of a program, a document, or a drawing that have been made pointer-sensitive
- Providing positional input for a graphics editing application

The VAX LISP graphics system provides functions that allow your program to make use of the pointer in various ways. Section 5.1 describes the various functions. Section 5.2 shows how you can establish and make use of pointer sensitivity. When you make part of a viewport pointer-sensitive, you request that the graphics system alert you when the pointer cursor enters that region. Pointer sensitivity is used to implement menus, icons, and other workstation features.

5.1 POINTER-RELATED FUNCTIONS

This section describes the functions that allow you to get input from the pointer and to control it from your program. The section is divided as follows:

- Section 5.1.1 describes pointer positional functions. These functions provide information on the location of the pointer cursor, and allow you to relocate the pointer cursor under program control.

- Section 5.1.2 describes pointer movement functions. These functions allow your application to react to movement of the pointer cursor within defined areas of a viewport.
- Section 5.1.3 describes button input functions. These functions allow your application to determine the state of the pointer buttons and to react to a button being pressed or released.

Throughout this section, reference will be made to interrupt functions. An interrupt function is a function that is meant to be invoked at some unknown time, interrupting the normal flow of program execution. Interrupt functions are necessary to handle pointer input because the user can move the pointer and press buttons at any time, not just when it is convenient for the program.

The *VAX LISP System Access Programming Guide* contains information on interrupt functions and how to use them.

5.1.1 Obtaining and Setting Pointer Position

Three functions return the position of the pointer cursor. There is one function for each of the three coordinate systems to which you have access:

- The GET-POINTER-POSITION function returns the position of the pointer cursor in world coordinates.
- The GET-POINTER-POSITION-PIXEL function returns the position of the pointer cursor in device coordinates.
- The GET-ABS-POINTER-POSITION returns the position of the pointer cursor on the screen in screen coordinates (centimeters).

All three of the functions return multiple values, where the first value is the X coordinate and the second value is the Y coordinate. The first two functions return NIL for their first value if the pointer cursor is not in the specified window.

The pointer cursor has one particular pixel, called the active pixel, that is used to calculate the pointer cursor position. For the default pointer cursor, the active pixel is at the tip of the arrow.

Example 5-1 illustrates the use of the GET-POINTER-POSITION function to implement a very simple form of "rubber-banding." In this familiar graphics editing technique, one end of a line is anchored and the other end of the line tracks the pointer cursor. (You can imagine a rubber band with a tack through one end and a stylus in the other end, stretching and moving it.) In a graphics editor program, the user would indicate the final position for the line by pressing a button. The function in the example simply loops until the pointer cursor leaves the viewport, causing GET-POINTER-POSITION to return NIL.

Example 5-1: Rubber-Banding with GET-POINTER-POSITION

```
;;; Do rubber-banding in WINDOW starting at the pointer position
;;; when the function is called and ending when the pointer leaves
;;; the window.  DISPLAY's display list is disabled during the
;;; operation.

(DEFUN RUBBER-BAND (DISPLAY WINDOW)
  (DISABLE-DISPLAY-LIST DISPLAY)
  (BEGIN-SEGMENT DISPLAY)
  (SET-ATTRIBUTE DISPLAY 0 1                      ; Use :COMPLEMENT to
                 :WRITING-MODE :COMPLEMENT)       ;  draw and erase lines
  (MULTIPLE-VALUE-BIND (ANCHORED-X ANCHORED-Y)     ; Get start position
                       (GET-POINTER-POSITION DISPLAY WINDOW)
      (DO* ((NEW-X ANCHORED-X)
            (NEW-Y ANCHORED-Y)
            (OLD-X NEW-X NEW-X)                   ; Coordinates for erasing
            (OLD-Y NEW-Y NEW-Y))                  ;  old line
           ((NOT NEW-X)                           ; Check for out-of-bounds
            (END-SEGMENT DISPLAY)                 ; Done; cancel block 1
            (ENABLE-DISPLAY-LIST DISPLAY))        ; Re-enable disp. list
;; Draw the line from start position to pointer position

        (PLOT DISPLAY 1
              ANCHORED-X ANCHORED-Y
              NEW-X NEW-Y)

;; Get new pointer position

        (MULTIPLE-VALUE-SETQ (NEW-X NEW-Y)
                             (GET-POINTER-POSITION DISPLAY WINDOW))

;; Erase old line

        (PLOT DISPLAY 1
              ANCHORED-X ANCHORED-Y OLD-X OLD-Y))))
```

Note that the GET-POINTER-POSITION function takes a virtual display and a window as its first two arguments. The window designates the viewport from which you want to get the pointer position. For the display argument, you can give the window's virtual display, or a transformation mapped into that display. The position information is returned in the coordinate system of the display or the transformation that you specify.

By contrast, the GET-POINTER-POSITION-PIXEL function takes only a window argument. The position information is returned in the device coordinate system of that window.

Two functions, SET-POINTER-POSITION and SET-POINTER-POSITION-PIXEL, allow you to relocate the pointer cursor in a specified viewport. There is no SET- equivalent to GET-ABS-POINTER-POSITION, since you cannot affect the workstation screen outside of viewports that your program has created.

5.1.2 Movement Input

A number of functions allow your program to respond to the movement of the pointer cursor within a specified viewport. The functions are the following:

- The SET-POINTER-ACTION and SET-POINTER-ACTION-PIXEL functions allow you to specify an interrupt function to be executed when the pointer cursor moves in or exits a viewport or specified portion of a viewport.

- The SET-POINTER-PATTERN and SET-POINTER-PATTERN-PIXEL functions allow you to specify a new image for the pointer cursor. The pointer cursor will change to that image when the pointer cursor is in the specified viewport or portion of the viewport.

The SET-POINTER-ACTION and SET-POINTER-PATTERN functions take a virtual display and a window as their first two arguments. The window designates the viewport in which you want to respond to pointer movement. For the display argument, you can give the window's virtual display, or a transformation mapped into that display. Optional arguments to each function let you specify a rectangle in which movement should trigger an action. The optional arguments are interpreted as world coordinates or transformation coordinates, depending on whether you supplied a virtual display or a transformation as the first argument.

The SET-POINTER-ACTION functions establish an action to be taken when the pointer cursor moves within a specified area, and a separate action to be taken if the pointer cursor exits the area. The area can be all of a viewport or a specified rectangle in the viewport. The actions can be one of two things: an interrupt function identifier (*iif-id*), designating a function to be executed when the cursor moves; or NIL, indicating that nothing should be done.

If you use SET-POINTER-ACTION to set up an interrupt function to respond to pointer movement, the function executes every time the pointer cursor moves. If the pointer cursor moves continuously, the interrupt function executes as often as the graphics system can invoke it. Sometimes this is the intended behavior; other times, you only want the interrupt function to execute once, to indicate that the pointer cursor has entered the viewport or rectangle. If this is the case, the interrupt function can turn itself off by resetting the action to NIL. An example of this can be seen in Section 5.2, which discusses pointer sensitivity.

Even in applications in which a pointer movement function should execute repeatedly, it is a good idea to set the movement action to NIL at the beginning of the function, and then re-establish it at the end of the function. This prevents requests for the movement function from queueing up while the movement function is executing.

Example 5-2 shows the use of SET-POINTER-ACTION to implement the function RUBBER-BAND. Recall that in Example 5-1 this function was implemented as a loop that continuously erased and replotted the line, whether or not the pointer cursor was moving. In Example 5-2, the line is only erased and plotted when the pointer cursor moves. RUBBER-BAND establishes initial values for the special variables *POINTER-X* and *POINTER-Y*, establishes the interrupt function DRAW-RUBBER-BAND as the action to take when the pointer cursor moves, sets the value of the special variable *DRAW-IIF-ID* to the *iif-id* of the interrupt function, and returns.

Every time the pointer cursor moves in the viewport associated with WINDOW, the graphics system invokes DRAW-RUBBER-BAND. DRAW-RUBBER-BAND first uses SET-POINTER-ACTION to turn off movement interrupt functions for the duration of its execution. It then erases the old line, obtains the current pointer position, and plots the new line. Finally, it re-establishes itself as the movement interrupt function and exits. If the cursor moves out of the viewport, DRAW-RUBBER-BAND does not execute again until the cursor re-enters the viewport.

The function STOP-RUBBER-BAND stops the drawing action by disabling the pointer movement interrupt function. It also uninstates the interrupt function. It is important to use UNINSTATE-INTERRUPT-FUNCTION to clean up interrupt functions that are no longer needed; otherwise, they consume system resources.

Note that DRAW-RUBBER-BAND uses special variables to retain the coordinates for one endpoint of the line last drawn. Since interrupt functions execute at unpredictable times and in unpredictable contexts, you cannot count on the binding of special variables during the execution of an interrupt function. In this case, you must be careful not to permit more than one function to alter the values of *POINTER-X* and *POINTER-Y* at a time. An example in Section 5.1.3 shows how special variables can be eliminated in this situation.

Example 5-2: Rubber-Banding with SET-POINTER-ACTION

```
(DEFVAR *POINTER-X*)
(DEFVAR *POINTER-Y*)
(DEFVAR *DRAW-IIF-ID*)

(DEFUN RUBBER-BAND (DISPLAY WINDOW)
  (BEGIN-SEGMENT DISPLAY)
  (SET-ATTRIBUTE DISPLAY 0 1 :WRITING-MODE :COMPLEMENT)
  (MULTIPLE-VALUE-SETQ (*POINTER-X* *POINTER-Y*)
                       (GET-POINTER-POSITION DISPLAY WINDOW))
  (SET-POINTER-ACTION
      DISPLAY
      WINDOW
      (SETF *DRAW-IIF-ID*
            (INSTATE-INTERRUPT-FUNCTION
                #'DRAW-RUBBER-BAND
                :ARGUMENTS (LIST DISPLAY WINDOW
                                 *POINTER-X*
                                 *POINTER-Y*)))
      NIL))

;;; DRAW-RUBBER-BAND is the interrupt function that erases the old
;;; line and plots a new one when the pointer moves.

(DEFUN DRAW-RUBBER-BAND
       (DISPLAY WINDOW ANCHORED-X ANCHORED-Y)

  (SET-POINTER-ACTION                          ; Turn off interrupts
      DISPLAY WINDOW NIL NIL)

;; Erase existing line

  (PLOT DISPLAY 1 ANCHORED-X ANCHORED-Y
            *POINTER-X* *POINTER-Y*)

;; Get new pointer position, plot a new line

  (MULTIPLE-VALUE-SETQ (*POINTER-X* *POINTER-Y*)
                       (GET-POINTER-POSITION DISPLAY WINDOW))
  (PLOT DISPLAY 1 ANCHORED-X ANCHORED-Y     ; Draw new line
        *POINTER-X* *POINTER-Y*)
  (SET-POINTER-ACTION                       ; Turn on interrupts
      DISPLAY
      WINDOW *DRAW-IIF-ID* NIL))

;;; Use STOP-RUBBER-BAND to turn off the rubberbanding action

(DEFUN STOP-RUBBER-BAND (DISPLAY WINDOW)
  (SET-POINTER-ACTION DISPLAY WINDOW NIL NIL)
  (UNINSTATE-INTERRUPT-FUNCTION *DRAW-IIF-ID*))
```

The call to INSTATE-INTERRUPT-FUNCTION function in RUBBER-BAND causes four arguments to be passed to DRAW-RUBBER-BAND: the display, the window, and the coordinates of the starting point. Every time the graphics system invokes DRAW-RUBBER-BAND, it passes these four arguments. This method of passing information to an interrupt function is safer than the use of special variables, and should be used when the information can be determined at the time that INSTATE-INTERRUPT-FUNCTION is evaluated.

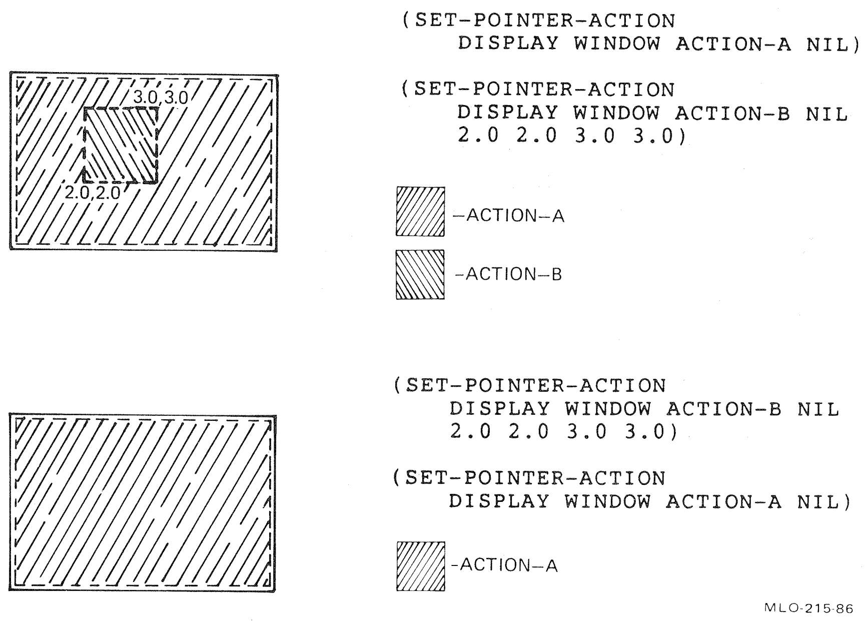
To establish an action to be taken when the pointer cursor exits an area, supply an *iif-id* for the *exit-action* argument of the SET-POINTER-ACTION functions. Once you have established an interrupt function as an exit action, the graphics system invokes that interrupt function every time the pointer cursor exits the specified viewport or viewport rectangle.

The SET-POINTER-ACTION functions both take four optional arguments that specify a rectangle in which the action is effective. Each pair of arguments supplies the coordinates of one corner of the rectangle. For SET-POINTER-ACTION, the coordinates are world or transformation coordinates. For SET-POINTER-ACTION-PIXEL, the coordinates are device coordinates.

You can use the SET-POINTER-ACTION functions more than once for a particular area. For example, you could use SET-POINTER-ACTION once to specify a movement action anywhere in a particular window, and then use it again with optional arguments to request a different action in a rectangle in that window. For any point in the window, the action will be the one that was last requested for that point. So, for example, if you issued SET-POINTER-ACTION for an entire window first and a portion of the window afterward, cursor movement would cause the first action everywhere in the window except in the portion specified in the second function. If you issued the functions in the reverse order, the effect of the second function would wipe out the effect of the first, and only the second action could be obtained. Figure 5-1 illustrates this.

The SET-POINTER-PATTERN and SET-POINTER-PATTERN-PIXEL functions allow you to specify that the pointer cursor should be changed in appearance whenever it enters a specified viewport or portion of a viewport. These functions can be useful when you have an application with a number of different windows, and you wish to help indicate the use of each window by the appearance of the cursor when it is in that window. For example, a window in which graphics editing takes place can have a crosshair cursor.

You specify the desired cursor appearance in the form of a 16x16 bitmap, that is, a two-dimensional array of bits. You must also specify the active pixel for the cursor, that is, the pixel which is used in calculating the actual cursor position.



```
(SET-POINTER-ACTION
    DISPLAY WINDOW ACTION-A NIL)

(SET-POINTER-ACTION
    DISPLAY WINDOW ACTION-B NIL
    2.0 2.0 3.0 3.0)
```

```
(SET-POINTER-ACTION
    DISPLAY WINDOW ACTION-B NIL
    2.0 2.0 3.0 3.0)

(SET-POINTER-ACTION
    DISPLAY WINDOW ACTION-A NIL)
```

Figure 5-1: Specifying Overlapping Areas for SET-POINTER-ACTION

Example 5-3 shows how you might set up a crosshair cursor and specify that it be used in a particular window. The two numbers (7 and 8) at the end of the SET-POINTER-PATTERN function specify the coordinates of the active pixel. Notice that the active pixel is given using device coordinates based on an origin of 0,0 at the lower-left corner of the bitmap, and not as an array reference.

You can use SET-POINTER-PATTERN and SET-POINTER-PATTERN-PIXEL interchangeably, as long as you are not specifying that only a portion of the viewport should cause the cursor to change. If you wish to specify a rectangle in the viewport, you must decide whether you want to use world coordinates or device coordinates. Use SET-POINTER-PATTERN if you want to use world coordinates, and SET-POINTER-PATTERN-PIXEL if you want to use device coordinates. For both functions, you use four optional arguments to specify a rectangle.

You can use SET-POINTER-PATTERN more than once for a particular area. For example, you could use SET-POINTER-PATTERN once to specify that the cursor should change to a crosshair anywhere in a particular window, and then use it again to request a different pattern in a rectangle in that window. For any point in the window, the cursor will be changed to the pattern that was last requested for that point. So, for example, if you issued SET-POINTER-PATTERN for an entire window first and a portion of the window afterward, the cursor would take on the first pattern everywhere in the window except in the portion specified in the second function. If you issued the functions in the reverse order, the effect of the second function would wipe out the effect of the first, and only the second pattern could be obtained.

Example 5-3: Setting the Cursor Pattern

```
(DEFCONSTANT CROSSHAIR-CURSOR
             (MAKE-ARRAY '(16 16)
                         :ELEMENT-TYPE 'BIT
                         :INITIAL-CONTENTS
                         '((0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0)
                           (0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0)
                           (0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0)
                           (0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0)
                           (0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0)
                           (0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0)
                           (0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0)
                           (1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0)
                           (0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0)
                           (0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0)
                           (0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0)
                           (0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0)
                           (0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0)
                           (0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0)
                           (0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0)
                           (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))))

(SET-POINTER-PATTERN NIL *EDITING-WINDOW* CROSSHAIR-CURSOR 7 8)
```

5.1.3 Button Input

Three functions allow your program to accept input from the pointer buttons. The functions are:

- GET-BUTTONS, which returns the state of the buttons at the time that it is called.
- SET-BUTTON-ACTION and its device-coordinate equivalent, SET-BUTTON-ACTION-PIXEL, which establish an action to perform when a button is pressed or released while the pointer cursor is in a specified viewport or portion of a viewport.

The GET-BUTTONS function returns two values. The first value is an integer that encodes the state of the buttons. (See GET-BUTTONS in Part II for an explanation and example of how to interpret this return value.) The second value is T or NIL, indicating whether the cursor is visible in the window you supply as an argument.

The two SET-BUTTON-ACTION functions allow you to establish an interrupt function which the graphics system will invoke when any button is pressed or released in a particular viewport. (You can also specify an action of NIL, meaning that the buttons should be ignored.) The interrupt function is passed two arguments by the graphics system: the code of the button that was pressed or released, and the direction of the transition (T if the button was pressed, NIL if released). The interrupt function must decide whether the particular button transition was meaningful and, if so, what to do about it.

Use the POINTER-BUTTON-*n* constants to test the button code in the interrupt function. Each of these constants corresponds to one of the buttons on the pointing device. For example, your interrupt function might look like this:

```
(DEFUN BUTTON-ACTION (BUTTON TRANSITION ...)
  (WHEN TRANSITION                        ; Ignore up transitions
    (CASE BUTTON
      (#.POINTER-BUTTON-1 ...)
      (#.POINTER-BUTTON-2 ...)
      ...)))
```

Example 5-4 shows the use of SET-BUTTON-ACTION to extend the RUBBER-BAND function of Example 5-2 into a crude method of drawing connected lines. DRAW-CONNECTED-LINES sets up a situation in which pressing the leftmost pointer button ends the current line and starts a new line, and pressing the rightmost button terminates the drawing operation. Moving the pointer cursor out of the window does not terminate the operation; instead, the window will sit idle until the pointer cursor returns, and then the drawing will resume.

It is important to note that the action established by the SET-BUTTON-ACTION functions is only effective when the pointer cursor is in the specified viewport. As soon as the pointer cursor leaves that viewport, the action becomes inactive until the cursor returns.

One problem with Example 5-4 is the number of special variables required. Even in this simple application, five special variables are required to keep track of the line coordinates and the *iif-id*. A more complicated application, such as a menu system, might have multiple areas in a viewport that would each be specified by three or four values. Thus, special variables are not a practical solution.

One way to solve the problem of maintaining information for interrupt functions is to pass a structure as an argument to the functions. The structure can have one slot for each piece of information that must be passed to each interrupt function call. Example 5-5 functionally reproduces Example 5-4, but it uses a structure instead of special variables. DRAW-CONNECTED-LINES sets up the structure and then causes it to be passed as an argument to DRAW-RUBBER-BAND and RUBBER-BAND-BUTTON-HANDLER. Those two functions communicate with each other and with successive calls to themselves by making alterations to the structure.

Example 5-4: Controlling Rubber-Banding with Pointer Buttons

```
(DEFVAR *ANCHORED-X*)
(DEFVAR *ANCHORED-Y*)
(DEFVAR *POINTER-X*)
(DEFVAR *POINTER-Y*)
(DEFVAR *RUBBER-IIF*)

;;; The user calls DRAW-CONNECTED-LINES to draw a series of
;;; lines.  The left button starts each new line and the right
;;; button finishes the series.

(DEFUN DRAW-CONNECTED-LINES (DISPLAY WINDOW)
  (BEGIN-SEGMENT DISPLAY)
  (SET-ATTRIBUTE DISPLAY 0 1 :WRITING-MODE :COMPLEMENT)

;; Set up the interrupt function that draws and erases lines

  (SETQ *RUBBER-IIF*
        (INSTATE-INTERRUPT-FUNCTION
            #'DRAW-RUBBER-BAND
            :ARGUMENTS (LIST DISPLAY WINDOW)))

;; Set up the interrupt function that handles button transitions.
;; LINE-DONE and DRAWING-DONE are flags that will be manipulated
;; by button handler.

  (LET* ((LINE-DONE (CONS NIL NIL))
         (DRAWING-DONE (CONS NIL NIL))
         (BUTTON-IIF
             (INSTATE-INTERRUPT-FUNCTION
                 #'RUBBER-BAND-BUTTON-HANDLER
                 :ARGUMENTS (LIST LINE-DONE DRAWING-DONE))))
    (SET-BUTTON-ACTION DISPLAY WINDOW BUTTON-IIF)

;; Wait for user to press left button to start first line

    (WAIT "Waiting for start of drawing" #'CAR LINE-DONE)
    (MULTIPLE-VALUE-SETQ                          ; Get start position
        (*ANCHORED-X* *ANCHORED-Y*)
        (GET-POINTER-POSITION DISPLAY WINDOW))
    (SETQ *POINTER-X* *ANCHORED-X* *POINTER-Y* *ANCHORED-Y*)

;; Execute the following loop once for each line that is drawn
;; permanently in the display.

    (LOOP
      (DISABLE-DISPLAY-LIST DISPLAY)
      (SET-POINTER-ACTION                         ; Start rubberbanding
         DISPLAY WINDOW *RUBBER-IIF* NIL)
      (WAIT "for end of rubberbanding" #'CAR LINE-DONE)
```

Example 5-4 (cont.)

```
      (SET-POINTER-ACTION                                 ; Stop rubberbanding
          DISPLAY WINDOW NIL NIL)
      (ENABLE-DISPLAY-LIST DISPLAY)                       ; Draw line permanently
      (PLOT DISPLAY 0 *ANCHORED-X* *ANCHORED-Y*
            (SETQ *ANCHORED-X* *POINTER-X*)
            (SETQ *ANCHORED-Y* *POINTER-Y*))
      (WHEN (CAR DRAWING-DONE) (RETURN))                  ; Right button?
      (SETF (CAR LINE-DONE) NIL))                         ; No, loop again
    (SET-BUTTON-ACTION DISPLAY WINDOW NIL)                ; Yes, dismantle the
    (UNINSTATE-INTERRUPT-FUNCTION BUTTON-IIF))            ;  machinery
  (DISABLE-DISPLAY-LIST DISPLAY)
  (END-SEGMENT DISPLAY)
  (UNINSTATE-INTERRUPT-FUNCTION *RUBBER-IIF*))

;;; DRAW-RUBBER-BAND draws and erases lines in response to
;;; pointer movement.

(DEFUN DRAW-RUBBER-BAND (DISPLAY WINDOW)

  (SET-POINTER-ACTION                                     ; Turn off interrupts
      DISPLAY WINDOW NIL NIL)

;; Erase existing line. Attribute block 1
;; contains :WRITING-MODE :COMPLEMENT

  (PLOT DISPLAY 1 *ANCHORED-X* *ANCHORED-Y*
            *POINTER-X* *POINTER-Y*)

;; Get new position and plot line

  (MULTIPLE-VALUE-SETQ (*POINTER-X* *POINTER-Y*)
                       (GET-POINTER-POSITION DISPLAY WINDOW))
  (PLOT DISPLAY 1
        *ANCHORED-X* *ANCHORED-Y*
        *POINTER-X* *POINTER-Y*)
  (SET-POINTER-ACTION                                     ; Restore interrupts
      DISPLAY WINDOW *RUBBER-IIF* NIL))

;;; RUBBER-BAND-BUTTON-HANDLER handles all button transitions.
;;; It ignores all transitions except down transitions of left and
;;; right buttons.

(DEFUN RUBBER-BAND-BUTTON-HANDLER
       (BUTTON TRANSITION           ; Args. from graphics system
        LINE-DONE DRAWING-DONE)     ; Args. supplied when instated
  (WHEN TRANSITION                  ; Ignore up transitions
    (CASE BUTTON
      (#.UIS::POINTER-BUTTON-1      ; Left button; start new line
         (SETF (CAR LINE-DONE) T))
```

Example 5-4 (cont.)

```
      (#.UIS::POINTER-BUTTON-3    ; Right button; end line and terminate
          (SETF (CAR LINE-DONE) T
                (CAR DRAWING-DONE) T)))))
```

Example 5-5: Using Structures to Eliminate Special Variables

```
;;; DRAW-INFO is a structure that holds all information necessary
;;; for line-drawing and button-handling interrupt functions.

(DEFSTRUCT DRAW-INFO
           DISPLAY
           WINDOW
           (ANCHORED-X 0.0 :TYPE SHORT-FLOAT)     ; Start position for
           (ANCHORED-Y 0.0 :TYPE SHORT-FLOAT)     ;  line
           (POINTER-X 0.0 :TYPE SHORT-FLOAT)      ; Current line endpoint
           (POINTER-Y 0.0 :TYPE SHORT-FLOAT)
           (LINE-DONE NIL)                        ; Flags for button
           (DRAW-DONE NIL)                        ;  handler
           RUBBER-IIF)

(DEFUN DRAW-CONNECTED-LINES (DISPLAY WINDOW)
  (BEGIN-SEGMENT DISPLAY)
  (SET-ATTRIBUTE DISPLAY 0 1 :WRITING-MODE :COMPLEMENT)
  (LET* ((DI (MAKE-DRAW-INFO                      ; Make structure to
                  :DISPLAY DISPLAY                ;  hold drawing
                  :WINDOW WINDOW))                ;  information
         (BUTTON-IIF                              ; Set up button handler
             (INSTATE-INTERRUPT-FUNCTION
                 #'RUBBER-BAND-BUTTON-HANDLER
                 :ARGUMENTS (LIST DI))))          ; Pass structure to
                                                  ;  interrupt function
    (SETF (DRAW-INFO-RUBBER-IIF DI)               ; Set up move handler
          (INSTATE-INTERRUPT-FUNCTION
              #'DRAW-RUBBER-BAND
              :ARGUMENTS (LIST DI)))              ; Pass structure
    (SET-BUTTON-ACTION DISPLAY WINDOW BUTTON-IIF)
    (WAIT "Waiting for start of drawing"
          #'DRAW-INFO-LINE-DONE DI)               ; Use slot as flag
    (MULTIPLE-VALUE-BIND (X Y)
        (GET-POINTER-POSITION DISPLAY WINDOW)
      (SETF                                       ; Put line coords.
          (DRAW-INFO-ANCHORED-X DI) X             ;  in structure
          (DRAW-INFO-ANCHORED-Y DI) Y
          (DRAW-INFO-POINTER-X DI) X
          (DRAW-INFO-POINTER-Y DI) Y))
    (LOOP
      (DISABLE-DISPLAY-LIST DISPLAY)
```

Example 5-5 (cont.)

```
      (SET-POINTER-ACTION                             ; Start rubberbanding
          DISPLAY WINDOW
          (DRAW-INFO-RUBBER-IIF DI) NIL)
      (WAIT "for end of rubberbanding"
            #'DRAW-INFO-LINE-DONE DI)                 ; Check struc. slot
      (SET-POINTER-ACTION DISPLAY WINDOW NIL NIL)
      (ENABLE-DISPLAY-LIST DISPLAY)
      (PLOT DISPLAY 0                                 ; Plot permanent line
            (DRAW-INFO-ANCHORED-X DI)                 ;  from coords. left
            (DRAW-INFO-ANCHORED-Y DI)                 ;  in structure
            (SETF (DRAW-INFO-ANCHORED-X DI)
                  (DRAW-INFO-POINTER-X DI))
            (SETF (DRAW-INFO-ANCHORED-Y DI)
                  (DRAW-INFO-POINTER-Y DI)))
      (IF (DRAW-INFO-DRAW-DONE DI) (RETURN))          ; Right button?
      (SETF (DRAW-INFO-LINE-DONE DI) NIL))
    (SET-BUTTON-ACTION DISPLAY WINDOW NIL)
    (END-SEGMENT DISPLAY)
    (UNINSTATE-INTERRUPT-FUNCTION (DRAW-INFO-RUBBER-IIF DI))
    (UNINSTATE-INTERRUPT-FUNCTION BUTTON-IIF)
    (DISABLE-DISPLAY-LIST DISP)))

;;; DRAW-RUBBER-BAND is now passed one argument, a structure

(DEFUN DRAW-RUBBER-BAND (DI)
  (SET-POINTER-ACTION
      NIL (DRAW-INFO-WINDOW DI) NIL NIL)
  (PLOT (DRAW-INFO-DISPLAY DI) 1
        (DRAW-INFO-ANCHORED-X DI)
        (DRAW-INFO-ANCHORED-Y DI)
        (DRAW-INFO-POINTER-X DI)
        (DRAW-INFO-POINTER-Y DI))
  (MULTIPLE-VALUE-BIND
      (X Y)
      (GET-POINTER-POSITION
          (DRAW-INFO-DISPLAY DI)
          (DRAW-INFO-WINDOW DI))
    (PLOT (DRAW-INFO-DISPLAY DI) 1
          (DRAW-INFO-ANCHORED-X DI)
          (DRAW-INFO-ANCHORED-Y DI)
          (SETF (DRAW-INFO-POINTER-X DI) X)
          (SETF (DRAW-INFO-POINTER-Y DI) Y)))
  (SET-POINTER-ACTION
      NIL
      (DRAW-INFO-WINDOW DI)
      (DRAW-INFO-RUBBER-IIF DI)
      NIL))

;;; RUBBER-BAND-BUTTON-HANDLER is passed a structure and
```

Example 5-5 (cont.)

```
;;; modifies two of its slots to show which button was pressed.

(DEFUN RUBBER-BAND-BUTTON-HANDLER
       (BUTTON TRANSITION DI)
  (WHEN TRANSITION            ; Ignore up transitions
    (CASE BUTTON
      (#.UIS::POINTER-BUTTON-1 ; Left button; start new line
          (SETF (DRAW-INFO-LINE-DONE DI) T))
      (#.UIS::POINTER-BUTTON-3 ; Right button; terminate
          (SETF (DRAW-INFO-LINE-DONE DI) T
                (DRAW-INFO-DRAW-DONE DI) T)))))
```

5.2 POINTER SENSITIVITY

In various applications it is useful to make a region of a viewport pointer-sensitive. A pointer-sensitive region is an area that visibly responds to the presence of the pointer cursor, and that initiates some action when the user presses a pointer button while the pointer cursor is in the region. In the standard VAXstation user interface, menu choices are pointer-sensitive; they change appearance when the pointer cursor enters them and they cause an action when the user presses a button over them.

There are two possible ways of making a region of a viewport pointer-sensitive:

- You can use the SET-POINTER-ACTION or SET-POINTER-ACTION-PIXEL function to define a rectangle within a viewport. Within this rectangle, pointer movement triggers an interrupt function. The first time the interrupt function executes, it changes the appearance of the contents of the rectangle. You also need to use SET-BUTTON-ACTION to establish an interrupt function to execute if a pointer button is pressed while in the rectangle. A second interrupt function executes when the pointer cursor leaves the rectangle, restoring the original appearance.

  A disadvantage of this method occurs when a viewport contains many pointer-sensitive regions, as is the case with a large menu. Each pointer-sensitive region consumes system I/O channels. In addition, data structures must be maintained for each region.

- A simpler and more economical alternative is to use SET-POINTER-ACTION and SET-BUTTON-ACTION just once for the entire viewport. In your program, you lay out a number of areas in the viewport, each containing an item that you wish to be pointer-sensitive. Each time the pointer movement interrupt function executes, it determines which region the pointer is in, and highlights that region if it has not already been highlighted. When the button interrupt function executes, it alters a data structure to indicate the region it occupies at that moment.

Example 5-6 shows the use of the second method outlined above to implement a simple menu system. The function MENU either constructs and displays a menu, or redisplays a menu that had previously been constructed and returned by the function. Since the menu contains only one column, the interrupt functions can determine the region simply by finding the vertical location of the pointer cursor in the viewport.

Example 5-6: A Simple Menu System

```
(USE-PACKAGE "UIS")

;; We need to remember the menu's window in the cases
;; when it has been made invisible.

(DEFSTRUCT (MENU   (:PREDICATE MENUP))
  WINDOW
  (RETURN-VALUE NIL)
)

;; Here's our function for drawing a box around an entry
;; (highlighting it), given the window entry,
;; its height and width.

(DEFUN MENU-BOX-PIXEL (W ATTR ENTRY ENTRY-HEIGHT ENTRY-WIDTH)
  (LET* (
       (LLX 0)                          ;; Lower left x position
       (LLY (* ENTRY ENTRY-HEIGHT)) ;; Lower left y position
       (URX ENTRY-WIDTH)                ;; Upper right x position
       (URY (+ LLY ENTRY-HEIGHT)))  ;; Upper right y position
   (PLOT-PIXEL W ATTR LLX LLY URX LLY URX URY LLX URY LLX LLY)))

;; We need to find out the size of the longest string in
;; the menu, so that the viewport size can be computed.

(DEFUN LONGEST (&REST ENTRIES)
    (REDUCE #'(LAMBDA (LOCAL-MAX THIS-ENTRY)
                   (MAX LOCAL-MAX (LENGTH THIS-ENTRY)))
            ENTRIES
            :INITIAL-VALUE 0))

;; The different action functions use this function to get
;; the distance from the bottom of the menu.
```

Example 5-6 (cont.)

```
(DEFUN GET-POINTER-POSITION-Y-PIXEL (WINDOW)
      (MULTIPLE-VALUE-BIND (X Y)
         (GET-POINTER-POSITION-PIXEL WINDOW)
         (DECLARE (IGNORE X))
         Y))

;;; The arguments to MENU will be an existing menu
;;; structure, T or NIL, and a number of strings.
;;;
;;;  Passing a old menu means you want to use it.
;;;  Passing NIL means make a new menu (that can be
;;;  used but once).
;;;  Passing T means make a new menu that is made
;;;  invisible once selected from.
;;;  The strings will be the elements of the menu
;;;  along with "Exit menu"

(DEFUN MENU (OLD-MENU &REST GIVEN-ENTRIES)
  (IF (MENUP OLD-MENU)
      (IF (NOT (WINDOWP (MENU-WINDOW OLD-MENU)))
          (ERROR "The given menu has been exited, ~
                  you must make a new menu")
         (PROGN ; Else make it visible
            (MOVE-VIEWPORT (MENU-WINDOW OLD-MENU) :INVISIBLE NIL)
            (SETF (MENU-RETURN-VALUE OLD-MENU) NIL)))

  (LET*
      ((ENTRIES (CONS "Exit menu" GIVEN-ENTRIES))
       (TEXT-ATTR-BLK 2)
       (DISPLAY (CREATE-DISPLAY 0.0 0.0 4.0 5.0 4.0 5.0)))
       ;; Boldface text looks nice in menus
      (SET-ATTRIBUTE DISPLAY 0 TEXT-ATTR-BLK :FONT '(:WEIGHT "p"))
      (MULTIPLE-VALUE-BIND (CHAR-SIZE-X-CM CHAR-SIZE-Y-CM)
                           (MEASURE-TEXT DISPLAY TEXT-ATTR-BLK " ")
      (LET*
          ;; Leave 2 characters of space on each side of the
          ;; longest entry
         ((MENU-WIDTH-CHARS (+ 4 (APPLY #'LONGEST ENTRIES)))
          (MENU-WIDTH (* CHAR-SIZE-X-CM MENU-WIDTH-CHARS))
          (MENU-HEIGHT (* CHAR-SIZE-Y-CM (LENGTH ENTRIES)))
          (WINDOW (CREATE-WINDOW DISPLAY NIL NIL NIL NIL
                                 :VIEWPORT-WIDTH MENU-WIDTH
                                 :VIEWPORT-HEIGHT MENU-HEIGHT
                                 :NOBANNER T)))

       (MULTIPLE-VALUE-BIND (CHAR-SIZE-X-PIXELS ENTRY-HEIGHT)
                            (MEASURE-TEXT-PIXEL WINDOW
                                                TEXT-ATTR-BLK " ")
```

Example 5-6 (cont.)

```
      (LET* (
          (COMPLEMENT-ATTR-BLK 200)

          ;; We will use last-entry to keep trace of the
          ;; currently highlighted entry.  A value of -1 means
          ;; no entry is highlighted.
          (LAST-ENTRY -1)
          (ONE-TIME (NOT (EQ T OLD-MENU)))
          (THIS-MENU (MAKE-MENU :WINDOW WINDOW))
          (WINDOW-RIGHT-MARGIN (* MENU-WIDTH-CHARS
                                  CHAR-SIZE-X-PIXELS))

          ;; The movement action finds where we are in the
          ;; viewport, what entry that corresponds to, highlights
          ;; the current entry and unhighlights the last entry if
          ;; they are different.

          (MOVEMENT-IIF
             (INSTATE-INTERRUPT-FUNCTION
                 #'(LAMBDA ()
                  (LET ((Y (GET-POINTER-POSITION-Y-PIXEL WINDOW)))
                     (WHEN Y
                        (SETF Y (FLOOR Y ENTRY-HEIGHT))
                        (WHEN  (NOT (EQ Y LAST-ENTRY))
                            (WHEN (NOT (EQ LAST-ENTRY -1))
                                  (MENU-BOX-PIXEL
                                    WINDOW COMPLEMENT-ATTR-BLK
                                    LAST-ENTRY ENTRY-HEIGHT
                                    WINDOW-RIGHT-MARGIN))
                            (MENU-BOX-PIXEL
                              WINDOW COMPLEMENT-ATTR-BLK
                              Y ENTRY-HEIGHT WINDOW-RIGHT-MARGIN)

                         (SETF LAST-ENTRY Y)))))))

          ;; The exit action unhighlights the current entry.
          ;; By racing the pointer very quickly across the menu, no
          ;; movement action may be recieved.  So we need to check
          ;; to see if any entries are highlighted.

          (EXIT-IIF
              (INSTATE-INTERRUPT-FUNCTION
                  #'(LAMBDA ()
                        (WHEN (NOT (EQ LAST-ENTRY -1))
                              (MENU-BOX-PIXEL
                                WINDOW COMPLEMENT-ATTR-BLK
                                LAST-ENTRY ENTRY-HEIGHT
                                WINDOW-RIGHT-MARGIN)
                          (SETF LAST-ENTRY -1)
```

Example 5-6 (cont.)

```
                    ))))

        ;; If the mouse is moved very quickly, the button action
        ;; may try to get the current pointer position after we
        ;; leave the menu's viewport.  WHERE-WE-ARE is used to
        ;; test this case.  If this menu is to be used only once,
        ;; or the "Exit menu" entry is selected, then we uninstate
        ;; the various actions and delete the display.

        (BUTTON-IIF
            (INSTATE-INTERRUPT-FUNCTION
                #'(LAMBDA (BUTTON-NUMBER TRANSITION)
                     (DECLARE (IGNORE BUTTON-NUMBER))
                     (WHEN (EQ TRANSITION NIL) ; Button released
                       (LET ((WHERE-WE-ARE
                                 (GET-POINTER-POSITION-Y-PIXEL
                                   WINDOW)))
                       (WHEN WHERE-WE-ARE
                        (LET ((THIS-ENTRY
                                  (FLOOR WHERE-WE-ARE
                                         ENTRY-HEIGHT)))

                          (WHEN (> THIS-ENTRY 0)
                              (SETF (MENU-RETURN-VALUE THIS-MENU)
                                    (ELT ENTRIES THIS-ENTRY)))

                           ;; An alternative here would be to
                           ;; not destroy the menu give the
                           ;; selection "Exit menu"

                         (IF (OR ONE-TIME (EQ THIS-ENTRY 0))
                           (PROGN
                             (UNINSTATE-INTERRUPT-FUNCTION
                               MOVEMENT-IIF)
                             (UNINSTATE-INTERRUPT-FUNCTION
                               EXIT-IIF)
                             (SETF (MENU-WINDOW THIS-MENU) NIL)
                             (DELETE-DISPLAY DISPLAY))
                           (MOVE-VIEWPORT WINDOW :INVISIBLE T))
                         )))
                       ))))

        )

    ;; We use the complement-attr-blk for highlighting entries.
    ;; The :COMPLEMENT writing mode will both highlight an entry,
    ;; and unhighlight a highlighted entry.

    (SETF (MENU-WINDOW THIS-MENU) WINDOW)
```

Example 5-6 (cont.)

```
    (SET-ATTRIBUTE DISPLAY 0 COMPLEMENT-ATTR-BLK
                   :WRITING-MODE :COMPLEMENT)
    (SET-ATTRIBUTE DISPLAY COMPLEMENT-ATTR-BLK COMPLEMENT-ATTR-BLK
                   :FONT "UIS$FILL_PATTERNS")
    (SET-ATTRIBUTE DISPLAY COMPLEMENT-ATTR-BLK COMPLEMENT-ATTR-BLK
                   :FILL-PATTERN :FOREGROUND)

    ;; The entries are put up on the screen

    (DO* ((THIS-ENTRY ENTRIES (CDR THIS-ENTRY))
          (Y ENTRY-HEIGHT (+ Y ENTRY-HEIGHT)))
         ((NULL THIS-ENTRY))

          (TEXT-PIXEL WINDOW TEXT-ATTR-BLK (CAR THIS-ENTRY)
                      (* 2 CHAR-SIZE-X-PIXELS) Y))

     ;; The action routines are set up,

     (SET-POINTER-ACTION-PIXEL WINDOW MOVEMENT-IIF EXIT-IIF)
     (SET-BUTTON-ACTION DISPLAY WINDOW BUTTON-IIF)

     ;; And the structure for this menu is returned, letting
     ;; you get the return value.  Passing the menu structure
     ;; to this function again will put the menu back on the
     ;; screen if it was made invisible last time.

     THIS-MENU
)))))))

;Another possiblity for the "exit menu" entry would have it
;just make the menu invisible.  In that case one would add a
;DESTROY-MENU function that cleaned up the menu structure
;and actions.
```

CHAPTER 6

KEYBOARD INPUT

The VAX LISP graphics system provides a means for your program to receive characters representing individual keystrokes from a keyboard attached to a viewport. The physical keyboard sends input to a program through a virtual keyboard which is associated with a viewport. From the user's point of view, the viewport to which the keyboard is attached shows the keyboard icon indicating that it is the active viewport. From the program's point of view, a particular virtual keyboard is generating keystrokes. The program can either define an interrupt function to receive keystrokes asynchronously, or can read the keystrokes in a synchronous fashion.

This chapter treats keyboard input in the following sections:

- Section 6.1 describes virtual keyboards and shows how to create them and attach them to windows.

- Section 6.2 explains how to capture and interpret keystrokes from a keyboard.

For the purposes of this chapter, the term keyboard refers to a virtual keyboard. The term physical keyboard refers to the keyboard on which you type.

6.1 VIRTUAL KEYBOARDS

A virtual keyboard is a LISP object that you create with the CREATE-KB function. It forms the link between the physical keyboard and your program. Although there is only one physical keyboard per workstation, you can have any number of virtual keyboards.

6.1.1 Using Virtual Keyboards: An Overview

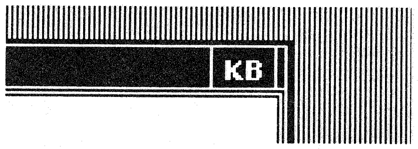
The first step in using a virtual keyboard is creating it. The CREATE-KB function creates and returns a virtual keyboard:

```
(SETF *KB* (CREATE-KB))
```

The next step is to associate the keyboard with a viewport. The ENABLE-VIEWPORT-KB function does this:

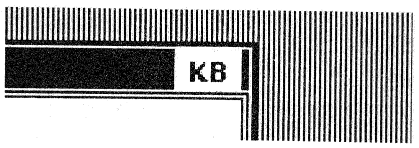
```
(ENABLE-VIEWPORT-KB *KB* *INPUT-WINDOW*)
```

When you associate a virtual keyboard with a viewport, a KB icon appears in the right corner of the viewport's banner:



This icon indicates to the user that the physical keyboard can be attached to this viewport. From the software point of view, the viewport is added to an assignment list of viewports that can have the physical keyboard attached to them. The CYCLE key (function key F5) attaches the physical keyboard to each viewport on the assignment list in turn.

When the physical keyboard is attached to the viewport, the KB icon is highlighted:



This indicates that keystrokes will now be directed through the virtual keyboard associated with this viewport.

NOTE

> If a call to CREATE-WINDOW includes :NOKB-ICON T, the associated viewport will be unable to acquire a KB icon. It can still, however, have a virtual keyboard associated with it.

Figure 6-1 illustrates the sequence of events just described.

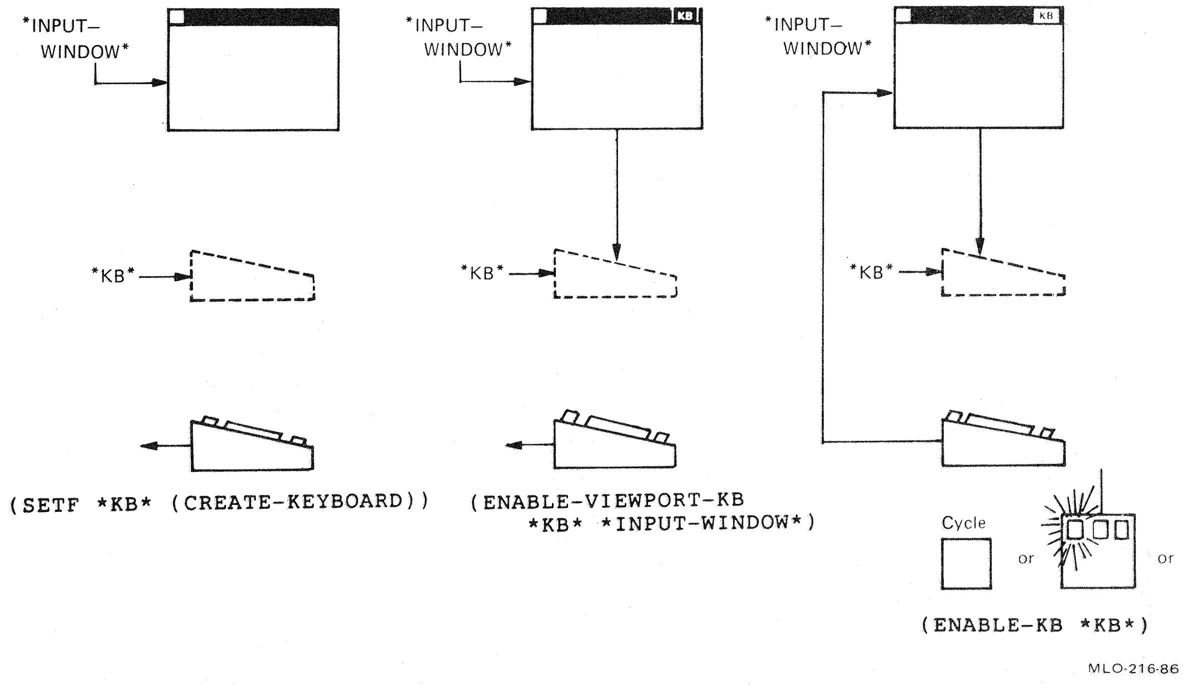


Figure 6-1: Creating and Attaching Virtual Keyboards

A virtual keyboard associated with a viewport to which the physical keyboard is attached is said to be active. This implies the following:

- Keyboard characteristics of the virtual keyboard (established with the SET-KB-ATTRIBUTES function) are imposed on the physical keyboard.

- Keystrokes on the physical keyboard are delivered through the virtual keyboard to your program. That is, an interrupt function specified for this virtual keyboard with the SET-KB-ACTION function will execute every time a key is struck. (See Section 6.2 for information on capturing and interpreting keystrokes.)

The ENABLE-KB function makes a particular virtual keyboard the active keyboard. It is equivalent to the user associating the physical keyboard with a viewport by pressing the CYCLE key or by pressing a pointer button. Use ENABLE-KB when you want to control which viewport the physical keyboard is connected to. Otherwise, let the user select the viewport with the CYCLE key or pointer.

A virtual keyboard can be associated with more than one viewport. (A viewport cannot, however, have more than one keyboard associated with it.) Consider this extension of the example presented above:

```
(ENABLE-VIEWPORT-KB *KB* *WINDOW-2*)
```

Now both *INPUT-WINDOW* and *WINDOW-2* are associated with the keyboard *KB*. Whenever *KB* becomes the active keyboard, through any means, the KB icon in both viewports will be highlighted. Both viewports have entries on the assignment list; that is, the CYCLE key will assign the physical keyboard to first one, then the other. However, whenever the physical keyboard is assigned to either, the KB icons in both will be highlighted.

It is important to note that the association of a virtual keyboard with a viewport makes no provision for the echoing of characters typed through that virtual keyboard, or for a cursor in the viewport. It is the responsibility of your program to take the appropriate response to input. Unless you set up an interrupt function to receive the keystrokes, they will be lost. Section 6.2 contains more information about this.

6.1.2 Creating and Deleting Virtual Keyboards

The CREATE-KB function creates and returns a virtual keyboard object. The keyboard returned by CREATE-KB is not associated with any viewport when it is created; you must do that later.

The DELETE-KB function deletes a virtual keyboard object. Since virtual keyboard objects consume system resources, you should take care to delete them when you no longer need them.

If you delete a keyboard that is currently associated with a viewport or with the physical keyboard, those associations are terminated.

6.1.3 Associating Keyboards with Viewports and the Physical Keyboard

Before you can receive any input from a virtual keyboard, you must associate it with a viewport. Two functions do this:

- The ENABLE-VIEWPORT-KB function associates the keyboard named in its first argument with the viewport corresponding to the window named in its second argument.

- The ENABLE-KB function's primary purpose is to make a keyboard the active keyboard. However, it takes an optional window argument which, if supplied, associates the keyboard with the corresponding viewport.

Associating a keyboard with a viewport causes any keyboards that were previously associated with that viewport to become dissociated. You can also use the DISABLE-VIEWPORT-KB function to explicitly dissociate a viewport and keyboard.

Once associated with a viewport, a keyboard can become active (associated with the physical keyboard) through user action or under program control:

- The user can make a keyboard active either by pressing the CYCLE key repeatedly, or by moving the pointer cursor into a viewport and pressing the left pointer button. (The second method does not work if the default button action has been superseded for that window; see Chapter 5.)

- The program can use the ENABLE-KB function to make a specific keyboard active.

Either the user or the program can make a virtual keyboard inactive. The user can press the CYCLE key or use the pointer to make another keyboard active; or the program can use the DISABLE-KB function.

Three functions let you find out if a virtual keyboard is active, and respond to a keyboard's becoming active and inactive:

- The TEST-KB function returns T if the keyboard named in its argument is connected to the physical keyboard, and NIL otherwise.

- The SET-GAIN-KB-ACTION and SET-LOSE-KB-ACTION functions specify actions to take when a specified virtual keyboard becomes active and inactive, respectively. The action can be an interrupt function or NIL to specify no action.

6.1.4 Setting Keyboard Attributes

Each virtual keyboard has associated with it a set of keyboard attributes. These attributes are imposed on the physical keyboard when the virtual keyboard becomes active. The attributes are the following:

- Autorepeat controls whether keys on the keyboard repeatedly generate a character when held down.

- Two keyclick attributes control whether, and how loudly, a click sounds when a key is pressed.

- Seven key group enabling and disabling attributes control whether keys in the following groups can generate keystrokes:

  - Function keys F6 through F10
  - Function keys F11 through F14
  - Function keys F17 through F20
  - The HELP and DO keys

- The six editing keys below the HELP and DO keys
- The arrow keys
- The numeric keypad keys

You set virtual keyboard attributes with the SET-KB-ATTRIBUTES function. The first argument to SET-KB-ATTRIBUTES is a virtual keyboard; the remaining arguments are keyword-value pairs that identify the attribute and its setting.

The GET-KB-ATTRIBUTE function returns the value of a particular attribute, and the GET-KB-ATTRIBUTE-LIST function returns a list of all the keyboard attributes and their settings for a specified virtual keyboard.

6.2 CAPTURING AND INTERPRETING KEYSTROKES

This section explains how to receive and interpret input from a virtual keyboard. Section 6.2.1 shows how to establish an interrupt function to handle keystrokes. Section 6.2.2 shows how to read keystrokes from a virtual keyboard synchronously. Section 6.2.3 describes the values that VAX LISP uses to designate each key.

6.2.1 Keyboard Interrupt Functions

To receive asynchronous input from a virtual keyboard, you must establish an interrupt function that will execute each time a keyboard key is pressed. The SET-KB-ACTION function establishes such an interrupt function for a specified virtual keyboard. The interrupt function receives at least two arguments: a character or integer designating the key that was struck,and a state flag. (The state flag is reserved for future use; you can ignore it.) You can supply additional arguments with the call to INSTATE-INTERRUPT-FUNCTION that defines the interrupt function.

Section 6.2.3 explains the first argument that is passed to a keyboard interrupt function.

6.2.2 Reading Keyboard Input Synchronously

You can also read individual keystrokes from a virtual keyboard in a synchronous fashion by using the READ-KB-CHAR function. This function returns the next character or integer from a specified keyboard. If no keystroke is available, READ-KB-CHAR does not return until one becomes available.

Section 6.2.3 explains the value returned by the READ-KB-CHAR function.

6.2.3 Characters Generated by Keys

For all the printing and control keys, the character received by a keyboard interrupt function is the LISP character corresponding to that key. However, some keys on the LK201 keyboard do not transmit single characters. These keys include the function keys, the arrow keys, the editing and numeric keypad keys, and the HELP and DO keys. Pressing these keys causes an integer to be generated by the virtual keyboard instead of a character. VAX LISP defines constants for each of these integers. Table 6-1 lists the constants corresponding to each key. Note that these constants are in a package called SMG.

Table 6-1: LISP Constants Corresponding to LK201 Keys

Key	Constant
	Numeric Keypad Keys
keypad 0	SMG:K-TRM-KP0
keypad 1	SMG:K-TRM-KP1
keypad 2	SMG:K-TRM-KP2
keypad 3	SMG:K-TRM-KP3
keypad 4	SMG:K-TRM-KP4
keypad 5	SMG:K-TRM-KP5
keypad 6	SMG:K-TRM-KP6
keypad 7	SMG:K-TRM-KP7
keypad 8	SMG:K-TRM-KP8
keypad 9	SMG:K-TRM-KP9
keypad -	SMG:K-TRM-MINUS
keypad ,	SMG:K-TRM-COMMA
keypad .	SMG:K-TRM-PERIOD
keypad Enter	SMG:K-TRM-ENTER
keypad PF1	SMG:K-TRM-PF1
keypad PF2	SMG:K-TRM-PF2
keypad PF3	SMG:K-TRM-PF3
keypad PF4	SMG:K-TRM-PF4
	Function, Help, and Do Keys
F6	SMG:K-TRM-F6
F7	SMG:K-TRM-F7
F8	SMG:K-TRM-F8
F9	SMG:K-TRM-F9
F10	SMG:K-TRM-F10
F11	SMG:K-TRM-F11
F12	SMG:K-TRM-F12

Table 6-1 (cont.)

Key	Constant
F13	SMG:K-TRM-F13
F14	SMG:K-TRM-F14
F15 (Help)	SMG:K-TRM-HELP
F16 (Do)	SMG:K-TRM-DO
F17	SMG:K-TRM-F17
F18	SMG:K-TRM-F18
F19	SMG:K-TRM-F19
F20	SMG:K-TRM-F20
Editing and Arrow Keys	
E1 (Find)	SMG:K-TRM-FIND
E2 (Insert Here)	SMG:K-TRM-INSERT-HERE
E3 (Remove)	SMG:K-TRM-REMOVE
E4 (Select)	SMG:K-TRM-SELECT
E5 (Prev Screen)	SMG:K-TRM-PREV-SCREEN
E6 (Next Screen)	SMG:K-TRM-NEXT-SCREEN
Up Arrow	SMG:K-TRM-UP
Down Arrow	SMG:K-TRM-DOWN
Right Arrow	SMG:K-TRM-RIGHT
Left Arrow	SMG:K-TRM-LEFT

CHAPTER 7

WINDOW OUTPUT STREAMS

The VAX LISP graphics system allows you to create a LISP output stream to a window. Output directed to this stream by any of the normal LISP output functions is displayed in the window's associated viewport. You can control the portion of the viewport in which output is displayed, and the way in which horizontal and vertical overflow are treated. You can also specify an attribute block to be used with the stream, allowing you to control the font and writing mode.

This chapter is divided as follows:

- Section 7.1 shows how to create and use window output streams.
- Section 7.2 shows how to alter the characteristics of a window output stream.
- Section 7.3 explains interactions between window output streams and other parts of the VAX LISP graphics system.

7.1 CREATING AND USING WINDOW OUTPUT STREAMS

One function and one macro create window output streams. The MAKE-WINDOW-OUTPUT-STREAM function creates and returns a window output stream. The WITH-OUTPUT-TO-WINDOW macro creates a window output stream and binds a variable to it while the forms in the macro's body execute. When the body terminates, WITH-OUTPUT-TO-WINDOW closes the window output stream.

Neither MAKE-WINDOW-OUTPUT-STREAM nor WITH-OUTPUT-TO-WINDOW creates a window object. Both take window objects as arguments and create a stream associated with that window. When the stream is closed, the window is not affected.

Once you have created a window output stream, you can use it with any of the COMMON LISP output functions that take a stream argument.

Characters output to the stream will appear in the window's viewport. The first output to the stream appears in the upper-left corner of the viewport, with subsequent output appended to the first output or on a line below it, depending on the LISP output function used.

MAKE-WINDOW-OUTPUT-STREAM and WITH-OUTPUT-TO-WINDOW both take keyword arguments that control the characteristics of the window output stream. The keywords, and the characteristics they control, are:

- :VIEWING-AREA -- controls the area of the viewport in which output text is displayed. By default, the entire viewport is used. To specify the viewing area, supply a list in the form (*x1 y1 x2 y2*), giving the device coordinates of a rectangle in the viewport. If you specify a viewing area, initial text output takes place in the upper-left corner of the viewing area.

  You can create multiple streams to the same window, with each stream having its own viewing area. This provides a facility similar to scrolling areas on a video terminal.

- :HORIZONTAL-OVERFLOW -- controls the behavior when an output operation attempts to write text beyond the right edge of the viewing area. The value can be :TRUNCATE, causing characters to be dropped, or :WRAP, causing characters to be wrapped onto the next line. Excess characters are wrapped by default. Text is truncated or wrapped until the stream encounters a #\NEWLINE character.

- :VERTICAL-OVERFLOW -- controls the behavior when an output operation attempts to write text below the bottom of the viewing area. The value can be :SCROLL, indicating that lines be scrolled upwards to accomodate the new line, or :WRAP, indicating that text be vertically wrapped, with the new line replacing the line at the top of the screen. Text is scrolled by default.

- :ATTRIBUTE-BLOCK -- designates the attribute block used for text output. By default, text is written with attribute block 0. However, if you have created a new attribute block for the display associated with the output window, you can specify that attribute block with this keyword. By specifying a different attribute block, you can change the font and writing mode used to output text.

The ERASE-VIEWING-AREA function erases anything within the viewing area of a window output stream, and resets the text position to the upper-left corner of the viewing area.

7.2 ALTERING WINDOW OUTPUT STREAMS

Once a window output stream has been created, you can alter any of the characteristics that you established when you created it. The following sections show how to do this:

- Section 7.2.1 shows how to change the viewing area.
- Section 7.2.2 shows how to change the overflow characteristics.
- Section 7.2.3 shows how to change the attribute block used to write output through the stream.

7.2.1 Changing the Viewing Area

The WINDOW-STREAM-VIEWING-AREA function returns the viewing area for a particular window output stream. You can use SETF with this function to change the viewing area. Use the same format that you would with the :VIEWING-AREA keyword to MAKE-WINDOW-OUTPUT-STREAM; that is, a list of device coordinates in the form (*x1 y1 x2 y2*).

Whenever you change a window output stream's viewing area, the stream's current writing position (that is, where the next text will be written) is set to the upper-left corner of the new viewing area, and the new viewing area is erased.

The viewing area of a window output stream is not automatically resized when the user resizes the associated viewport. For example, if you set up a window output stream without a specific viewing area -- that is, the viewing area consists of the entire viewport -- and the user enlarges the viewport, your window output stream still displays and scrolls within the bounds of the original viewport. Similarly, if the user shrinks the viewport, text can be lost beyond the upper and right borders of the viewport. To avoid this problem, you can use the SET-RESIZE-ACTION function to establish an interrupt function to be called when the user resizes the viewport. The interrupt function can resize the viewing area appropriately. Example 7-1 shows how you can do this.

Example 7-1: Resizing the Viewing Area Automatically

```
(DEFUN MAKE-RESIZABLE-WINDOW-OUTPUT-STREAM (WINDOW)
  (LET* ((STR (MAKE-WINDOW-OUTPUT-STREAM WINDOW))
         (RESIZE-IIF (INSTATE-INTERRUPT-FUNCTION
                       #'RESIZE-VIEWING-AREA
                       :ARGUMENTS (LIST STR))))
    (SET-RESIZE-ACTION NIL WINDOW RESIZE-IIF)
    STR))
```

Example 7-1 (cont.)

```
(DEFUN RESIZE-VIEWING-AREA (NEW-SCREEN-X NEW-SCREEN-Y
                            NEW-WIDTH NEW-HEIGHT
                            X1 Y1 X2 Y2
                            STR)
  (MULTIPLE-VALUE-BIND (JUNK-1 JUNK-2 H-RES V-RES)
                       (GET-DISPLAY-SIZE)
    (DECLARE (FLOAT H-RES V-RES)
             (IGNORE JUNK-1 JUNK-2))
    (SETF (WINDOW-STREAM-VIEWING-AREA STR)
          (LIST 0 0
                (FLOOR (* H-RES NEW-WIDTH))       ; New viewport size
                (FLOOR (* V-RES NEW-HEIGHT)))))) ; in pixels
  (RESIZE-WINDOW NIL (WINDOW-STREAM-WINDOW STR)
                 NEW-SCREEN-X NEW-SCREEN-Y NEW-WIDTH NEW-HEIGHT
                 X1 Y1 X2 Y2))
```

7.2.2 Changing Overflow Behavior

You can change the way a window output stream behaves when text overflows either the right or bottom edge of the viewing area. The WINDOW-STREAM-HORIZONTAL-OVERFLOW function returns the current horizontal overflow behavior of a window stream in the form of a keyword, either :TRUNCATE or :WRAP. Use SETF with this function to change the behavior, specifying the appropriate keyword as the new value. Similarly, WINDOW-STREAM-VERTICAL-OVERFLOW returns the vertical overflow behavior, either :SCROLL or :WRAP. You can also use SETF with this function to change the behavior.

7.2.3 Changing the Attribute Block

A window output stream writes text to a window through an attribute block. The significant attributes are :FONT and :WRITING-MODE. (Two other attributes, :LEFT-MARGIN-PIXEL and :CHARACTER-SPACING, are ignored.) You can specify an attribute block at the time the stream is created. You can also change the attribute block used by an existing stream.

The WINDOW-STREAM-ATTRIBUTE-BLOCK function returns the attribute block used by a window output stream. You can use SETF with this function to specify a new attribute block. The new attribute block must be associated with the window stream's virtual display.

NOTE

Changing the size of the font in which text is written may cause vertical interference between characters on consecutive lines.

You should use caution when making changes (using SET-ATTRIBUTE) to an attribute block used by a window output stream. Changing an attribute block in this way can cause unpredictable results in the window output stream. If you make changes to an attribute block, use SETF with the WINDOW-STREAM-ATTRIBUTE-BLOCK function immediately afterwards. This allows the window output stream to adjust to the changes in the attribute block.

7.3 WINDOW OUTPUT STREAMS AND OTHER GRAPHICS FUNCTIONS

This section contains information on how window output streams interact with other features of the VAX LISP graphics system.

7.3.1 Window Text Position

Window output streams write text to windows using the TEXT-PIXEL function. Each use of TEXT-PIXEL moves the window text position for all windows into a virtual display to the end of the text just written. Therefore, if you have several windows into a virtual display, and one of those windows has a window output stream associated with it, use of that stream will cause the window text position to be changed for all windows into the virtual display.

However, each window output stream maintains its own current writing position. Therefore, you can have multiple window output streams into a single window or into multiple windows on the same virtual display without interference.

7.3.2 Vertical Scrolling and Erasing

Window output streams use MOVE-AREA-PIXEL to scroll text upward. MOVE-AREA-PIXEL moves everything in the viewport, no matter how it got there. This means that graphic information, such as lines, that you may have placed in the viewport will be scrolled along with the text. If the graphic information is encoded in the display list, it will reappear the next time the display list is executed, for example when the user resizes the viewport.

In the same way, ERASE-VIEWING-AREA uses ERASE-PIXEL to erase the viewing area. Everything in the viewing area is erased.

7.3.3 Display List

Since window output streams write text with TEXT-PIXEL, they do not modify the display list.

PART II

GRAPHICS SYSTEM COMPONENTS

BEGIN-SEGMENT Function

Begins a new segment. The contents of all attribute blocks are propagated to the new segment. Changes to attribute blocks made during segment creation are cancelled when the segment ends, and the original settings are restored. A segment is terminated by a call to END-SEGMENT.

See Section 3.7 for more information on segments.

Format

UIS:BEGIN-SEGMENT *display*

Arguments

display

A virtual display

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$BEGIN_SEGMENT

BITBLT Function

Performs the operation specified by its argument, which must be a BITBLT object. See the description of the MAKE-BITBLT function for a description of the operation.

This function is in package LISP.

Format

LISP:BITBLT *bitblt*

Arguments

bitblt

An object of type BITBLT

Return Value

Undefined

Corresponding MicroVMS Routine

None

BITBLT Type Specifier

Designates objects of type BITBLT; these objects represent specific cases of moving and modifying a block of bits. When supplied as the argument to the BITBLT function, an object of type BITBLT causes a specific operation to be performed on a specific bitmap. BITBLT objects can be created as needed with the MAKE-BITBLT function. To perform identical or similar operations on different bitmaps, a BITBLT object can be modified with any of a number of accessor functions and reused.

The various parameters and operations specified by an object of type BITBLT are described with the MAKE-BITBLT function.

BITBLT Accessor Functions

BITBLT-DESTINATION	**BITBLT-DST-X**	**BITBLT-DST-Y**
BITBLT-DST-W	**BITBLT-DST-H**	**BITBLT-DST-OP**
BITBLT-SOURCE	**BITBLT-SRC-X**	**BITBLT-SRC-Y**
BITBLT-SRC-W	**BITBLT-SRC-H**	**BITBLT-SRC-OP**
	BITBLT-TEXTURE	

These accessor functions return the indicated component of their argument, which must be a BITBLT argument. They can also be used with SETF to modify that component of their argument. See the description of MAKE-BITBLT for descriptions of these components.

These functions are in package LISP.

Format

LISP:BITBLT-xxx *bitblt*

Arguments

bitblt

An object of type BITBLT

Return Value

The indicated component of the BITBLT object

Corresponding MicroVMS Routines

None

BITBLT-P Function

Returns T if its argument is a BITBLT object and NIL otherwise. This function is in package LISP.

Format

LISP:BITBLT-P *object*

Arguments

object

Any LISP object

Return Value

T or NIL

Corresponding MicroVMS Routine

None

BITMAP-P Function

Returns T if its argument is a two-dimensional array of unsigned bytes, suitable for use with the BITBLT and IMAGE functions, and NIL otherwise.

This function is in package LISP.

Format

LISP:BITMAP-P *object*

Arguments

object

Any LISP object

Return Value

T or NIL

Corresponding MicroVMS Routine

None

CIRCLE Function

Draws a circle or arc in a virtual display. Calls to this function draw a full circle unless the optional *start-radians* and *end-radians* arguments are specified. If these arguments are included, CIRCLE draws an arc from *start-radians* counterclockwise to *end-radians*. The radian start and end positions are measured from the right-hand intersection of the circle and its horizontal diameter.

For more information on drawing circles, see Section 3.5.2.

Format

```
UIS:CIRCLE display att-block center-x center-y radius
     &OPTIONAL start-radians end-radians
```

Arguments

display

A virtual display or transformation

att-block

A fixnum in the range 0-255, designating an attribute block from which graphics attributes will be taken

center-x center-y

Two single floats designating, in world coordinates, the center of the circle or arc

radius

A single float designating the radius of the circle or arc in world-coordinate units

start-radians

A single float designating the starting position of the circle in radians. If this argument is NIL or omitted, it defaults to 0.0.

end-radians

A single float designating the end position of the circle in radians. If this argument is NIL or omitted, the end position is at 2*PI.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$CIRCLE

CIRCLE-PIXEL Function

Draws a circle or arc in a window. Calls to this function draw a full circle unless the optional *start-radians* and *end-radians* arguments are specified. If these arguments are included, CIRCLE-PIXEL draws an arc from *start-radians* counterclockwise to *end-radians*. The radian start and end positions are measured from the right-hand intersection of the circle and its horizontal diameter.

For more information on drawing circles, see Section 3.5.2.

Format

UIS:CIRCLE-PIXEL *window att-block center-x center-y radius*
&OPTIONAL *start-radians end-radians*

Arguments

window

A window

att-block

A fixnum in the range 0-255, designating an attribute block from which graphics attributes will be taken

center-x center-y

Two fixnums designating, in device coordinates, the center of the circle or arc

radius

A fixnum designating the radius of the circle or arc in device-coordinate units

start-radians

A single float designating the starting position of the circle in radians. If this argument is NIL or omitted, it defaults to 0.0.

end-radians

A single float designating the end position of the circle in radians. If this argument is NIL or omitted, the end position is at 2*PI.

Return Value

Undefined

Corresponding MicroVMS Routine

UISDC$CIRCLE

COMPARE-BITMAPS Function

Returns T as its first value if the two bitmap arrays given as its arguments are identical in dimensions and contents, and NIL otherwise. If the two arrays are not identical, the function's second and third return values give more information about the differences. The two bitmaps are compared in row-major order, that is, with the rightmost index varying the most quickly.

This function is in package LISP.

Format

LISP:COMPARE-BITMAPS *bitmap1 bitmap2*

Arguments

bitmap1 bitmap2

Two bitmap arrays

Return Value

Three values:

1. T if the two bitmap arrays are identical in dimensions and contents, and NIL otherwise

2. T if the two bitmap arrays differ in dimensions, and the index number of the row containing the first different bit otherwise

3. The index number of the column containing the first different bit

Corresponding MicroVMS Routine

None

COPY-BITBLT Function

Creates and returns a new BITBLT object that has the same effect as the function's argument. The new BITBLT object shares destination, source, and texture bitmap arrays with its argument, but the remaining components of the argument are copied. Thus, destructive changes to the new object do not affect the original object.

This function is in package LISP.

Format

LISP:COPY-BITBLT *bitblt*

Arguments

bitblt

An object of type BITBLT

Return Value

A new object of type BITBLT

Corresponding MicroVMS Routine

None

CREATE-DISPLAY Function

Creates a UIS virtual display and returns the display object. For more informatin about virtual displays, see Section 2.3.

Format

UIS:CREATE-DISPLAY *x1 y1 x2 y2 width height*

Arguments

x1 y1

Single floats specifying the lower left corner of the virtual display's world coordinate space

x2 y2

Single floats specifying the upper right corner of the virtual display's world coordinate space *width height*

Single floats specifying, in centimeters, the default width and height of the virtual display when it appears on the output device

Return Value

An object of type DISPLAY

Corresponding MicroVMS Routine

UIS$CREATE_DISPLAY

CREATE-KB Function

Creates and returns a virtual keyboard. See Chapter 6 for information on using virtual keyboards.

Format

```
UIS:CREATE-KB &OPTIONAL device
```

Arguments

device

A character string specifying the device on which the virtual keyboard is to be created. If this argument is omitted, the device defaults to SYS$WORKSTATION.

Return Value

A keyboard

Corresponding MicroVMS Routine

UIS$CREATE_KB

CREATE-TERMINAL Function

Creates a terminal emulation window of the specified type and returns its device name string.

Format

```
UIS:CREATE-TERMINAL
     &KEY :TYPE :BANNER-TITLE
          :GENERAL-PLACEMENT :CENTER
          :ABSOLUTE-POSITION-X :ABSOLUTE-POSITION-Y
          :NOBANNER :NOBORDER :NOKB-ICON
          :NOMENU-ICON :ALIGNED
```

Arguments

:TYPE

A character string indicating the terminal type. The string can be "WT", indicating a VT100, or "TK", indicating a TEK4014. The default is a VT100 terminal emulator window.

:BANNER-TITLE

A character string specifying the terminal emulator title. The default depends on the terminal emulator type.

:GENERAL-PLACEMENT

Either :TOP, :BOTTOM, :LEFT, or :RIGHT, indicating a general preference for terminal emulator position on the screen; or a list of two of these, for example (:TOP :RIGHT); or NIL, indicating no preference for general placement. The default is NIL.

:CENTER

T or NIL. If T, the terminal emulator will be centered at the position specified by :ABSOLUTE-POSITION-X and :ABSOLUTE-POSITION-Y. If NIL, the emulator's lower-left corner will be aligned on that position. The default is NIL.

:ABSOLUTE-POSITION-X :ABSOLUTE-POSITION-Y

Two single floats indicating, in centimeters, the terminal emulator's displacement from the left and bottom edges of the display screen. The value provided with the :CENTER keyword determines the placement of the emulator relative to the :ABSOLUTE-POSITION values. By default, the emulator is not placed in an absolute screen location.

:NOBANNER

T or NIL (the default), disabling or enabling a banner above the terminal emulator

:NOBORDER

T or NIL (the default), disabling or enabling a border around the terminal emulator. If :NOBORDER T is specified, :NOBANNER is forced to T.

:NOKB-ICON

T or NIL (the default), specifying that the terminal emulator should or should not be created without a KB icon in the upper right-hand corner

:NOMENU-ICON

T or NIL (the default), specifying that the terminal emulator should or should not be created without a menu icon in the upper left-hand corner

:ALIGNED

T (the default) or NIL, specifying that tne terminal emulator's left inner edge should or should not be aligned on byte boundaries in video memory. :ALIGNED T allows text drawing optimizations.

Return Value

A character string that is the device name of the terminal emulator window

Corresponding MicroVMS Routine

UIS$CREATE_TERMINAL

CREATE-TRANSFORMATION Function

Creates a transformation into a virtual display and returns the transformation object. A transformation allows a program to write into a virtual display using a coordinate system other than that defined when the virtual display was created. A transformation object can be used in place of the *display* argument for any function that requires a virtual display as input.

See Section 2.5 for information on using transformations.

Format

UIS:CREATE-TRANSFORMATION *display x1 y1 x2 y2*
    &OPTIONAL *vd-x1 vd-y1 vd-x2 vd-y2*

Arguments

display

A display object

x1 y1 x2 y2

Four single floats designating two opposite corners of the new coordinate space

vd-x1 vd-y1 vd-x2 vd-y2

Four single floats designating, in the display's world coordinate system, a rectangle that the new coordinate space will be mapped onto. If these arguments are omitted, the new coordinate space is mapped onto the entire virtual display.

Return Value

An object of type TRANSFORMATION

Corresponding MicroVMS Routine

UIS$CREATE_TRANSFORMATION

CREATE-UIS-STRUCTURE Function

Creates and returns the LISP representation for a virtual display, a window, a virtual keyboard, or a transformation that has been created outside of LISP. You can use CALL-OUT to call a MicroVMS workstation graphics software routine or a routine written in another language. Such a routine can create a display, window, transformation, or virtual keyboard, and return the identifying integer to you. CREATE-UIS-STRUCTURE takes this identifying integer and creates from it the LISP representation of the object. This allows you to use graphics objects created outside of LISP in a LISP program.

This function takes one keyword-value pair as its arguments, where the keyword specifies the type of object you want to create and the value is the identifying integer for that object. The exception is :WINDOW. If you specify :WINDOW, you must also specify :DISPLAY with the identifying integer for the display that the window maps into.

NOTE

Using this function can place LISP in an inconsistent state. Graphic objects created using the normal VAX LISP CREATE- functions maintain some information about other graphic objects; for example, virtual displays keep a list of windows that map into them. Objects created with CREATE-UIS-STRUCTURE cannot maintain this information correctly. They may behave unpredictably with some functions.

Format

```
UIS:CREATE-UIS-STRUCTURE
     &KEY :WINDOW :DISPLAY :KEYBOARD :TRANSFORMATION
```

Arguments

:WINDOW

An integer that is the *wd_id* of the window you want to represent. You must also use the :DISPLAY keyword when you use :WINDOW.

:DISPLAY

An integer that is the *vd_id* of the display you want to represent

:KEYBOARD

An integer that is the *kb_id* of the keyboard you want to represent

:TRANSFORMATION

An integer that is the *tr_id* of the transformation you want to represent

Return Value

The LISP representation of the specified object

Corresponding MicroVMS Routine

None

CREATE-WINDOW Function

Creates a window into a virtual display and a corresponding viewport on the physical device, and returns the window. See Section 2.4 for more information about windows.

Format

```
     UIS:CREATE-WINDOW display
          &OPTIONAL x1 y1 x2 y2
          &KEY :BANNER-TITLE :NOBANNER :NOBORDER
               :NOKB-ICON :NOMENU-ICON :ALIGNED
               :INVISIBLE :VIEWPORT-WIDTH :VIEWPORT-HEIGHT
               :GENERAL-PLACEMENT :CENTER
               :ABSOLUTE-POSITION-X :ABSOLUTE-POSITION-Y
               :DEVICE
```

Arguments

display

A virtual display into which the window is to be opened

x1 y1 x2 y2

Four single floats, specifying (in world coordinates) the portion of the virtual display to be mapped into the viewport. If these arguments are omitted or NIL, the window is mapped to the default rectangle specified when the virtual display was created.

.BANNER-TITLE

A character string specifying a name to be inserted into the border of the viewport; the default is no title. If :NOBANNER T is specified, :BANNER-TITLE is ignored.

:NOBANNER

T or NIL (the default), disabling or enabling a banner above the viewport

:NOBORDER

T or NIL (the default), disabling or enabling a border around the viewport. If :NOBORDER T is specified, :NOBANNER is forced to T.

:NOKB-ICON

T or NIL (the default), specifying whether or not the viewport will be able to acquire a KB icon at a later time. Even if a viewport cannot acquire a KB icon, its window can still be associated with a virtual keyboard. (See Chapter 6.)

:NOMENU-ICON

T or NIL (the default), specifying that the viewport should or should not be created without a menu icon in the upper left-hand corner

:ALIGNED

T (the default) or NIL, specifying that the viewport's left inner edge should or should not be aligned on byte boundaries in video memory. :ALIGNED T allows text drawing optimizations.

:INVISIBLE

T or NIL (the default), specifying whether the viewport should be made visible on the display device when it is created. An "invisible" viewport can be moved onto the display device with MOVE-VIEWPORT at a later time.

:VIEWPORT-WIDTH

A single float specifying (in centimeters) the width of the viewport on the display device. If this argument is NIL or omitted, viewport width is based on the *width* argument supplied when the virtual display was created.

:VIEWPORT-HEIGHT

A single float specifying (in centimeters) the height of the viewport on the display device. If this argument is NIL or omitted, viewport height is based on the *height* argument supplied when the virtual display was created.

:GENERAL-PLACEMENT

Either :TOP, :BOTTOM, :LEFT, or :RIGHT, indicating a general preference for viewport position on the screen; or a list of two of these, for example (:TOP:RIGHT); or NIL, indicating no preference for general placement. The default is NIL.

:CENTER

NIL (the default) or T. If T, the viewport will be centered over the position specified by :ABSOLUTE-POSITION-X and :ABSOLUTE-POSITION-Y. If NIL, the viewport's lower-left corner will be aligned on the position.

:ABSOLUTE-POSITION-X :ABSOLUTE-POSITION-Y

Two single floats indicating, in centimeters, the viewport's displacement from the left and bottom edges of the display screen. The value provided with the :CENTER keyword determines the placement of the viewport relative to the ABSOLUTE-POSITION values. If these arguments are omitted or NIL, the viewport is not placed in an absolute location.

:DEVICE

A character string identifying the device on which the viewport is to be created. The default is "SYS$WORKSTATION".

Return Value

A window object

Corresponding MicroVMS Routine

UIS$CREATE_WINDOW

DELETE-DISPLAY Function

Deletes a virtual display. All associated windows and viewports are also deleted, as are transformations into the display.

Format

UIS:DELETE-DISPLAY *display*

Arguments

display

A display object

Return Value

Undefined

Corresponding MicroVMS Routine

None

DELETE-KB Function

Deletes a virtual keyboard. If the specified keyboard is bound to a window or to the physical keyboard, those bindings are terminated.

Format

UIS:DELETE-KB *keyboard*

Arguments

keyboard

A keyboard

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$DELETE_KEYBOARD

DELETE-TRANSFORMATION Function

Deletes a transformation into a virtual display. The virtual display is not affected.

Format

UIS:DELETE-TRANSFORMATION *transformation*

Arguments

transformation

A transformation object

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$DELETE_TRANSFORMATION

DELETE-WINDOW Function

Deletes a window and removes its viewport from the screen. The associated virtual display is not affected. Any window output streams that place text in the window are closed.

Format

UIS:DELETE-WINDOW *window*

Arguments

window

A window object

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$DELETE_WINDOW

DISABLE-DISPLAY-LIST Function

Disables further additions to the display list for the specified display until ENABLE-DISPLAY-LIST is executed. By default, a virtual display's display list is disabled.

See Section 3.1 for more information about the display list.

Format

UIS:DISABLE-DISPLAY-LIST *display*

Arguments

display

A virtual display

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$DISABLE_DISPLAY

DISABLE-KB Function

Disconnects the physical keyboard from the specified virtual keyboard. Any connections between the virtual keyboard and windows are unaffected.

Format

UIS:DISABLE-KB *keyboard*

Arguments

keyboard

A keyboard

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$DISABLE_KB

DISABLE-VIEWPORT-KB Function

Disables the specified window from being assigned a keyboard.

Format

UIS:DISABLE-VIEWPORT-KB *window*

Arguments

window

A window

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$DISABLE_VIEWPORT_KB

DISPLAY Type Specifier

Designates objects of type DISPLAY, created by the CREATE-DISPLAY function.

DISPLAYP Function

Returns T if its argument is a virtual display object and NIL otherwise.

Format

UIS:DISPLAYP *object*

Arguments

object

Any LISP object

Return Value

T or NIL

Corresponding MicroVMS Routine

None

DISPLAY-WINDOWS Function

Returns a list of the windows that map into a specified display.

Format

UIS:DISPLAY-WINDOWS *display*

Arguments

display

A virtual display

Return Value

A list of the windows that map into *display*

Corresponding MicroVMS Routine

None

DUMP-BITMAP Function

Writes a binary file containing the size and contents of a bitmap array. The file may later be retrieved with the LOAD-BITMAP function.

This function is in package LISP.

Format

LISP:DUMP-BITMAP *bitmap pathname*

Arguments

bitmap

A bitmap array

pathname

A pathname, string, stream, or symbol

Return Value

Undefined

Corresponding MicroVMS Routine

None

ELLIPSE Function

Draws an ellipse or a partial ellipse in a virtual display. Calls to this function draw a full ellipse unless the optional *start-radians* and *end-radians* arguments are specified. If these arguments are included, ELLIPSE draws a partial ellipse from *start-radians* counterclockwise to *end-radians*. The radian start and end positions are measured from the right-hand intersection of the ellipse and its horizontal axis.

For more information on drawing ellipses, see Section 3.5.2.

Format

UIS:ELLIPSE *display att-block center-x center-y x-radius y-radius*
&OPTIONAL *start-radians end-radians*

Arguments

display

A virtual display or transformation

att-block

A fixnum in the range 0-255, designating an attribute block from which graphics attributes will be taken

center-x center-y

Two single floats designating, in world coordinates, the center of the ellipse

x-radius y-radius

Two single floats designating, in the world coordinate system, the radius of the ellipse along its X-axis and Y-axis, respectively

start-radians

A single float designating the starting postion of the ellipse in radians. If this argument is NIL or omitted, the default is 0.0.

end-radians

A single float designating the end position of the ellipse in radians. If this argument is omitted, the default is 2*PI.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$ELLIPSE

ELLIPSE-PIXEL Function

Draws an ellipse or a partial ellipse in a window. Calls to this function draw a full ellipse unless the optional *start-radians* and *end-radians* arguments are specified. If these arguments are included, ELLIPSE-PIXEL draws a partial ellipse from *start-radians* counterclockwise to *end-radians*. The radian start and end positions are measured from the right-hand intersection of the ellipse and its horizontal axis.

For more information on drawing ellipses, see Section 3.5.2.

Format

```
UIS:ELLIPSE-PIXEL window att-block center-x center-y
     x-radius y-radius
     &OPTIONAL start-radians end-radians
```

Arguments

window

A window

att-block

A fixnum in the range 0-255, designating an attribute block from which graphics attributes will be taken

center-x center-y

Two fixnums designating, in device coordinates, the center of the ellipse

x-radius y-radius

Two fixnums designating, in device-coordinate units, the radius of the ellipse along its X-axis and Y-axis, respectively *start-radians*

A single float designating the starting postion of the ellipse in radians. If this argument is NIL or omitted, the default is 0.0.

end-radians

A single float designating the end position of the ellipse in radians. If this argument is omitted, the default is 2*PI.

Return Value

Undefined

Corresponding MicroVMS Routine

UISDC$ELLIPSE

ENABLE-DISPLAY-LIST Function

Commences or resumes additions to the display list for the specified display. By default, a virtual display's display list is disabled. (See also DISABLE-DISPLAY-LIST.)

Calls to device-coordinate functions (whose names end in -PIXEL) never add to the display list, even if it is enabled.

See Section 3.1 for more information about the display list.

Format

UIS:ENABLE-DISPLAY-LIST *display*

Arguments

display

A virtual display

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$ENABLE_DISPLAY_LIST

ENABLE-KB Function

Connects the physical keyboard to the specified virtual keyboard. If a window argument is also given, this function additionally executes the ENABLE-VIEWPORT-KB function; that is, it enables the specified window to be assigned a keyboard, then connects the virtual keyboard to the window.

Format

UIS:ENABLE-KB *keyboard* &OPTIONAL *window*

Arguments

keyboard

A keyboard

window

A window. If this argument is omitted, the virtual keyboard is not connected to any new window, although it remains connected to any window(s) to which it was previously connected.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$ENABLE_KB

ENABLE-VIEWPORT-KB Function

Enables the specified window to be assigned a keyboard, then connects the specified virtual keyboard to the window.

Format

UIS:ENABLE-VIEWPORT-KB *keyboard window*

Arguments

keyboard

A virtual keyboard to be connected to the window

window

A window

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$ENABLE_VIEWPORT_KB

END-SEGMENT Function

Ends the segment most recently begun in the specified virtual display. The values of attributes in all attribute blocks are restored to what they were when the segment was begun. (See also BEGIN-SEGMENT.)

See Section 3.7 for information about segments.

Format

UIS:END-SEGMENT *display*

Arguments

display

A virtual display

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$END_SEGMENT

ERASE Function

Erases a virtual display or a specified portion of a virtual display. The display list is updated by removing all primitives that were completely erased.

See Section 3.8 for more information about erasing in a virtual display.

Format

UIS:ERASE *display* &OPTIONAL *x1 y1 x2 y2*

Arguments

display

A virtual display or transformation

x1 y1 x2 y2

Four single floats designating the world coordinates of two opposite corners of a rectangle. Everything within the rectangle is erased. If these arguments are omitted, the entire virtual display is erased.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$ERASE

ERASE-PIXEL Function

Erases a viewport or a specified portion of a viewport. The display list is not affected by this operation.

See Section 3.8 for more information about erasing in a viewport.

Format

UIS:ERASE-PIXEL *window* &OPTIONAL *x1 y1 x2 y2*

Arguments

window

A window

x1 y1 x2 y2

Four fixnums designating the device coordinates of two opposite corners of a rectangle. Everything within the rectangle is erased. If these arguments are omitted, the entire window is erased.

Return Value

Undefined

Corresponding MicroVMS Routine

UISDC$ERASE

ERASE-VIEWING-AREA Function

Erases the viewing area of a window output stream. See Chapter 7 for information about window output streams.

Format

UIS:ERASE-VIEWING-AREA *window-output-stream*

Arguments

window-output-stream

A window output stream previously created with MAKE-WINDOW-OUTPUT-STREAM or WITH-OUTPUT-TO-WINDOW

Return Value

Undefined

Corresponding MicroVMS Routine

None

GET-ABS-POINTER-POSITION Function

Returns the absolute position of the pointer on the display screen in centimeters relative to the lower left corner of the screen.

Format

UIS:GET-ABS-POINTER-POSITION &OPTIONAL *device*

Arguments

device

A character string designating the device for which pointer information is to be returned. If omitted, this argument defaults to SYS$WORKSTATION.

Return Value

Two values:

1. A single float desionating the X position of the pointer in centimeters

2. A single float designating the Y position of the pointer in centimeters

Corresponding MicroVMS Routine

UIS$GET_ABS_POINTER_POSITION

GET-ALIGNED-POSITION Function

Returns the aligned text position for a specified virtual display. See SET-ALIGNED-POSITION for a description of the aligned position.

Format

UIS:GET-ALIGNED-POSITION *display att-block*

Arguments

display

The virtual display or transformation for which the aligned position is to be returned.

att-block

A fixnum in the range 0-255, designating an attribute block from which a font will be taken

Return Value

Two values:

1. A single float designating the X position in world coordinates

2. A single float designating the Y position in world coordinates

Corresponding MicroVMS Routine

UIS$GET_ALIGNED_POSITION

GET-ALIGNED-POSITION-PIXEL Function

Returns the aligned text position for all windows mapped into a particular virtual display. Although the argument to this function is a single window, the aligned text position is the same for all windows mapped into that window's display. See SET-ALIGNED-POSITION-PIXEL for a description of the aligned position.

Format

UIS:GET-ALIGNED-POSITION-PIXEL *window att-block*

Arguments

window

The window for which the aligned position is to be returned

att-block

A fixnum in the range 0-255, designating an attribute block from which a font will be taken

Return Value

Two values:

1. A fixnum designating the X position in device coordinates

2. A fixnum designating the Y position in device coordinates

Corresponding MicroVMS Routine

UISDC$GET_ALIGNED_POSITION

GET-ATTRIBUTE Function

Returns the value for the specified attribute in the specified attribute block. You can supply a display or a window mapped into the display as the first argument to GET-ATTRIBUTE. If you supply a window, you can get the value of any attribute. If you supply a display and you want to get the value of :CLIP-PIXEL or :LEFT-MARGIN-PIXEL, at least one window must be mapped into the display or the function will fail.

The value returned by GET-ATTRIBUTE is informational only; you cannot modify an attribute block by using SETF with this function. Use the SET-ATTRIBUTE function to modify an attribute block.

See Section 3.3 for information about attributes.

Format

UIS:GET-ATTRIBUTE *display-or-window att-block attribute*

Arguments

display-or-window

A virtual display, or a window

att-block

A fixnum in the range 0-255, representing an attribute block associated with the display

attribute

One of the attribute keywords :ARC-TYPE, :BACKGROUND-INDEX, :CHARACTER-SPACING, :CLIP, :CLIP-PIXEL, :FILL-PATTERN, :FONT, :LEFT-MARGIN, :LEFT-MARGIN-PIXEL, :LINE-STYLE, :LINE-WIDTH, :RIGHT-MARGIN, :WRITING-INDEX, :WRITING-MODE

Return Value

The value of the specified attribute (see the list of attribute keywords and their possible arguments in the description of SET-ATTRIBUTE).

- For :FONT, the return value is a list of keyword-value pairs, where each keyword is one of the font specification keywords displayed by the SHOW-FONTS function. Only those keywords whose values differ from those of the default font are included in the list.

- For :LINE-STYLE, the return value is one of the line style specification keywords if the line style is one of the DIGITAL-supplied line styles, or a bit vector of length 32 specifying a nonstandard line style. (See Section 3.5.3.3.)

- For :LINE-WIDTH, the return value is a floating-point number unless you have specified line width in world coordinate units, in which case the return value is a list of the form '(*n* :WORLD-COORDINATES), where *n* is a floating-point number.

Corresponding MicroVMS Routines

Attribute	**Routine**
:ARC-TYPE	UIS$GET_ARC_TYPE
:BACKGROUND-INDEX	UIS$GET_BACKGROUND_INDEX
:CHARACTER-SPACING	UIS$GET_CHAR_SPACING
:CLIP	UIS$GET_CLIP

Attribute	Routine
:CLIP-PIXEL	UISDC$GET_CLIP
:FILL-PATTERN	UIS$GET_FILL_PATTERN
:FONT	UIS$GET_FONT
:LEFT-MARGIN	UIS$GET_LEFT_MARGIN
:LEFT-MARGIN-PIXEL	UISDC$GET_LEFT_MARGIN
:LINE-STYLE	UIS$GET_LINE_STYLE
:LINE-WIDTH	UIS$GET_LINE_WIDTH
:WRITING-INDEX	UIS$GET_WRITING_INDEX
:WRITING-MODE	UIS$GET_WRITING_MODE

GET-ATTRIBUTE-LIST Function

Returns a property list of all the attributes and their values for a specified attribute block. You can supply a display or a window mapped into the display as the first argument to GET-ATTRIBUTE-LIST. If you supply a window, the list includes the values of all attributes. If you supply a display, the list includes the values of all attributes except :CLIP-PIXEL or :LEFT-MARGIN-PIXEL.

See Section 3.3 for information about attributes.

Format

UIS:GET-ATTRIBUTE-LIST *display-or-window att-block*

Arguments

display-or-window

A virtual display or a window

att-block

A fixnum in the range 0-255, representing an attribute block associated with the display

Return Value

A list in the form (*attr1 value1 attr2 value2* ...)

Corresponding MicroVMS Routine

None

GET-BUTTONS Function

Returns two values representing the state of the pointer buttons. The first value is an integer that encodes the actual button state; the second indicates whether the pointer cursor was visible in the window you supply as an argument.

The integer returned by GET-BUTTONS encodes the button state as follows:

- When one (and only one) button is down, the integer equals the value of the POINTER-BUTTON-*n* constant corresponding to that button.
- When more than one button is down, the integer equals the result of calling the COMMON LISP LOGAND function with the corresponding POINTER-BUTTON-*n* constants as arguments.

For example:

```
(CASE (GET-BUTTONS WINDOW)        ; Ignore second return value
  (#.POINTER-BUTTON-1 ... )       ; Only button 1 down
  (#.POINTER-BUTTON-2 ... )       ; Only button 2 down
  (#.(LOGAND
       POINTER-BUTTON-1
       POINTER-BUTTON-2) ... ))   ; Buttons 1 and 2 down
```

This example shows how you can test for single buttons and combinations of buttons in a call to the CASE macro.

Format

UIS:GET-BUTTONS *window*

Arguments

window

A window

Return Value

Two values:

1. An integer that encodes the button state as described above
2. T or NIL, indicating whether or not the pointer cursor is in *window*

Corresponding MicroVMS Routine

UIS$GET_BUTTONS

GET-COLOR Function

Returns as multiple values the R (red), G (green), and B (blue) values for an entry in the color map associated with a virtual display. See Section 3.4 for information about color.

Format

UIS:GET-COLOR *display color-id* &OPTIONAL *window*

Arguments

display

A virtual display

color-id

An integer specifying an entry in the color map associated with *display*

window

A window. If this argument is supplied, the return values are the realized colors for the specific device on which the window was created.

Return Value

Three values:

1. The red value
2. The blue value
3. The green value

Each return value is a floating-point number in the range 0.0 - 1.0, inclusive.

Corresponding MicroVMS Routine

UIS$GET_COLOR

GET-DISPLAY-SIZE Function

Returns the size and resolution of the display screen.

Format

UIS:GET-DISPLAY-SIZE &OPTIONAL *device*

Arguments

device

A string specifying the device for which information is to be returned. The default is SYS$WORKSTATION.

Return Value

Six values:

1. A single float giving the width of the screen in centimeters
2. A single float giving the height of the screen in centimeters
3. A single float giving the horizontal resolution of the screen in pixels/centimeter
4. A single float giving the vertical resolution of the screen in pixels/centimeter
5. A fixnum giving the width of the screen in pixels
6. A fixnum giving the height of the screen in pixels

Corresponding MicroVMS Routine

UIS$GET_DISPLAY_SIZE

GET-FONT-SIZE Function

Returns the height and width of a text string in centimeters for a specified font.

Format

UIS:GET-FONT-SIZE *font-id text-string*

Arguments

font-id

A pathname, string, stream, or symbol specifying a font file; or, a list of keyword-value pairs specifying a font. (See the description of the SET-ATTRIBUTE function, :FONT attribute, for an explanation of this list.)

text-string

A string to be measured

Return Value

Two values:

1. A single float designating the width of the character string in centimeters

2. A single float designating the height of the character string in centimeters

Corresponding MicroVMS Routine

UIS$GET_FONT_SIZE

GET-INTENSITY Function

Returns the equivalent monochrome intensity value for an entry in the color map associated with a virtual display. See Section 3.4 for information about color maps.

Format

UIS:GET-INTENSITY *display color-id* &OPTIONAL *window*

Arguments

display

A virtual display

color-id

An integer specifying an entry in the color map associated with *display*

window

A window. If this argument is specified, the return value is the realized intensity for the specific device on which the window was created.

Return Value

A floating-point number in the range 0.0 - 1.0, inclusive

Corresponding MicroVMS Routine

UIS$GET_INTENSITY

GET-KB-ATTRIBUTE Function

Returns the value of a single keyboard attribute for a specified virtual keyboard. You cannot modify the value of a keyboard attribute by using SETF with this function. Use the SET-KB-ATTRIBUTES function to modify a virtual keyboard.

See Section 6.1.4 for information about keyboard attributes.

Format

UIS:GET-KB-ATTRIBUTE *keyboard attribute*

Arguments

keyboard

A virtual keyboard

attribute

One of the keyboard attribute keywords listed under the description of SET-KB-ATTRIBUTES

Return Value

The value of the specified attribute for the specified keyboard

Corresponding MicroVMS Routine

None

GET-KB-ATTRIBUTE-LIST Function

Returns a list of keyword-value pairs indicating the value of each attribute for the specified keyboard. This list is informational only; you cannot modify a virtual keyboard by using SETF with this function. Use the SET-KB-ATTRIBUTES function to modify a virtual keyboard.

See Section 6.1.4 for information about keyboard attribues.

Format

UIS:GET-KB-ATTRIBUTE-LIST *keyboard*

Arguments

keyboard

A keyboard

Return Value

A list of keyword-value pairs, where the keyword is one of the keyboard attribute keywords listed in the description of SET-KB-ATTRIBUTES, and the value is its value for this keyboard

Corresponding MicroVMS Routine

UIS$GET_KB_ATTRIBUTES

GET-POINTER-POSITION Function

Returns the world coordinate position of the pointer cursor, or NIL if the pointer cursor is not in the visible portion of the viewport.

Format

UIS:GET-POINTER-POSITION *display window*

Arguments

display

A virtual display, transformation, or NIL. The virtual display must be the one into which *window* is mapped; NIL is equivalent to specifying this display. A transformation must be one that is mapped into that display. Specifying a transformation allows you to interpret the return values as transformation coordinates rather than world coordinates.

window

A window

Return Value

Two values:

1. A single float designating the X position of the pointer cursor in world or transformation coordinates, or NIL if the pointer cursor is not in the viewport

2. A single float designating the Y position of the pointer cursor in world or transformation coordinates

Corresponding MicroVMS Routine

UIS$GET_POINTER_POSITION

GET-POINTER-POSITION-PIXEL Function

Returns the device-coordinate position of the pointer cursor, or NIL if the pointer cursor is not in the visible portion of the viewport.

Format

UIS:GET-POINTER-POSITION-PIXEL *window*

Arguments

window

A window

Return Value

Two values:

1. A fixnum designating the X position of the pointer cursor in device coordinates, or NIL if the pointer cursor is not in the viewport
2. A fixnum designating the Y position of the pointer cursor in device coordinates

Corresponding MicroVMS Routine

UISDC$GET_POINTER_POSITION

GET-POSITION Function

Returns the current text position for a specified virtual display. See Section 3.6.2.1 for information about the text position.

Format

UIS:GET-POSITION *display*

Arguments

display

The virtual display or transformation for which the text position is to be returned

Return Value

Two values:

1. A single float representing the X portion of the world-coordinate text position

2. A single float representing the Y portion of the world-coordinate text position

Corresponding MicroVMS Routine

UIS$GET_POSITION

GET-POSITION-PIXEL Function

Returns the current device-coordinate text position for windows mapped into a virtual display. The device-coordinate text position is independent of the display text position; however, all windows mapped into a virtual display share the same device-coordinate text position.

See Section 3.6.2.1 for information about the device-coordinate text position.

Format

UIS:GET-POSITION-PIXEL *window*

Arguments

window

The window for which the text position is to be returned

Return Value

Two values:

1. A fixnum representing the X portion of the device-coordinate text position

2. A fixnum representing the Y portion of the device-coordinate text position

Corresponding MicroVMS Routine

UISDC$GET_POSITION

GET-VIEWPORT-POSITION Function

Returns the position of the lower-left corner of a display viewport in relation to the lower-left corner of the physical display, in centimeters.

Format

UIS:GET-VIEWPORT-POSITION *window*

Arguments

window

The window associated with the viewport whose position is to be returned

Return Value

Multiple values:

1. A single float representing the X position of the viewport in centimeters

2. A single float representing the Y position of the viewport in centimeters

Corresponding MicroVMS Routine

UIS$GET_VIEWPORT_POSITION

GET-VIEWPORT-SIZE Function

Returns the width and height of a display viewport in centimeters.

Format

UIS:GET-VIEWPORT-SIZE *window*

Arguments

window

The window associated with the viewport whose size is to be returned

Return Value

Multiple values:

1. A single float representing the width of the viewport

2. A single float representing the height of the viewport

Corresponding MicroVMS Routine

UIS$GET_VIEWPORT_SIZE

GET-VISIBILITY Function

Returns T if an entire viewport or all of a specified portion thereof is unoccluded by other viewports, and NIL if any part of it is occluded. If no optional arguments are given, the entire viewport is checked for visibility. If a single world-coordinate pair is given, then that single point is checked. If two world-coordinate pairs are given, then the rectangle they define is checked. If the point or rectangle falls outside the window, then GET-VISIBILITY returns NIL.

Format

UIS:GET-VISIBILITY *display window* &OPTIONAL *x1 y1 x2 y2*

Arguments

display

A virtual display, transformation, or NIL. The virtual display must be the one into which *window* is mapped; NIL is equivalent to specifying this display. A transformation must be one that is mapped into that display. Specifying a transformation allows you to use transformation coordinates instead of world coordinates.

window

The window whose associated viewport (or portion thereof) is to be checked for visibility.

x1 y1

Two single floats defining a point in world or transformation coordinate space; this point is checked for visibility if no more arguments are given

x2 y2

Two single floats defining another point in world or transformation coordinate space; the rectangle defined by the two points is checked for visibility

Return Value

T or NIL

Corresponding MicroVMS Routine

UIS$GET_VISIBILITY

GET-VISIBILITY-PIXEL Function

Returns T if an entire viewport or all of a specified portion thereof is unoccluded by other viewports, and NIL if any part of it is occluded. If no optional arguments are given, the entire viewport is checked for visibility. If a single device-coordinate pair is given, then that single point is checked. If two device-coordinate pairs are given, then the rectangle they define is checked. If the point or rectangle falls outside the window, then GET-VISIBILITY-PIXEL returns NIL.

Format

UIS:GET-VISIBILITY-PIXEL *window* &OPTIONAL *x1 y1 x2 y2*

Arguments

window

The window whose associated viewport (or portion thereof) is to be checked for visibility.

x1 y1

Two fixnums defining a pixel in the window; this pixel is checked for visibility if no more arguments are given

x2 y2

Two fixnums defining another pixel in the window; the rectangle defined by the two points is checked for visibility

Return Value

T or NIL

Corresponding MicroVMS Routine

UISDC$GET_VISIBILITY

GET-WINDOW-ATTRIBUTE-LIST Function

Returns a disembodied property list of the attributes of a viewport. This list is informational only; you cannot modify a viewport by using SETF with this function. Each of these attributes is established by the call to CREATE-WINDOW that creates the viewport. The attributes, and the values of their meanings, are:

:NOBANNER -- T indicates that the viewport has no banner; NIL indicates that the viewport has a banner.
:NOBORDER -- T indicates that the viewport has no border; NIL indicates that the viewport has a border.
:NOMENU-ICON -- T indicates that the viewport has no menu icon in its banner; NIL indicates that the viewport has a menu icon.
:NOKB-ICON -- T indicates that the viewport cannot acquire a keyboard icon in its banner; NIL indicates that the viewport can acquire a keyboard icon. A viewport that cannot acquire a keyboard icon can still have a virtual keyboard associated with it.
:ALIGNED -- T indicates that the viewport's left inner edge is aligned on byte boundaries in video memory; NIL indicates that the viewport is not so aligned.

Format

UIS:GET-WINDOW-ATTRIBUTE-LIST *window*

Arguments

window

A window

Return Value

A disembodied property list

Corresponding MicroVMS Routine

UIS$GET_WINDOW_ATTRIBUTES

GET-WS-COLOR Function

Returns as multiple values the R (red), G (green), and B (blue) values for a specified workstation standard color that was in effect at the time the virtual display was created.

See Section 3.4 for information about color.

Format

UIS:GET-WS-COLOR *display color-id* &OPTIONAL *window*

Arguments

display

A virtual display

color-id

An integer specifying an entry in the workstation standard color map that was in effect when *display* was created

window

A window. If this argument is specified, the return values are the realized colors for the specific device on which the window was created.

Return Value

Three values:

1. The red value

2. The blue value

3. The green value

Each return value is a floating-point number in the range 0.0 - 1.0, inclusive.

Corresponding MicroVMS Routine

UIS$GET_WS_COLOR

GET-WS-INTENSITY Function

Returns the equivalent monochrome intensity value for a specified workstation standard color that was in effect at the time the virtual display was created. See Section 3.4 for information about color.

Format

UIS:GET-WS-INTENSITY *display color-id* &OPTIONAL *window*

Arguments

display

A virtual display

color-id

An integer specifying an entry in the workstation standard color map that was in effect when *display* was created *window*

A window. If this argument is specified, the return value is the realized intensity for the specific device for which the window was created.

Return Value

A floating-point number in the range 0.0 - 1.0, inclusive

Corresponding MicroVMS Routine

UIS$GET_WS_INTENSITY

IMAGE Function

Writes a bitmap to a rectangle in a virtual display. If the display list is enabled, IMAGE adds the bitmap to the display list.

The bitmap is displayed in viewports as follows:

- If the size of the bitmap is larger than the corresponding rectangle in a viewport, the bitmap is clipped at the top and right to fit the rectangle.

- If the size of the bitmap is smaller than the corresponding rectangle in a viewport, the bitmap is scaled up to the largest integral multiple that will still fit within the rectangle. If the rectangle is not an exact integer multiple of the bitmap, the excess on the top and right of the rectangle is left unchanged.

The horizontal and vertical dimensions of the bitmap are scaled or clipped independent of each other. That is, a bitmap may be scaled by different factors in the horizontal and vertical dimensions, or may be clipped in one dimension and scaled in the other.

See Chapter 6 for information about bitmaps and screen images.

Format

UIS:IMAGE *display att-block bitmap x1 y1 x2 y2*

Arguments

display

A virtual display

att-block

A fixnum in the range 0-255, designating an attribute block associated with *display*

bitmap

An array of unsigned bytes

x1 y1 x2 y2

Four single floats specifying, in world coordinates, two opposite corners of the rectangle in which the image is to be placed

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$IMAGE

IMAGE-PIXEL Function

Writes a bitmap to a viewport. See Chapter 4 for information about screen images and bitmaps.

Format

UIS:IMAGE-PIXEL *window att-block bitmap x y*

Arguments

window

A window

att-block

A fixnum in the range 0-255, designating an attribute block associated with *window*'s display

bitmap

An array of unsigned bytes

x y

Two fixnums specifying, in device coordinates, the lower left corner of the rectangle in which the image is to be placed. The size of the bitmap determines the size of the rectangle.

Return Value

Undefined

Corresponding MicroVMS Routine

UISDC$IMAGE

KEYBOARD Type Specifier

Designates objects of type KEYBOARD, created by the CREATE-KB function.

KEYBOARDP Function

Returns T if its argument is a keyboard and NIL otherwise.

Format

UIS:KEYBOARDP *object*

Arguments

object

A LISP object

Return Value

T or NIL

Corresponding MicroVMS Routine

None

K-TRM-xxx Constants

These constants have integer values, each of which represents one of the function, numeric keypad, editing, or arrow keys. The return value of of READ-KB-CHAR and the first argument passed to an interrupt function established with SET-KB-ACTION can be compared with these values. Note that all of these constants are in a package called SMG.

Key	Constant
Numeric Keypad Keys	
keypad 0	SMG:K-TRM-KP0
keypad 1	SMG:K-TRM-KP1
keypad 2	SMG:K-TRM-KP2
keypad 3	SMG:K-TRM-KP3
keypad 4	SMG:K-TRM-KP4
keypad 5	SMG:K-TRM-KP5
keypad 6	SMG:K-TRM-KP6
keypad 7	SMG:K-TRM-KP7
keypad 8	SMG:K-TRM-KP8
keypad 9	SMG:K-TRM-KP9
keypad -	SMG:K-TRM-MINUS
keypad ,	SMG:K-TRM-COMMA
keypad .	SMG:K-TRM-PERIOD
keypad Enter	SMG:K-TRM-ENTER
keypad PF1	SMG:K-TRM-PF1
keypad PF2	SMG:K-TRM-PF2
keypad PF3	SMG:K-TRM-PF3
keypad PF4	SMG:K-TRM-PF4
Function, Help, and Do Keys	
F6	SMG:K-TRM-F6
F7	SMG:K-TRM-F7
F8	SMG:K-TRM-F8
F9	SMG:K-TRM-F9
F10	SMG:K-TRM-F10
F11	SMG:K-TRM-F11
F12	SMG:K-TRM-F12
F13	SMG:K-TRM-F13
F14	SMG:K-TRM-F14
F15 (Help)	SMG:K-TRM-HELP
F16 (Do)	SMG:K-TRM-DO
F17	SMG:K-TRM-F17
F18	SMG:K-TRM-F18
F19	SMG:K-TRM-F19
F20	SMG:K-TRM-F20
Editing and Arrow Keys	
E1 (Find)	SMG:K-TRM-FIND
E2 (Insert Here)	SMG:K-TRM-INSERT-HERE
E3 (Remove)	SMG:K-TRM-REMOVE
E4 (Select)	SMG:K-TRM-SELECT
E5 (Prev Screen)	SMG:K-TRM-PREV-SCREEN
E6 (Next Screen)	SMG:K-TRM-NEXT-SCREEN
Up Arrow	SMG:K-TRM-UP
Down Arrow	SMG:K-TRM-DOWN
Right Arrow	SMG:K-TRM-RIGHT
Left Arrow	SMG:K-TRM-LEFT

LIST-ALL-DISPLAYS Function

Returns a list of all user-created displays and transformations that have not been deleted.

Format

```
UIS:LIST-ALL-DISPLAYS
```

Arguments

None

Return Value

A list

Corresponding MicroVMS Routine

None

LIST-ALL-WINDOWS Function

Returns a list of all user-created windows that have not been deleted.

Format

```
UIS:LIST-ALL-WINDOWS
```

Arguments

None

Return Value

A list

Corresponding MicroVMS Routine

None

LOAD-BITMAP Function

Returns a bitmap array read from a file that was previously written by the DUMP-BITMAP function.

This function is in package LISP.

Format

LISP:LOAD-BITMAP *pathname*

Arguments

pathname

A pathname, string, stream, or symbol; the file must have been written previously by the DUMP-BITMAP function

Return Value

A bitmap array

Corresponding MicroVMS Routine

None

MAKE-BITBLT Function

Creates and returns an object of type BITBLT. When used as an argument to the BITBLT function, a BITBLT object causes a specific operation to be performed on a specific bitmap. A BITBLT object has the following components:

- A destination bitmap, that is, the bitmap to which the operation is applied. The destination bitmap can be an entire bitmap array or a specified rectangle of a bitmap array.

- A source bitmap, that is, the bitmap that is applied to the destination bitmap. The source bitmap can be an entire bitmap array or can extend upward and to the right from a specified pixel in a bitmap array. The source bitmap is not altered by BITBLT, unless it happens to overlap with the destination bitmap.

- A texture bitmap. The texture bitmap is combined with the source bitmap before the source bitmap is applied to the destination bitmap. It is constrained to be a bitmap array that is up to 32 bits wide; it can have any height. A texture bitmap wider than 32 bits is trimmed on the right to fit. The texture bitmap itself is not altered by BITBLT.

- Source and destination operations. These specify how the texture bitmap is combined with the source bitmap, and the texture-source result with the destination, respectively. They can be any of the operations that can be specified with BOOLE.

The operation described by a BITBLT object in combination with the BITBLT function is the following:

1. The destination, source, and texture bitmaps are evaluated. They must all have elements of the same type.

2. For each pixel in the destination bitmap:

   a. The corresponding pixel in the source bitmap is combined with the corresponding pixel in the texture bitmap according to the source operation, with the source pixel as the first argument and the texture pixel as the second argument. If the pixel coordinates exceed the limits of the texture bitmap, the texture pixel is located by wrapping around to the left and/or upper edge of the texture bitmap. This has the effect of replicating the texture bitmap across the source bitmap.

   b. The result of combining the source and texture pixels is combined with the destination pixel according to the destination operation, with the source-texture result as the first argument and the destination pixel as the second argument.

   c. The result of combining the source-texture result with the destination pixel is used to modify the destination pixel.

A BITBLT operation is described by the following algorithm (ignoring array bounds, and assuming heights and widths are the same):

```
(DO ((dj dst-y (1+ dj))
     (sj src-y (1+ sj)))
    ((= dj (+ dst-y dst-h)))
  (DO ((di dst-x (1+ di))
       (si src-x (1+ si)))
      ((= di (+ dst-x dst-w)))
    (SETF (AREF destination dj di)
          (BOOLE dst-op
                 (BOOLE src-op
                        (AREF source sj si)
                        (AREF filled-texture
                              (REM dj texture-height)
                              (REM di 32)))
                 (AREF destination dj di)))))
```

This function is in package LISP.

See Section 4.6 for more information on using this function.

Format

```
LISP:MAKE-BITBLT
     &KEY :DESTINATION :DST-X :DST-Y :DST-W :DST-H :DST-OP
          :SOURCE :SRC-X :SRC-Y :SRC-W :SRC-H :SRC-OP
          :TEXTURE
```

Arguments

:DESTINATION

A bitmap array. If none of the next four arguments are given, the entire array is used as the target of the operation.

:DST-X :DST-Y

Two integers specifying the array reference (*not* the device coordinates) of a pixel in the destination bitmap. The destination rectangle of the operation extends downward and to the right from this point. If these arguments are NIL or omitted they default to 0 (the upper-left corner of the destination bitmap).

:DST-W :DST-H

Two integers specifying the width and height of the destination rectangle in pixels. If these arguments are NIL or omitted, they default to the full width and height of the destination bitmap, minus the values given with :DST-X and :DST-Y.

:DST-OP

Any of the operations that can be specified with the BOOLE function (see Section 12.7 of *COMMON LISP: The Language*). The operation specifies how each pixel in the source bitmap is combined with the corresponding pixel in the destination bitmap. The following constants are particularly useful with :DST-OP:

- BOOLE-1 - replaces the destination with the source
- BOOLE-IOR - "paints" the destination with the source
- BOOLE-XOR - inverts any destination pixel that is also occupied by a source pixel
- BOOLE-ANDC1 - erases any destination pixel that corresponds to an illuminated pixel in the source

If this argument is NIL or omitted, the default operation is BOOLE-1.

:SOURCE

A bitmap array. Its elements must have the same type (that is, number of bits per element) as those of the :DESTINATION bitmap.

If this argument is NIL or omitted, it defaults to the destination bitmap.

:SRC-X :SRC-Y

Two integers specifying the array reference (*not* the device coordinates) of a pixel in the source bitmap. The source rectangle for the operation extends downward and to the right from this point. If these arguments are NIL or omitted they default to 0 (the upper-left corner of the source bitmap).

:SRC-W :SRC-H

Two integers specifying the width and height of the source rectangle in pixels. If these arguments are NIL or omitted they default to the full width and height of the destination bitmap, minus the values given with :SRC-X and :SRC-Y.

:SRC-OP

Any of the operations that can be specified with the BOOLE function (see Section 12.7 of *COMMON LISP: The Language*). The operation specifies how each pixel in the source bitmap is combined with the corresponding pixel in the texture bitmap. The following operations are particularly useful with :SRC-OP:

- BOOLE-1 - ignores the texture bitmap and uses the source bitmap
- BOOLE-2 - ignores the source bitmap and uses the texture bitmap
- BOOLE-AND - merges the source and texture bitmaps by only setting pixels in the result that are set in both the source and texture

If this argument is NIL or omitted, it defaults to BOOLE-1.

:TEXTURE

A bitmap array. Its elements must have the same type (that is, number of bits per element) as those of the :DESTINATION bitmap. If the specified :TEXTURE bitmap is narrower than 32 bits, it is replicated horizontally to fill out a rectangle 32 bits wide. If the bitmap is wider than 32 bits, it is trimmed on the right to fit.

The graphics system copies the bitmap array you specify with :TEXTURE. Therefore, if you later modify this array, you must use SETF on the BITBLT-TEXTURE function to make that change effective in a particular BITBLT object.

If you do not specify a texture bitmap, it defaults to all 1s.

Return Value

An object of type BITBLT

Corresponding MicroVMS Routine

None

MAKE-BITMAP Function

Creates and returns a two-dimensional array of the specified dimensions, each of whose elements is an unsigned byte. The length of each unsigned byte is given by the *bits-per-pixel* argument, which defaults to 1. This function has the same effect as:

```
(MAKE-ARRAY (LIST height width)
            :ELEMENT-TYPE `(UNSIGNED-BYTE ,bits-per-pixel)
            :ALLOCATION space)
```

This function is in package LISP.

See Chapter 4 for information about creating and altering bitmaps.

Format

LISP:MAKE-BITMAP *width height* &OPTIONAL *bits-per-pixel space*

Arguments

width height

Two integers specifying the width and height of the requested bitmap in pixels

bits-per-pixel

An integer specifying how many bits represent each pixel. If this argument is omitted, one bit is used to represent each pixel. This argument is currently limited to 1.

space

Either :DYNAMIC (the default) or :STATIC, indicating in which space the array should be created. (See the description of MAKE-ARRAY in the *VAX LISP/VMS User's Guide*.)

Return Value

An array of unsigned bytes

Corresponding MicroVMS Routine

None

MAKE-WINDOW-OUTPUT-STREAM Function

Creates a window output stream and returns the stream object. The stream can be used as an argument to COMMON LISP output functions.

See Chapter 7 for information about window output streams.

Format

```
UIS:MAKE-WINDOW-OUTPUT-STREAM window
     &KEY :ATTRIBUTE-BLOCK :VIEWING-AREA
          :HORIZONTAL-OVERFLOW :VERTICAL-OVERFLOW
```

Arguments

window

A window

:ATTRIBUTE-BLOCK

A fixnum designating the attribute block to be used for all output operations; the default is attribute block 0. The attribute block can be changed later by using the WINDOW-STREAM-ATTRIBUTE-BLOCK function.

:VIEWING-AREA

A list of four integers in the form (*x1 y1 x2 y2*) that specifies the lower-left and upper-right corners of a rectangle within which text is written. If this argument is NIL or omitted, text is written into the entire window. The viewing area can be changed later by using the WINDOW-STREAM-VIEWING-AREA function.

:HORIZONTAL-OVERFLOW

Either :WRAP, the default, indicating that text be wrapped onto the next line, or :TRUNCATE, indicating that text be truncated at the right margin. The horizontal overflow behavior can be changed later by using the WINDOW-STREAM-HORIZONTAL-OVERFLOW function.

:VERTICAL-OVERFLOW

Eitehr :SCROLL, the default, indicating that text be scrolled upwards to accomodate new text at the bottom of the viewing area; :WRAP, indicating that text be vertically wrapped; or :TRUNCATE, indicating that text below the bottom of the viewing area be discarded until the viewing area is erased. The vertical overflow behavior can be changed later by using the WINDOW-STREAM-VERTICAL-OVERFLOW function.

Return Value

A window output stream

Corresponding MicroVMS Routine

None

MEASURE-TEXT Function

Returns the width and height of a text string in the world coordinate system of a virtual display.

Format

UIS:MEASURE-TEXT *display att-block text-string*

Arguments

display

A virtual display or transformation

att-block

A fixnum in the range 0-255, designating an attribute block from which font and character spacing attributes will be taken

text-string

A character string to be measured

Return Value

Two values:

1. A single float designating the width of the text string in world coordinates

2. A single float designating the height of the text string in world coordinates

Corresponding MicroVMS Routine

UIS$MEASURE_TEXT

MEASURE-TEXT-PIXEL Function

Returns the width and height of a text string in device-coordinate units.

Format

```
UIS:MEASURE-TEXT-PIXEL window att-block text-string
```

Arguments

window

A window

att-block

A fixnum in the range 0-255, designating an attribute block from which font and character spacing attributes will be taken

text-string

A character string to be measured

Return Value

Two values:

1. A fixnum designating the width of the text string in device-coordinate units
2. A fixnum designating the height of the text string in device-coordinate units

Corresponding MicroVMS Routine

UISDC$MEASURE_TEXT

MOVE-AREA Function

Shifts a portion of a virtual display from one place to another in the display. The display list is updated to reflect the alteration.

See Section 3.8 for information about moving portions of a virtual display.

Format

```
UIS:MOVE-AREA display x1 y1 x2 y2 dest-x dest-y
```

Arguments

display

A virtual display or transformation

x1 y1 x2 y2

Four single floats designating the world coordinates of two opposite corners of a rectangle

dest-x dest-y

Two single floats designating the world coordinates of the lower-left corner of the destination rectangle. The size of the destination rectangle is taken from the size of the source rectangle.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$MOVE_AREA

MOVE-AREA-PIXEL Function

Shifts a rectangle in a viewport to another place in the viewport. The display list is not affected.

See Section 3.8 for information about moving portions of a viewport.

Format

UIS:MOVE-AREA-PIXEL *window x1 y1 x2 y2 dest-x dest-y*

Arguments

window

A window

x1 y1 x2 y2

Four fixnums designating the device coordinates of two opposite corners of a rectangle *dest-x dest-y*

Two fixnums designating the device coordinates of the lower-left corner of the destination rectangle. The size of the destination rectangle is taken from the size of the source rectangle.

Return Value

Undefined

Corresponding MicroVMS Routine

UISDC$MOVE_AREA

MOVE-VIEWPORT Function

Moves a display viewport on the physical display.

Format

```
UIS:MOVE-VIEWPORT window
     &KEY :GENERAL-PLACEMENT :CENTER
          :ABSOLUTE-POSITION-X :ABSOLUTE-POSITION-Y
          :INVISIBLE
```

Arguments

window

The window whose associated viewport is to be moved

:GENERAL-PLACEMENT

Either :TOP :BOTTOM, :LEFT, or :RIGHT, indicating a general preference for viewport position on the screen; or a list of two of these, for example (:TOP :RIGHT); or NIL, indicating no preference for viewport placement

:CENTER

NIL (the default) or T. If T, the viewport will be centered over the position specified by :ABSOLUTE-POSITION-X and :ABSOLUTE-POSITION-Y. If NIL, the viewport's lower left corner will be aligned on the position.

:ABSOLUTE-POSITION-X :ABSOLUTE-POSITION-Y

Two single floats indicating, in centimeters, the viewport's new displacement from the left and bottom edges of the display screen. The value provided with the :CENTER keyword determines the placement of the viewport relative to the ABSOLUTE-POSITION values.

:INVISIBLE

Either T or NIL (the default). If T, the viewport is moved to a location off the screen; it becomes invisible.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$MOVE_VIEWPORT

MOVE-WINDOW Function

Moves a window within a virtual display, optionally allowing the window to change size and/or aspect ratio. See Section 2.4.3 for information about using this function.

Format

UIS:MOVE-WINDOW *display window x1 y1 x2 y2*

Arguments

display

A virtual display, transformation, or NIL. The virtual display must be the one into which *window* is mapped; NIL is equivalent to specifying this display. A transformation must be one that is mapped into that display. Specifying a transformation allows you to use transformation coordinates instead of world coordinates.

window

A window object

x1 y1 x2 y2

Four single floats specifying (in world or transformation coordinates) the lower-left and upper-right corners of the new window

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$MOVE_WINDOW

NEW-TEXT-LINE Function

Moves the current text position to the left margin and down by the height of one line.

Format

UIS:NEW-TEXT-LINE *display att-block*

Arguments

display

The virtual display or transformation in which the operation is to be performed

att-block

A fixnum in the range 0-255, designating an attribute block from which font, left margin, and line spacing attributes are taken

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$NEW_TEXT_LINE

NEW-TEXT-LINE-PIXEL Function

Moves the current text position to the left margin and down by the height of one line.

Format

UIS:NEW-TEXT-LINE-PIXEL *window att-block*

Arguments

window

The window in which the operation is to be performed *att-block*

A fixnum in the range 0-255, designating an attribute block from which font, left margin, and line spacing attributes are taken

Return Value

Undefined

Corresponding MicroVMS Routine

UISDC$NEW_TEXT_LINE

PLOT Function

Draws a point, a single line, or up to 124 lines, depending on the number of positions specified. Use the PLOT-ARRAY function to draw more than 124 lines in a single operation. If the attribute block specifies a fill pattern, the PLOT function does not draw lines, but instead fills the area that the lines contain.

See Section 3.5.1 for information about drawing lines. See Section 3.5.3.2 for information about specifying a fill pattern.

Format

UIS:PLOT *display att-block x1 y1*
&OPTIONAL *x2 y2 x3 y3 ... x125 y125*

Arguments

display

A virtual display or transformation

att-block

A fixnum in the range 0-255, designating an attribute block from which the graphics attributes will be taken

x1 y1

Two single floats designating a world coordinate. If this is the only point specified, this single point will be plotted.

x2 y2 x3 y3 ... x125 y125

Single floats designating additional world coordinates. A line is drawn between each point specified and the previous point.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$PLOT

PLOT-ARRAY Function

Draws a point, line, or multiple connected lines, depending on the number of points specified. It differs from PLOT in that you specify the points in the form of two vectors, one for the X value and one for the Y value.

If the attribute block specifies a fill pattern, the PLOT-ARRAY function does not draw lines, but instead fills the area that the lines contain.

See Section 3.5.1 for information about drawing lines. See Section 3.5.3.2 for information about specifying a fill pattern.

Format

UIS:PLOT-ARRAY *display att-block x-vector y-vector*
&OPTIONAL *count*

Arguments

display

A virtual display or transformation

att-block

A fixnum in the range 0-255, designating an attribute block from which graphics attributes will be taken

x-vector y-vector

Two specialized vectors with elements of type SINGLE-FLOAT. (If you supply general vectors, VAX LISP creates specialized vectors, with a resulting loss of efficiency.) If a *count* argument is supplied, each vector must be of length *count* or greater. Points are specified by taking an element from *x-vector* and *y-vector* and interpreting them as the X and Y values of a world coordinate. Successive points are joined by lines.

count

A fixnum designating the number of points to be drawn. If this argument is not supplied, it defaults to the minimum of the number of elements in *x-vector* and *y-vector*.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$PLOT_ARRAY

PLOT-ARRAY-PIXEL Function

Draws a point, line, or multiple connected lines, depending on the number of points specified. It differs from PLOT in that you specify the points in the form of two vectors, one for the X value and one for the Y value.

If the attribute block specifies a fill pattern, the PLOT-ARRAY-PIXEL function does not draw lines, but instead fills the area that the lines contain.

See Section 3.5.1 for information about drawing lines. See Section 3.5.3.2 for information about specifying a fill pattern.

Format

UIS:PLOT-ARRAY-PIXEL *window att-block x-vector y-vector*
&OPTIONAL *count*

Arguments

window

A window

att-block

A fixnum in the range 0-255, designating an attribute block from which graphics attributes will be taken

x-vector y-vector

Two specialized vectors with elements of type (SIGNED-BYTE 32). If a *count* argument is supplied, each vector must be of length *count* or greater. Points are specified by taking an element from *x-vector* and *y-vector* and interpreting them as the X and Y values of a device coordinate. Successive points are joined by lines.

count

A fixnum designating the number of points to be drawn. If this argument is not supplied, it defaults to the minimum of the number of elements in *x-vector* and *y-vector*.

Return Value

Undefined

Corresponding MicroVMS Routine

UISDC$PLOT_ARRAY

PLOT-PIXEL Function

Draws a point, a single line, or up to 124 lines, depending on the number of positions specified. Use the PLOT-ARRAY-PIXEL function to draw more than 124 lines in a single operation.

If the attribute block specifies a fill pattern, the PLOT-PIXEL function does not draw lines, but instead fills the area that the lines contain.

See Section 3.5.1 for information about drawing lines. See Section 3.5.3.2 for information about specifying a fill pattern.

Format

UIS:PLOT-PIXEL *window att-block x1 y1*
        &OPTIONAL *x2 y2 x3 y3 ... x125 y125*

Arguments

window

A window

att-block

A fixnum in the range 0-255, designating an attribute block from which graphics attributes will be taken

x1 y1

Two fixnums designating a device coordinate. If this is the only point specified, this single point will be plotted.

x2 y2 x3 y3 ... x125 y125

Fixnums designating additional device coordinates. A line is drawn between each point specified and the previous point.

Return Value

Undefined

Corresponding MicroVMS Routine

UISDC$PLOT

POINTER-BUTTON-n Constants

Have values that specify a button on the pointing device. An interrupt function that is given as the *action* argument in SET-BUTTON-ACTION or SET-BUTTON-ACTION-PIXEL receives, as its first argument, the code of the button whose transition invoked the function. Use these constants to compare with that code. You also use these constants to interpret the value returned by the GET-BUTTONS function. The constants start with POINTER-BUTTON-1 and go to an *n* as high as required for the supported pointing device with the greatest number of buttons.

See Section 5.1.3 for information about getting input from pointer buttons.

Format

UIS:POINTER-BUTTON-*n*

POP-VIEWPORT Function

Pops a display viewport to the forefront of the display screen, in front of any other viewports which may have been occluding it.

Format

UIS:POP-VIEWPORT *window*

Arguments

window

A window whose associated display viewport is to be popped.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$POP_VIEWPORT

PUSH-VIEWPORT Function

Pushes a display viewport to the back of the display screen, in back of any other viewports which may have been occluded by it.

Format

UIS:PUSH-VIEWPORT *window*

Arguments

window

The window whose associated display viewport is to be pushed

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$PUSH_VIEWPORT

READ-IMAGE-PIXEL Function

Reads a bitmap from a rectangle in a window on the screen and either creates a new bitmap representing that image or alters an existing bitmap. In either case, the function returns the bitmap.

If the *bitmap* argument to READ-IMAGE-PIXEL is NIL, the function creates a new bitmap that is the size of the screen rectangle you specify. If you do not specify a rectangle, the bitmap is taken from the entire viewport.

If the *bitmap* argument is a valid bitmap, the function alters the bitmap and returns it. In this case, the function determines the size of the screen image from the size of the supplied bitmap. Therefore, if you supply a bitmap, the function ignores the second coordinate pair if you give it.

See Chapter 4 for information about screen images and bitmaps.

Format

UIS:READ-IMAGE-PIXEL *window bitmap* &OPTIONAL *x1 y1 x2 y2*

Arguments

window

A window

bitmap

Either an array of unsigned bytes, or NIL. If you supply an array, the function determines the size of the screen image from the size of the array, then modifies the array with the bits of the screen image. If you specify NIL, the function creates an array of size *x2-x1,y2-y1* whose bits correspond to the bits of the screen image.

x1 y1

Two fixnums specifying, in device coordinates, the lower-left corner of the rectangle containing the image. If these arguments are omitted or NIL, they default to 0, specifying the lower-left corner of the viewport. If you supply a bitmap array for the *bitmap* argument, the bitmap array determines the size of the rectangle.

x2 y2

Two fixnums specifying, in device coordinates, the upper-right corner of the rectangle containing the image. These are exclusive bounds; the image extends up to, but does not include, the bounds specified by *x2 y2*. These arguments are ignored if you supply a bitmap array for the *bitmap* argument. If you supply NIL for *bitmap*, *x2* and *y2* default to the upper-right corner of the viewport in such a way that the topmost row and rightmost column of pixels are included in the returned bitmap.

Return Value

A bitmap array

Corresponding MicroVMS Routine

UISDC$READ_IMAGE

READ-KB-CHAR Function

Reads the next keystroke from a virtual keyboard and returns either a character or an integer to represent the keystroke. If the key was a printing or control key, the return value is a character; otherwise, it is an integer. (See Chapter 6 and the description of the KEY-xxx constants for more information about this return value.) If no character is available, the function does not return until one becomes available.

See Chapter 6 for information about using virtual keyboards and interpreting keystrokes.

Format

UIS:READ-KB-CHAR *keyboard*

Arguments

keyboard

A virtual keyboard

Return Value

A character or integer

Corresponding MicroVMS Routine

UIS$READ_CHAR

RESIZE-WINDOW Function

Changes the specified window to have the specified viewport size and location. You can optionally change the size and location of the window within the virtual display as well. The function re-executes the virtual display's display list, if one exists.

You can use this function by itself to alter the size and location of windows and viewports. RESIZE-WINDOW is also useful in an interrupt function supplied as the *action* for the SET-RESIZE-ACTION function. See Section 2.4.6.2 and the description of SET-RESIZE-ACTION.

Format

```
UIS:RESIZE-WINDOW display window x y width height
     &OPTIONAL x1 y1 x2 y2
```

Arguments

display

A virtual display, transformation, or NIL. The virtual display must be the one into which *window* is mapped; NIL is equivalent to specifying this display. A transformation must be one that is mapped into that display. Specifying a transformation allows you to use transformation coordinates instead of world coordinates.

window

A window object

x y

Two single floats specifying the position of the lower-left corner of the window's viewport in centimeters, relative to the lower-left corner of the display screen

width height

Two single floats specifying the width and height of the window's viewport in centimeters

x1 y1 x2 y2

Four single floats specifying the lower-left and upper-right corners of a rectangle in world or transformation coordinates. The resized window is mapped to this rectangle. If these arguments are omitted, the existing window coordinates are used.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$RESIZE_WINDOW

SET-ALIGNED-POSITION Function

Sets the aligned position for text output in a specified virtual display. The aligned position differs from the position established with SET-POSITION in that it refers to the upper-left corner of the next character to be output, rather than the leftmost point on the character's baseline. For this reason, the function requires an input attribute block from which to take the font.

See Section 3.6.2.1 for more information about the aligned text position.

Format

UIS:SET-ALIGNED-POSITION *display att-block x y*

Arguments

display

The virtual display or transformation for which the aligned position is to be set

att-block

A fixnum in the range 0-255, designating an attribute block from which font information will be taken

x *y*

Two single floats designating the world coordinate position of the new aligned text position

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$SET_ALIGNED_POSITION

SET-ALIGNED-POSITION-PIXEL Function

Sets the aligned position for text output in a specified window. The aligned position differs from the position established with SET-POSITION-PIXEL in that it refers to the upper-left corner of the next character to be output, rather than the leftmost point on the character's baseline. For this reason, the function requires an input attribute block from which to take the font.

See Section 3.6.2.1 for more information about the aligned text position.

Format

UIS:SET-ALIGNED-POSITION-PIXEL *window att-block* x *y*

Arguments

window

The window for which the aligned position is to be set

att-block

A fixnum in the range 0-255, designating an attribute block from which font information will be taken *x y*

Two fixnums designating the world coordinate position of the new aligned text position

Return Value

Undefined

Corresponding MicroVMS Routine

UISDC$SET_ALIGNED_POSITION

SET-ATTRIBUTE Function

Modifies an output attribute block by changing the value of one attribute from an input attribute block. The input attribute block is not modified. After the call to SET-ATTRIBUTE, the output attribute block has the same values as the input attribute block, with the exception of the attribute that was changed. The output attribute and input attribute block may be the same, resulting in modification of the input attribute block.

You can supply a display or a window mapped into the display as the first argument to SET-ATTRIBUTE. If you supply a window, you can set the value of any attribute. If you supply a display and you want to set the value of :CLIP-PIXEL or :LEFT-MARGIN-PIXEL, at least one window must be mapped into the display or the function will fail.

See Section 3.3 for more information about attributes.

Format

UIS:SET-ATTRIBUTE *display-or-window input-ab output-ab*
    attribute new-value

Arguments

display-or-window

A virtual display, or a window

input-ab

A fixnum in the range 0-255, representing an attribute block associated with the display

output-ab

A fixnum in the range 1-255, representing an attribute block that will be modified *attribute new-value*

An attribute to be set and its new value:

:ARC-TYPE - :OPEN (the default), :PIE, or :CHORD.

:BACKGROUND-INDEX - An integer specifying an entry in a color map. The integer must be in the range of the entries in the virtual display's color map. The default is 0.

:CHARACTER-SPACING - A list of two single floats, specifying the extra space to leave between characters (horizontally) and lines (vertically). The default is (0.0 0.0). The numbers specify fractions of the character width and font height, respectively.

:CLIP - NIL (the default), to indicate that clipping in the virtual display is turned off in this attribute block; or a list of four single floats in the form (*x1 y1 x2 y2*) to designate the clipping rectangle in world coordinates.

:CLIP-PIXEL - NIL (the default), to indicate that clipping in viewports is turned off in this attribute block; or a list of four fixnums in the form (*x1 y1 x2 y2*) to designate the clipping rectangle in device coordinates.

:FILL-PATTERN - One of the keywords displayed by the SHOW-FILL-PATTERNS function.

:FONT - Either a character string containing the file specification of a font file (or a logical name equated to a font file), a pathname to a font file, or a list containing keyword-value pairs. If you supply a list, each keyword must be one of the font specification keywords displayed by the SHOW-FONTS function, and its value must be a character string. Supply only those keyword-value pairs displayed by SHOW-FONTS for the font you want.

:LEFT-MARGIN - A single-float number specifying the left margin for text operations. The default is the left edge of the virtual display's default window.

:LEFT-MARGIN-PIXEL - A fixnum specifying the left margin for text operations in all windows mapped into a particular virtual display. The default is 0.

:LINE-STYLE - Either one of the line style specification keywords (:DASHED, :DOTTED, :DASHED-DOTTED, or :SOLID), or a bit vector that specifies the line style through the value of its bits. The default is :SOLID. If you supply a bit vector, it must be of length 32 or less; if less, it will be replicated to form a 32-bit vector.

:LINE-WIDTH - Either a single float number specifying the width of a line in multiples of the normal line width, or a list in the form (*n* :WORLD-COORDINATES), where *n* is a single float that specifies the width in world coordinates. The default is 1.0. If you use the list form to specify world coordinates, the width of lines drawn with that attribute block is subject to scaling when displayed in a viewport. Lines whose width is specified as a multiple of the normal width are not scaled.

:WRITING-INDEX - An integer specifying an entry in a color map. The integer must be in the range of the entries in the virtual display's color map. The default is 1.

:WRITING-MODE - :OVERLAY (the default), :TRANSPARENT, :OVERLAY-NEGATE, :COMPLEMENT, :REPLACE, :REPLACE-NEGATE, :ERASE, :ERASE-NEGATE, or :COPY.

Return Value

Undefined

Corresponding MicroVMS Routines

Attribute	**Routine**
:ARC-TYPE	UIS$SET_ARC_TYPE
:BACKGROUND-INDEX	UIS$SET_BACKGROUND_INDEX
:CHARACTER-SPACING	UIS$SET_CHAR_SPACING
:CLIP	UIS$SET_CLIP
:CLIP-PIXEL	UISDC$SET_CLIP
:FILL-PATTERN	UIS$SET_FILL_PATTERN
:FONT	UIS$SET_FONT
:LEFT-MARGIN	UIS$SET_LEFT_MARGIN
:LEFT-MARGIN-PIXEL	UISDC$SET_LEFT_MARGIN
:LINE-STYLE	UIS$SET_LINE_STYLE
:LINE-WIDTH	UIS$SET_LINE_WIDTH
:WRITING-INDEX	UIS$SET_WRITING_INDEX
:WRITING-MODE	UIS$SET_WRITING_MODE

SET-BUTTON-ACTION Function

Specifies the action to be performed when a pointer button is pressed or released. The action can be specified for an entire window or for a specified rectangle in a window. The action can be either an interrupt function identifier (*iif-id*) previously returned by INSTATE-INTERRUPT-FUNCTION, or NIL, in which case no interrupt function will be invoked. If a function is specified, it will receive two arguments: the button code, and an indication of whether the button was pressed (T) or released (NIL). These two arguments precede any that you specify with INSTATE-INTERRUPT-FUNCTION. You can compare the button code with the POINTER-BUTTON-*n* constants.

NOTE

When you suspend a LISP system, all your interrupt functions are uninstated and all *actions* are reset to NIL. You must reinstate interrupt functions and reissue SET-BUTTON-ACTION when the system is resumed.

See Section 5.1.3 for more information about getting input from pointer buttons.

Format

```
UIS:SET-BUTTON-ACTION display window action
     &OPTIONAL x1 y1 x2 y2
```

Arguments

display

A virtual display, transformation, or NIL. The virtual display must be the one into which *window* is mapped; NIL is equivalent to specifying this display. A transformation must be one that is mapped into that display. Specifying a transformation allows you to use transformation coordinates instead of world coordinates.

window

A window

action

Either an *iif-id* to specify an interrupt function, or NIL to specify no action

x1 y1 x2 y2

Four single floats designating the coordinates of two opposite corners of a rectangle in world or transformation coordinates. Button transitions trigger the *action* only if the pointer is within the rectangle at the time. If these arguments are omitted, the entire window is sensitive to button transitions.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$SET_BUTTON_AST

SET-BUTTON-ACTION-PIXEL Function

Specifies the action to be performed when a pointer button is pressed or released. The action can be specified for an entire viewport or for a rectangle in a viewport specified in device coordinates. The action can be either an interrupt function identifier (*iif-id*) previously returned by INSTATE-INTERRUPT-FUNCTION, or NIL, in which case no interrupt function will be invoked. If a function is specified, it will receive two arguments: the button code, and an indication of whether the button was pressed (T) or released (NIL). These two arguments precede any that you specify with INSTATE-INTERRUPT-FUNCTION. You can compare the button code with the POINTER-BUTTON-*n* constants.

NOTE

> When you suspend a LISP system, all your interrupt functions are uninstated and all *actions* are reset to NIL. You must reinstate interrupt functions and reissue SET-BUTTON-ACTION-PIXEL when the system is resumed.

See Section 5.1.3 for more information about getting input from pointer buttons.

Format

UIS:SET-BUTTON-ACTION-PIXEL *window action*
&OPTIONAL *x1 y1 x2 y2*

Arguments

window

A window

action

Either an *iif-id* to specify an interrupt function, or NIL to specify no action

x1 y1 x2 y2

Four fixnums designating the device coordinates of two opposite corners of a rectangle in the viewport associated with *window*. Button transitions trigger the *action* only if the pointer is within the rectangle at the time. If these arguments are omitted, the entire viewport is sensitive to button transitions.

Return Value

Undefined

Corresponding MicroVMS Routine

UISDC$SET_BUTTON_AST

SET-CLOSE-ACTION Function

Specifies the action to be performed when the user closes a window via the Delete selection of the Window Options menu. The action can be one of the following:

- An interrupt function identifier (*iif-id*) previously returned by INSTATE-INTERRUPT-FUNCTION. No arguments are passed to the function unless you specified them with INSTATE-INTERRUPT-FUNCTION.
- :DISALLOW, in which case the user is not allowed to close the window.
- :DELETE, in which case the DELETE-WINDOW function is invoked for the window when the user closes the window.
- :DELETE-DISPLAY, in which case the DELETE-DISPLAY function is invoked for the display into which the window is mapped. Deleting the display automatically deletes the window as well.

The default action taken by VAX LISP is to prohibit the user from closing the window. Specifying :DISALLOW as the *action* argument to SET-CLOSE-ACTION restores this default. See the description of DELETE-WINDOW for information about the :DELETE action.

NOTE

When you suspend a LISP system, all your interrupt functions are uninstated and all *actions* are reset to NIL. You must reinstate interrupt functions and reissue SET-CLOSE-ACTION when the system is resumed.

See Section 2.4.6.3 for more information about using this function.

Format

UIS:SET-CLOSE-ACTION *window action*

Arguments

window

The window associated with the viewport for which a close action is to be specified

action

Either an *iif-id* to specify an interrupt function, or :DISALLOW to prevent the user from closing the window, or :DELETE to delete the window, or :DELETE-DISPLAY to delete the window's display

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$SET_CLOSE_AST

SET-COLOR Function

Sets an entry in the color map associated with a virtual display to a specified color. See Section 3.4 for more information about color.

Format

UIS:SET-COLOR *display color-id r g b*

Arguments

display

A virtual display

color-id

An integer specifying an entry in the color map associated with *display*

r g b

Three floating-point numbers in the range 0.0 - 1.0, inclusive, specifying the intensities for the red, green, and blue components of the color

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$SET_COLOR

SET-GAIN-KB-ACTION Function

Specifies the action to be performed when the physical keyboard is attached to a specified virtual keyboard. The action can be either an interrupt function identifier (*iif-id*) previously returned by INSTATE-INTERRUPT-FUNCTION, or NIL, in which case no interrupt function is invoked. If an interrupt function is specified, no arguments are passed to it upon invocation unless you specified them with INSTATE-INTERRUPT-FUNCTION.

NOTE

When you suspend a LISP system, all your interrupt functions are uninstated and all *actions* are reset to NIL. You must reinstate interrupt functions and reissue SET-GAIN-KB-ACTION when the system is resumed.

See Section 6.1 for more information about using virtual keyboards.

Format

UIS:SET-GAIN-KB-ACTION *keyboard action*

Arguments

keyboard

A virtual keyboard

action

Either an *iif-id* to specify an interrupt function, or NIL to specify no action

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$SET_GAIN_KB_AST

SET-INTENSITY Function

Sets an entry in the color map associated with a virtual display to a specified equivalent monochrome intensity. See Section 3.4 for more information about color and intensity.

Format

UIS:SET-INTENSITY *display color-id i*

Arguments

display

A virtual display

color-id

An integer specifying an entry in the color map associated with *display*

i

A floating-point number in the range 0.0 - 1.0, inclusive

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$SET_INTENSITY

SET-KB-ACTION Function

Specifies the action to be performed when any keyboard key is pressed. The action can be either an interrupt function identifier (*iif-id*) previously returned by INSTATE-INTERRUPT-FUNCTION, or NIL, in which case no interrupt function will be invoked. If an interrupt function is speciifed, it receives two arguments:

1. A character if the key was a control or printing key, and an integer otherwise. (See Chapter 6 and the description of the K-TRM-xxx constants for more information about this return value.)

2. A flag to indicate the key state. This argument is reserved for future use.

Additional arguments are passed to the function if you specified them with INSTATE-INTERRUPT-FUNCTION.

The *action* is invoked only once for each keystroke. This is unlike SET-BUTTON-ACTION, for which the *action* is invoked once when a pointer button is pressed and again when it is released.

NOTE

> When you suspend a LISP system, all your interrupt functions are uninstated and all *actions* are reset to NIL. You must reinstate interrupt functions and reissue SET-KB-ACTION when the system is resumed.

See Section 6.2 for more information about interpreting keystrokes from virtual keyboards.

Format

UIS:SET-KB-ACTION *keyboard action*

Arguments

keyboard

The virtual keyboard for which the function is to be returned

action

Either an *iif-id* to specify an interrupt function, or NIL to specify no action

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$SET_KB_AST

SET-KB-ATTRIBUTES Function

Enables or disables one or more attributes for a specified virtual keyboard. Each virtual keyboard maintains its own set of attributes, which take effect only when the virtual keyboard is connected to the physical keyboard.

Unless the keyword specifying an attribute is specifically mentioned in the function call, the current setting of the attribute remains unaffected. In other words, a default of NIL is *not* assumed for those keywords not included in the argument list.

See Section 6.1.4 for more information about keyboard attributes.

Format

```
UIS:SET-KB-ATTRIBUTES keyboard
     &KEY :AUTOREPEAT :KEYCLICK :FUNCTION-KEYS-6-10
          :FUNCTION-KEYS-11-14 :FUNCTION-KEYS-17-20
          :HELP-DO-KEYS :EDITING-KEYS-1-6 :ARROW-KEYS
          :KEYPAD-KEYS :CLICK-VOLUME
```

Arguments

keyboard

A keyboard

:AUTOREPEAT

T or NIL, causing keyboard autorepeat to be enabled or disabled

:KEYCLICK

T or NIL, causing keyboard keyclick to be enabled or disabled

:FUNCTION-KEYS-6-10

T or NIL, enabling or disabling delivery of keys F6-F10

:FUNCTION-KEYS-11-14

T or NIL, enabling or disabling delivery of keys F11-F14

:FUNCTION-KEYS-17-20

T or NIL, enabling or disabling delivery of keys F17-F20

:HELP-DO-KEYS

T or NIL, enabling or disabling delivery of HELP and DO keys

:EDITING-KEYS-1-6

T or NIL, enabling or disabling delivery of editing keypad keys E1-E6

:ARROW-KEYS

T or NIL, enabling or disabling delivery of arrow keys

:KEYPAD-KEYS

T or NIL, enabling or disabling delivery of numeric keypad keys

:CLICK-VOLUME

An integer between 1 (quiet) and 8 (loud), specifying the keyclick volume

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$SET_KB_ATTRIBUTES

SET-KB-COMPOSE2 Function

This function loads a 2-stroke compose sequence table into a virtual keyboard. Omitting the *table* and *length* arguments returns the keyboard to the system default state.

A keyboard table can be implemented in LISP as an alien structure. The *MicroVMS Workstation Video Device Driver Manual* contains a description of a keyboard table.

Two-stroke compose sequences can be used on all keyboards except the North American keyboard.

Format

UIS:SET-KB-COMPOSE2 *keyboard* &OPTIONAL *table length*

Arguments

keyboard

A virtual keyboard

table length

An appropriate table, as described in the *MicroVMS Workstation Video Device Driver Manual*, and its length in bytes. Omitting these arguments returns the keyboard to the system default 2-stroke compose sequence table.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$SET_KB_COMPOSE2

SET-KB-COMPOSE3 Function

This function loads a 3-stroke compose sequence table into a virtual keyboard. Omitting the *table* and *length* arguments returns the keyboard to the system default state.

A keyboard table can be implemented in LISP as an alien structure. The *MicroVMS Workstation Video Device Driver Manual* contains a description of a keyboard table.

Format

```
    UIS:SET-KB-COMPOSE3 keyboard &OPTIONAL table length
```

Arguments

keyboard

A virtual keyboard

table length

An appropriate table, as described in the *MicroVMS Workstation Video Device Driver Manual*, and its length in bytes. Omitting these arguments returns the keyboard to the system default three-stroke compose key table.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$SET_KB_COMPOSE3

SET-KB-KEYTABLE Function

This function loads a keyboard equivalence table into a virtual keyboard. Omitting the *table* and *length* arguments returns the keyboard to the system default state.

A keyboard table can be implemented in LISP as an alien structure. The *MicroVMS Workstation Video Device Driver Manual* contains a description of a keyboard table.

Format

```
    UIS:SET-KB-KEYTABLE keyboard &OPTIONAL table length
```

Arguments

keyboard

A virtual keyboard

table length

An appropriate table, as described in the *MicroVMS Workstation Video Device Driver Manual*, and its length in bytes. If these arguments are omitted, the keyboard is returned to the default keyboard table.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$SET_KB_KEYTABLE

SET-LOSE-KB-ACTION Function

Specifies the action to be performed when the physical keyboard is disconnected from a specified virtual keyboard. The action can be either an interrupt function identifier (*iif-id*) previously returned by INSTATE-INTERRUPT-FUNCTION, or NIL, in which case no interrupt function is invoked. If an interrupt function is specified, no arguments are passed to it upon invocation unless you specified them with INSTATE-INTERRUPT-FUNCTION.

NOTE

When you suspend a LISP system, all your interrupt functions are uninstated and all *actions* are reset to NIL. You must reinstate interrupt functions and reissue SET-LOSE-KB-ACTION when the system is resumed.

See Section 6.1 for more information about using virtual keyboards.

Format

UIS:SET-LOSE-KB-ACTION *keyboard action*

Arguments

keyboard

A keyboard

action

Either an *iif-id* to specify an interrupt function, or NIL to specify no action

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$SET_LOSE_KB_AST

SET-MOVE-INFO-ACTION Function

Specifies the action to be performed when a specified viewport is moved. The action can be either an interrupt function identifier (*iif-id*) previously returned by INSTATE-INTERRUPT-FUNCTION, or NIL, in which case no interrupt function is invoked. If an interrupt function is specified, no arguments are passed to it upon invocation unless you specified them with INSTATE-INTERRUPT-FUNCTION.

NOTE

When you suspend a LISP system, all your interrupt functions are uninstated and all *actions* are reset to NIL. You must reinstate interrupt functions and reissue SET-MOVE-INFO-ACTION when the system is resumed.

See Section 2.4.3 for information about, and an example of, using this function.

Format

UIS:SET-MOVE-INFO-ACTION *window action*

Arguments

window

The window associated with the viewport for which a move action is to be specified

action

Either an *iif-id* to specify an interrupt function, or NIL to specify no action

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$SET_MOVE_INFO_AST

SET-POINTER-ACTION Function

Specifies the action to be performed when the pointer cursor moves within and/or exits a window or a specified world-coordinate rectangle in a window. The action(s) can be either an interrupt function identifier (*iif-id*) previously returned by INSTATE-INTERRUPT-FUNCTION, or NIL, in which case no interrupt function is invoked. If an interrupt function is specified, no arguments are passed to it upon invocation unless you specified them with INSTATE-INTERRUPT-FUNCTION.

NOTE

When you suspend a LISP system, all your interrupt functions are uninstated and all *actions* are reset to NIL. You must reinstate interrupt functions and reissue SET-POINTER-ACTION when the system is resumed.

See Section 5.1.2 for information about using this function. Chapter 5 contains many examples of its use.

Format

```
UIS:SET-POINTER-ACTION display window
                       move-action exit-action
     &OPTIONAL x1 y1 x2 y2
```

Arguments

display

A virtual display, transformation, or NIL. The virtual display must be the one into which *window* is mapped; NIL is equivalent to specifying this display. A transformation must be one that is mapped into that display. Specifying a transformation allows you to use transformation coordinates instead of world coordinates.

window

A window

move-action

Either an *iif-id* to specify an interrupt function, or NIL to specify no action, when the cursor moves within the rectangle

exit-action

Either an *iif-id* to specify an interrupt function, or NIL to specify no action, when the cursor exits the rectangle

x1 y1 x2 y2

Four single floats designating the world or transformation coordinates of two opposite corners of a rectangle. The *actions* are triggered when the pointer cursor moves within or exits the rectangle. If these arguments are omitted, the *actions* are triggered when the pointer cursor moves within or exits the specified window.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$SET_POINTER_AST

SET-POINTER-ACTION-PIXEL Function

Specifies the action to be performed when the pointer cursor moves within and/or exits a viewport or a specified device-coordinate rectangle in a viewport. The action can be either an interrupt function identifier (*iif-id*) previously returned by INSTATE-INTERRUPT-FUNCTION, or NIL, in which case no interrupt function is invoked. If an interrupt function is specified, no arguments are passed to it upon invocation unless you specified them with INSTATE-INTERRUPT-FUNCTION.

NOTE

When you suspend a LISP system, all your interrupt functions are uninstated and all *actions* are reset to NIL. You must reinstate interrupt functions and reissue SET-POINTER-ACTION-PIXEL when the system is resumed.

See Section 5.1.2 for information about using this function.

Format

```
UIS:SET-POINTER-ACTION-PIXEL window move-action exit-action
     &OPTIONAL x1 y1 x2 y2
```

Arguments

window

A window

move-action

Either an *iif-id* to specify an interrupt function, or NIL to specify no action, when the cursor moves within the rectangle

exit-action

Either an *iif-id* to specify an interrupt function, or NIL to specify no action, when the cursor exits the rectangle

x1 y1 x2 y2

Four fixnums designating the device coordinates of two opposite corners of a rectangle in the viewport associated with *window*. The *actions* are triggered when the pointer cursor moves within or exits the rectangle. If these arguments are NIL or omitted, the *actions* are triggered when the pointer cursor moves within or exits the specified window.

Return Value

Undefined

Corresponding MicroVMS Routine

UISDC$SET_POINTER_AST

SET-POINTER-PATTERN Function

Establishes a new pointer cursor pattern to be used when the pointer cursor is within a given window or specified portion of a window. Once established, the new pattern will be substituted for the current pointer pattern when the pointer cursor enters that area. This function can be called as often as desired to establish different patterns for different rectangles.

See Section 5.1.2 for information about using this function, and an example of its use.

Format

```
UIS:SET-POINTER-PATTERN display window bitmap
                        active-x active-y
    &OPTIONAL x1 y1 x2 y2
```

Arguments

display

A virtual display, transformation, or NIL. The virtual display must be the one into which *window* is mapped; NIL is equivalent to specifying this display. A transformation must be one that is mapped into that display. Specifying a transformation allows you to use transformation coordinates instead of world coordinates.

window

A window

bitmap

The new cursor pattern, in the form of a 16x16 bitmap

active-x active-y

Two fixnums in the range 0-15 designating the active bit in the cursor pattern, relative to the bit at the lower-left corner (0,0). The active bit is used to calculate the pointer position.

x1 y1 x2 y2

Four single floats designating the world or transformation coordinates of two opposite corners of a rectangle. The new cursor pattern will be used when the cursor enters this rectangle. If these arguments are omitted, the entire window is used.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$SET_POINTER_PATTERN

SET-POINTER-PATTERN-PIXEL Function

Establishes a new pointer cursor pattern to be used when the pointer is within a given viewport or specified portion of a viewport. Once established, the new pattern will be substituted for the current pointer pattern when the pointer enters that area. This function can be called as often as desired to establish different patterns for different rectangles.

See Section 5.1.2 for information about using this function.

Format

```
UIS:SET-POINTER-PATTERN-PIXEL window bitmap active-x active-y
     &OPTIONAL x1 y1 x2 y2
```

Arguments

window

A window

bitmap

The new cursor pattern, in the form of a 16x16 bitmap

active-x active-y

Two fixnums in the range 0-15 designating the active bit in the cursor pattern, relative to the bit at the lower-left corner (0,0). The active bit is used to calculate the pointer position.

x1 y1 x2 y2

Four fixnums designating the device coordinates of two opposite corners of a rectangle in the viewport associated with *window*. The new cursor pattern will be used when the cursor enters this rectangle. If these arguments are omitted, the entire viewport is used.

Return Value

Undefined

Corresponding MicroVMS Routine

UISDC$SET_POINTER_PATTERN

SET-POINTER-POSITION Function

Moves the pointer cursor to a new position, specified in world coordinates. The function returns T if the operation was successful, that is, if the new position is within the specified window and is visible. If the new position is not within the specified window or is invisible on the screen, the function returns NIL and does not move the pointer cursor.

Format

UIS:SET-POINTER-POSITION *display window x y*

Arguments

display

A virtual display, transformation, or NIL. The virtual display must be the one into which *window* is mapped; NIL is equivalent to specifying this display. A transformation must be one that is mapped into that display. Specifying a transformation allows you to use transformation coordinates instead of world coordinates.

window

A window

x *y*

Two single floats designating a world or transformation coordinate

Return Value

T if the new position falls within the specified window and is visible; NIL if the new position falls outside the window or if it is invisible

Corresponding MicroVMS Routine

UIS$SET_POINTER_POSITION

SET-POINTER-POSITION-PIXEL Function

Moves the pointer cursor to a new position, specified in device coordinates. The function returns T if the operation was successful, that is, if the new position is within the specified viewport and is visible. If the new position is outside the specified viewport or is invisible, the function returns NIL and does not move the pointer cursor.

Format

UIS:SET-POINTER-POSITION-PIXEL *window* x *y*

Arguments

window

A window

x y

Two fixnums designating a device coordinate in the window

Return Value

T if the new position falls within the specified viewport and is visible; NIL if the new position falls outside the viewport or if it is invisible

Corresponding MicroVMS Routine

UISDC$SET_POINTER_POSITION

SET-POSITION Function

Sets the current text position for a virtual display. The text position is the alignment point on the baseline of the next character to be output.

See Section 3.6.2 for information about text positioning.

Format

UIS:SET-POSITION *display x y*

Arguments

display

The virtual display or transformation for which the text position is to be set.

x y

Two single floats designating the world coordinates of the text position.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$SET_POSITION

SET-POSITION-PIXEL Function

Sets the device-coordinate text position for windows mapped into a virtual display. The text position is the leftmost point on the baseline of the next character to be output. The device-coordinate text position is independent of the display text position; however, all windows mapped into a virtual display share the same device-coordinate text position.

See Section 3.6.2 for information about text positioning.

Format

UIS:SET-POSITION-PIXEL *window x y*

Arguments

window

The window for which the text position is to be set.

x y

Two fixnums designating the device coordinates of the text position.

Return Value

Undefined

Corresponding MicroVMS Routine

UISDC$SET_POSITION

SET-RESIZE-ACTION Function

Specifies the action to be performed when the user resizes a window via the "Change the size" selection of the "Window Options" menu. The action can be one of the following:

- An interrupt function identifier (*iif-id*) previously returned by INSTATE-INTERRUPT-FUNCTION. The interrupt function receives the following eight arguments:
  - Two floating point numbers indicating the new position, in screen coordinates (centimeters), of the lower-left corner of the viewport.
  - Two floating point numbers indicating the new width and height, in centimeters, of the viewport.
  - Four floating point numbers indicating the new coordinates of the corners of the window, in the form *x1 y1 x2 y2*. The coordinates are world coordinates or transformation coordinates, depending on the first argument to

SET-RESIZE-ACTION.

- :DISALLOW, in which case the user is not allowed to resize the window.
- :DEFAULT or NIL, in which case a default action is invoked when the user resizes the window. The default action is to change the size of both the viewport and the window by the amount and in the direction the user requests. For example, if the user stretches the viewport to the right, the window into the virtual display will also stretch to the right, exposing more of what is in the virtual display.

If you specify an interrupt function as the *action*, the interrupt function receives the eight arguments noted above before any arguments that you specified with INSTATE-INTERRUPT-FUNCTION. Note that if you specify an interrupt function, the graphics system does not resize the window and viewport automatically. The interrupt function can resize the window and viewport by calling the RESIZE-WINDOW function and passing it the same eight arguments it received when it was invoked.

NOTE

When you suspend a LISP system, all your interrupt functions are uninstated and all *actions* are reset to NIL. You must reinstate interrupt functions and reissue SET-RESIZE-ACTION when the system is resumed.

See Section 2.4.6.2 for information about using this function. Section 7.2.1 contains an example of its use.

Format

UIS:SET-RESIZE-ACTION *display window action*

Arguments

display

A virtual display, transformation, or NIL. The virtual display must be the one into which *window* is mapped; NIL is equivalent to specifying this display. A transformation must be one that is mapped into that display.

window

The window for which the function is to be returned

action

Either an *iif-id* to specify an interrupt function, or :DISALLOW to prevent the user from resizing the window, or NIL to specify no action. See the description of the function above for information about arguments received by the interrupt function.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$SET_RESIZE_AST

SHOW-FILL-PATTERNS Function

Displays all the available fill patterns, along with the keyword that designates each one. Use this function to select a fill pattern, then use the corresponding keyword as the value of the :FILL-PATTERN keyword in the SET-ATTRIBUTE function.

See Section 3.5.3.2 for information about specifying fill patterns.

Format

```
SHOW-FILL-PATTERNS
```

Arguments

None

Return Value

Undefined

Corresponding MicroVMS Routine

None

SHOW-FONTS Function

Displays a table of all the available fonts. The table consists of a row for each font, and a column for each of the font specification keywords that you can use with the :FONT keyword of the SET-ATTRIBUTE function. Each table entry is a character string that is the value of the font specification keyword for that particular font. No entry in a column indicates that the font has the default value for that font specification keyword.

See Section 3.6.3.2 for information about specifying fonts.

Format

```
SHOW-FONTS
```

Arguments

None

Return Value

Undefined

Corresponding MicroVMS Routine

None

SOUND-BELL Function

Sounds the keyboard "bell."

Format

UIS:SOUND-BELL &OPTIONAL *volume device*

Arguments

volume

A fixnum in the range 1-8 specifying the volume of the sound. If this argument is omitted, the volume is taken from the default workstation bell volume.

device

A character string specifying the output device. The default is SYS$WORKSTATION.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$SOUND_BELL

SOUND-CLICK Function

Sounds the keyboard keyclick.

Format

UIS:SOUND-CLICK &OPTIONAL *volume device*

Arguments

volume

A fixnum in the range 1-8 specifying the volume of the sound. If this argument is omitted, the volume is taken from the default workstation keyclick volume.

device

A character string specifying the output device. The default is SYS$WORKSTATION.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$SOUND_CLICK

TEST-KB Function

Returns T if the specified virtual keyboard is connected to the physical keyboard, and NIL otherwise.

Format

UIS:TEST-KB *keyboard*

Arguments

keyboard

A keyboard

Return Value

T or NIL

Corresponding MicroVMS Routine

UIS$TEST_KB

TEXT Function

Draws a character string in a virtual display, and moves the display's text position to the end of the string. See Section 3.6 for information on text output operations.

Format

```
UIS:TEXT display att-block text-string &OPTIONAL x y
```

Arguments

display

The virtual display or transformation in which the text is to be drawn

att-block

A fixnum in the range 0-255, designating the attribute block from which font and other text attributes are to be taken

text-string

A character string or a single character

x *y*

Two single floats specifying the world coordinates of the upper-left corner of the text string. If these arguments are omitted, the string begins at the current text position.

Return Value

Undefined

Corresponding MicroVMS Routine

UIS$TEXT

TEXT-PIXEL Function

Draws a character string in a viewport, and moves the device-coordinate text position to the end of the string. See Section 3.6 for information on text output operations.

Format

```
UIS:TEXT-PIXEL window att-block text-string &OPTIONAL x y
```

Arguments

window

The window in which the text is to be drawn

att-block

A fixnum in the range 0-255, designating the attribute block from which font and other text attributes are to be taken

text-string

A character string or a single character

x y

Two fixnums specifying the device coordinates of the upper-left corner of the text string. If these arguments are omitted, the string begins at the current device-coordinate text position.

Return Value

Undefined

Corresponding MicroVMS Routine

UISDC$TEXT

TRANSFORMATION Type Specifier

Designates objects of type TRANSFORMATION, created by the CREATE-TRANSFORMATION function.

TRANSFORMATIONP Function

Returns T if its argument is a transformation object and NIL otherwise.

Format

UIS:TRANSFORMATIONP *object*

Arguments

object

Any LISP object

Return Value

T or NIL

Corresponding MicroVMS Routine

None

UIS-ID Function

Returns the UIS ID of its argument for use with CALL-OUT. The argument may be a display, transformation, window, or keyboard.

Format

UIS:UIS-ID *object*

Arguments

object

An object of type DISPLAY, TRANSFORMATION, WINDOW, or KEYBOARD

Return Value

An integer that is the UIS ID of the object

Corresponding MicroVMS Routine

None

WINDOW-DISPLAY Function

Returns the display with which a specified window is associated.

Format

UIS:WINDOW-DISPLAY *window*

Arguments

window

A window object

Return Value

A display

Corresponding MicroVMS Routine

None

WINDOW Type Specifier

Designates objects of type WINDOW, created by the CREATE-WINDOW function.

WINDOWP Function

Returns T if its argument is a window and NIL otherwise.

Format

UIS:WINDOWP *object*

Arguments

object

Any LISP object

Return Value

T or NIL

Corresponding MicroVMS Routine

None

WINDOW-STREAM-ATTRIBUTE-BLOCK Function

Returns the attribute block used to write output in a particular window output stream. You can use this function with SETF to change the attribute block.

See Chapter 7 for information about window output streams.

Format

UIS:WINDOW-STREAM-ATTRIBUTE-BLOCK *window-output-stream*

Arguments

window-output-stream

A window output stream previously created with MAKE-WINDOW-OUTPUT-STREAM or WITH-OUTPUT-TO-WINDOW

Return Value

An integer that designates the attribute block used to write output to *window-output-stream*

Corresponding MicroVMS Routine

None

WINDOW-STREAM-HORIZONTAL-OVERFLOW Function

Returns a keyword indicating the behavior of a particular window output stream when an attempt is made to write text to the right of the viewing area. The keyword can be either :TRUNCATE or :WRAP. You can use this function with SETF to change the horizontal overflow behavior.

See Chapter 7 for information about window output streams.

Format

UIS:WINDOW-STREAM-HORIZONTAL-OVERFLOW *window-output-stream*

Arguments

window-output-stream

A window output stream previously created with MAKE-WINDOW-OUTPUT-STREAM or WITH-OUTPUT-TO-WINDOW

Return Value

A keyword (either :TRUNCATE or :WRAP) that indicates the horizontal overflow behavior of *window-output-stream*

Corresponding MicroVMS Routine

None

WINDOW-STREAM-VERTICAL-OVERFLOW Function

Returns a keyword indicating the behavior of a particular window output stream when an attempt is made to output text beyond the bottom of the viewing area. The keyword can be either :SCROLL, :WRAP, or :TRUNCATE. You can use this function with SETF to change the vertical overflow behavior.

See Chapter 7 for information about window output streams.

Format

UIS:WINDOW-STREAM-VERTICAL-OVERFLOW *window-output-stream*

Arguments

window-output-stream

A window output stream previously created with MAKE-WINDOW-OUTPUT-STREAM or WITH-OUTPUT-TO-WINDOW

Return Value

A keyword (either :SCROLL, :WRAP, or :TRUNCATE) that indicates the vertical overflow behavior of *window-output-stream*

Corresponding MicroVMS Routine

None

WINDOW-STREAM-VIEWING-AREA Function

Returns a list of integers in the form (*x1 y1 x2 y2*) to indicate the viewing area for a specified window output stream in device coordinates. You can use this function with SETF to change the viewing area. When you change the viewing area, the current text position (the position where text from the stream will next be placed) is moved to the top left corner of the new viewing area, and the new viewing area is erased. If you specify a value of NIL when using SETF with WINDOW-STREAM-VIEWING-AREA, the viewing area is changed to occupy the entire viewport.

See Chapter 7 for information about window output streams. Section 7.2.1 contains an example of this function in use.

Format

UIS:WINDOW-STREAM-VIEWING-AREA *window-output-stream*

Arguments

window-output-stream

A window output stream previously created with MAKE-WINDOW-OUTPUT-STREAM or WITH-OUTPUT-TO-WINDOW

Return Value

A list of four integers designating the viewing area rectangle for *window-output-stream* in device coordinates

Corresponding MicroVMS Routine

None

WINDOW-STREAM-WINDOW Function

Returns the window object to which the window output stream given in its argument sends output. You cannot use SETF with this function to change a stream's output window.

See Chapter 7 for information about window output streams.

Format

UIS:WINDOW-STREAM-WINDOW *window-output-stream*

Arguments

window-output-stream

A window output stream previously created with MAKE-WINDOW-OUTPUT-STREAM or WITH-OUTPUT-TO-WINDOW

Return Value

A window object

Corresponding MicroVMS Routine

None

WINDOW-STREAM-X-POSITION Function

Returns an integer indicating, in device coordinates, the horizontal position at which text will be output from a specified window output stream. You can use SETF with this function to change the horizontal output position.

See Chapter 7 for information about window output streams.

Format

UIS:WINDOW-STREAM-X-POSITION *window-output-stream*

Arguments

window-output-stream

A window output stream previously created with MAKE-WINDOW-OUTPUT-STREAM or WITH-OUTPUT-TO-WINDOW

Return Value

An integer indicating the horizontal text position for this window output stream in device-coordinate units

Corresponding MicroVMS Routine

None

WINDOW-STREAM-Y-POSITION Function

Returns an integer indicating, in device coordinates, the vertical position at which text will be output from a specified window output stream. You can use SETF with this function to change the vertical output position.

See Chapter 7 for information about window output streams.

Format

UIS:WINDOW-STREAM-Y-POSITION *window-output-stream*

Arguments

window-output-stream

A window output stream previously created with MAKE-WINDOW-OUTPUT-STREAM or WITH-OUTPUT-TO-WINDOW

Return Value

An integer indicating the vertical text position for this window output stream in device-coordinate units

Corresponding MicroVMS Routine

None

WITH-OUTPUT-TO-WINDOW Macro

Creates a window output stream and binds it to a variable. The macro then executes its forms as an implicit PROGN and returns the value of the last form. Finally, the stream is closed and unbound from the variable to which it was bound.

See Chapter 7 for information about window output streams.

Format

```
UIS:WITH-OUTPUT-TO-WINDOW (var window &REST options)
     {declaration}* {form}*
```

Arguments

var window

A variable *var* which is bound to a window output stream into *window*. The stream and binding exist only while the forms in the body of the function are executing.

options

Keyword-value pairs; the allowable keywords are those you can specify with the MAKE-WINDOW-OUTPUT-STREAM function

Return Value

The value of the last form evaluated

Corresponding MicroVMS Routine

None

INDEX

Page numbers in the Index in the form c-n (for example, 2-13) refer to a page in Part I. Page numbers in the form n (for example, 25) refer to a page in Part II.

HOW TO ORDER ADDITIONAL DOCUMENTATION

From	Call	Write
Chicago	312–640–5612 8:15 A.M. to 5:00 P.M. CT	Digital Equipment Corporation Accessories & Supplies Center 1050 East Remington Road Schaumburg, IL 60195
San Francisco	408–734–4915 8:15 A.M. to 5:00 P.M. PT	Digital Equipment Corporation Accessories & Supplies Center 632 Caribbean Drive Sunnyvale, CA 94086
Alaska, Hawaii	603–884–6660 8:30 A.M. to 6:00 P.M. ET *or* 408–734–4915 8:15 A.M. to 5:00 P.M. PT	
New Hampshire	603–884–6660 8:30 A.M. to 6:00 P.M. ET	Digital Equipment Corporation Accessories & Supplies Center P.O. Box CS2008 Nashua, NH 03061
Rest of U.S.A., Puerto Rico*	1–800–258–1710 8:30 A.M. to 6:00 P.M. ET	

*Prepaid orders from Puerto Rico must be placed with the local DIGITAL subsidiary (call 809–754–7575)

From	Call	Write
Canada British Columbia	1–800–267–6146 8:00 A.M. to 5:00 P.M. ET	Digital Equipment of Canada Ltd 940 Belfast Road Ottawa, Ontario K1G 4C2 Attn: A&SG Business Manager
Ottawa–Hull	613–234–7726 8:00 A.M. to 5:00 P.M. ET	
Elsewhere	112–800–267–6146 8:00 A.M. to 5:00 P.M. ET	
Elsewhere		Digital Equipment Corporation A&SG Business Manager*

*c/o DIGITAL's local subsidiary or approved distributor

VAX LISP/VMS Graphics
Programming Guide
AA-GH76A-TE

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent:

Assembly language programmer
Higher-level language programmer
Occasional programmer (experienced)
User with little programming experience
Student programmer
Other (please specify) ______

Name ______ Date ______

Organization ______

Street ______

City ______ State ______ Zip Code ______
or Country

— — Do Not Tear — Fold Here and Tape — — — — — — — — — — — — — — — — — —

digital

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
CORPORATE USER PUBLICATIONS
MLO5–5/E45
146 MAIN STREET
MAYNARD, MA 01754–2571

— — Do Not Tear — Fold Here — — — — — — — — — — — — — — — — — —

Cut Along Dotted Line