VMS Workstation Software Graphics Programming Guide

Order Number: AA-GI10C-TE

May 1988

This document provides programming information about the VMS Workstation Software graphics. It describes the general concepts and specific routine calls used to write application programs.

Revision/Update Information:	This manual supersedes the *MicroVMS Workstation Graphics Programming Guide*, Version 3.0.
Software Version:	This manual contains information for both VWS Version 3.3 and VWS Version 4.0
Operating System:	For systems running VMS Version 4.6 and later, VWS Version 3.3. For systems running VMS Version 5.0, VWS Version 4.0.

digital equipment corporation maynard, massachusetts

May 1988

DEC	DIBOL	UNIBUS
DEC/CMS	EduSystem	VAX
DEC/MMS	IAS	VAXcluster
DECnet	MASSBUS	VMS
DECsystem-10	PDP	VT
DECSYSTEM-20	PDT	digital™
DECUS	RSTS	
DECwriter	RSX	

This document was prepared using VAX DOCUMENT, Version 1.0

Contents

PART III UIS ROUTINE DESCRIPTIONS

Preface

This programming guide describes the VMS workstation graphics software. It contains general information about basic VMS graphics concepts, a tutorial for learning to program with VMS graphics, and complete descriptions and reference information about the system routines for all callable functions.

Intended Audience

This guide is intended for general users and programmers who want to learn the concepts and use appropriate routines in graphics application programs.

Document Structure

This guide is divided into four major sections, VMS Workstation Software Graphics Concepts, How to Program with VMS Workstation Software Graphics, UIS Routine Descriptions, and UIS Device Coordinate (UISDC) Routines. These sections are briefly described in the following paragraphs.

Part I — VMS Workstation Software Graphics Concepts

This section contains five chapters with a general overview of the basic concepts of VMS workstation graphics.

- Chapter 1 — System Description

  This chapter briefly describes the hardware, software, and options that are parts of the VMS workstation system.

- Chapter 2 — Display Management Concepts

  This chapter discusses the concepts of world coordinates, device coordinates, virtual displays, windows, viewports, window and viewport scaling, and distortion of graphic objects.

- Chapter 3 — Graphic Objects and Attributes

  This chapter describes and shows the relationship between graphics routines, attribute blocks, text attributes, graphics attributes, and segments.

- Chapter 4 — Color Concepts

  This chapter discusses the various color and intensity environments supported by the VAXstation color systems.

- Chapter 5 — Input Devices

  This chapter shows how the workstation input devices relate to the workstation graphics system.

Part II — How to Program with VMS Workstation Software Graphics

This section contains step-by-step tutorial information about writing application programs using VMS workstation software graphics. Practical programming examples are provided throughout this section. It is divided according to routine functions into the following chapters:

- Chapter 6 — Programming Considerations

  This chapter describes the programming interface and topics relating to program execution.

- Chapter 7 — Creating Basic Graphic Objects

  This chapter describes the underlying structures and shows how to create graphic objects.

- Chapter 8 — Display Windows and Viewports

  This chapter shows how to create and manipulate display windows and display viewports.

- Chapter 9 — General Attributes

  This chapter describes writing modes, display background and foreground, and the writing index.

- Chapter 10 — Text Attributes

  This chapter describes how attributes may be used to enhance and modify text.

- Chapter 11 — Graphics Attributes

  This chapter describes how attributes may be used to enhance and modify the appearance of graphic objects.

- Chapter 12 — Inquiry Routines

  This chapter discusses how information can be returned to the application program.

- Chapter 13 — Display Lists and Segmentation

  This chapter describes how to create and manipulate display lists and segments.

- Chapter 14 — Geometric and Attribute Transformations

  This chapter describes the various ways graphic objects and components of graphic objects can be manipulated with the respect to the coordinate space.

- Chapter 15 — Metafiles and Private Data

  This chapter discusses how to extract the contents of a display list and store the data in a buffer or external file. There is additional information about how to associate private data with a graphics display.

- Chapter 16 — Programming in Color

  The chapter describes how to create and display graphic objects in color.

- Chapter 17 — Asynchronous System Trap Routines

This chapter discusses how to make use of program-related events to increase the interactive nature of your applications.

Part III — UIS Routine Descriptions

This section contains reference material about the device-independent VMS workstation software graphics routines.

- Chapter 18 — UIS Routine Descriptions
- UIS Routine Descriptions

Part IV — UIS Device Coordinate (UISDC) Routines

This section contains reference material about device-dependent VMS workstation software graphics routines.

- Chapter 19 — UIS Device Coordinate Graphics Routines
- UISDC Routines

Appendix A — Summary of UIS Calling Sequences

Appendix B — Summary of UISDC Calling Sequences

Appendix C — UIS Fonts

Appendix D — UIS Fill Patterns

Appendix E — Error Messages

Appendix F — VMS Data Types

Glossary

How To Use This Guide

This guide is designed so that different types of users can benefit by its information:

- General users and programmers new to graphics software and VMS workstation software graphics can use it as a learning tool.
- Programmers already familiar with graphics software in general and/or VMS workstation software graphics can use it as a reference tool.

Inexperienced Users

If you are unfamiliar with the VMS workstation software graphics system, you should begin by reading Part I of this guide. It gives you an overview of the graphics concepts discussed in subsequent sections of the book.

The programming tutorial in Part II provides a step-by-step approach for learning how to write applications that take advantage of the graphics capabilities of the VMS workstation.

Part III provides you with reference information about all of the UIS routines used in VMS workstation software graphics. It is easier to use after you have read Part II of this guide.

Part IV contains appendices that provide reference material about UISDC graphics routines and error messages.

Experienced Users

Once you have become familiar with VMS workstation graphics, you will seldom need to refer to Part I of this guide, except when reviewing basic concepts.

Refer to Part II for examples and suggestions on the proper use of VMS workstation software graphics routines.

Part III is an alphabetically arranged reference section that you can use to get detailed descriptions of VMS workstation software graphics routines. Before using this section, you should already be familiar with Parts I and II of this guide.

Part IV contains appendices that provide reference material about UISDC graphics routines and error messages.

Associated Documents

The following VMS manuals are related to this guide:

- *VMS Workstation Software Release Notes*
- *VMS Workstation Software Installation Guide*
- *VMS Workstation Software Guide to Printing Graphics*
- *VMS Workstation Software User's Guide*
- *VMS Workstation Software Video Device Driver Manual*
- *VMS User's Manual*
- *VMS User's Primer*
- *VMS Programmer's Manual*
- *VMS FORTRAN Programmer's Primer*
- *VMS Programming Pocket Reference*
- *VMS Programming Support Manual*
- *Installing or Upgrading VMS from Diskettes*
- *Installing or Upgrading VMS from a Tape Cartridge*

Documentation Conventions

This manual uses the following conventions:

Convention	Meaning
$\boxed{\text{RET}}$	A symbol with a one- to six-character abbreviation indicates that you press a key on the terminal, for example, $\boxed{\text{RET}}$.
$\boxed{\text{CTRL/x}}$	The phrase CTRL/x indicates that you must press the key labeled CTRL while you simultaneously press another key, for example, CTRL/C, CTRL/Y, CTRL/O.
$ SHOW TIME 05-JUN-1986 11:55:22	Command examples show all output lines orprompting characters that the system prints or displays in black letters. All user-entered commands are shown in red letters.
$ TYPE MYFILE.DAT . . .	Vertical series of periods, or ellipsis, mean either that not all the data that the system would display in response to the particular command is shown or that not all the data a user would enter is shown.
file-spec,...	Horizontal ellipsis indicates that additional parameters, values, or information can be entered.
[logical-name]	Square brackets indicate that the enclosed item is optional. (Square brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
quotation marks apostrophes	The term quotation marks is used torefer to double quotation marks ("). The term apostrophe (') is used to refer to a single quotation mark.

New and Changed Features

(New and Changed Features) The following sections describe changes to the programming interface since UIS Version 2.0.

New UIS Routines

The following UIS routines were added.

Function	Routine
AST-enabling	UIS$SET_ADDOPT_AST
	UIS$SET_EXPAND_ICON_AST
	UIS$SET_TB_AST
	UIS$SET_SHRINK_TO_ICON_AST
Color	UIS$CREATE_COLOR_MAP
	UIS$CREATE_COLOR_MAP_SEG
	UIS$DELETE_COLOR_MAP
	UIS$DELETE_COLOR_MAP_SEG
	UIS$GET_COLORS
	UIS$GET_HW_COLOR_INFO
	UIS$GET_INTENSITIES
	UIS$GET_VCM_ID
	UIS$HLS_TO_RGB
	UIS$HSV_TO_RGB
	UIS$RESTORE_CMS_COLORS
	UIS$RGB_TO_HLS
	UIS$RGB_TO_HSV
	UIS$SET_INTENSITIES
Display list	UIS$COPY_OBJECT
	UIS$DELETE_OBJECT
	UIS$DELETE_PRIVATE
	UIS$EXECUTE
	UIS$EXECUTE_DISPLAY
	UIS$EXTRACT_HEADER
	UIS$EXTRACT_OBJECT
	UIS$EXTRACT_PRIVATE
	UIS$EXTRACT_REGION
	UIS$EXTRACT_TRAILER
	UIS$FIND_PRIMITIVE
	UIS$FIND_SEGMENT
	UIS$GET_CURRENT_OBJECT
	UIS$GET_NEXT_OBJECT
	UIS$GET_OBJECT_ATTRIBUTES
	UIS$GET_PARENT_SEGMENT
	UIS$GET_PREVIOUS_OBJECT
	UIS$GET_ROOT_SEGMENT
	UIS$INSERT_OBJECT
	UIS$PRIVATE
	UIS$SET_INSERTION_POSITION
	UIS$TRANSFORM_OBJECT
Graphics	UIS$LINE
	UIS$LINE_ARRAY

Function	Routine
Keyboard and pointer	UIS$CREATE_TB
	UIS$DELETE_TB
	UIS$DISABLE_TB
	UIS$ENABLE_TB
	UIS$GET_TB_INFO
	UIS$GET_TB_POSITION
Text	UIS$GET_CHAR_ROTATION
	UIS$GET_CHAR_SIZE
	UIS$GET_CHAR_SLANT
	UIS$GET_FONT_ATTRIBUTES
	UIS$GET_TEXT_FORMATTING
	UIS$GET_TEXT_MARGINS
	UIS$GET_TEXT_PATH
	UIS$GET_TEXT_SLOPE
	UIS$SET_CHAR_ROTATION
	UIS$SET_CHAR_SIZE
	UIS$SET_CHAR_SLANT
	UIS$SET_TEXT_FORMATTING
	UIS$SET_TEXT_MARGINS
	UIS$SET_TEXT_PATH
	UIS$SET_TEXT_SLOPE
Windowing	UIS$EXPAND_ICON
	UIS$GET_VIEWPORT_ICON
	UIS$GET_WINDOW_SIZE
	UIS$SHRINK_TO_ICON

New UISDC Routines

The following UISDC routines were new for Version 3.0.

- UISDC$ALLOCATE_DOP
- UISDC$EXECUTE_DOP_ASYNCH
- UISDC$EXECUTE_DOP_SYNCH
- UISDC$GET_CHAR_SIZE
- UISDC$GET_TEXT_MARGINS
- UISDC$LINE
- UISDC$LINE_ARRAY
- UISDC$LOAD_BITMAP
- UISDC$QUEUE_DOP
- UISDC$SET_CHAR_SIZE
- UISDC$SET_TEXT_MARGINS

New Chapters

Three new chapters describing color concepts and color programming considerations have been added since Version 2.0.

- Color Concepts
- Geometric and Attribute Transformations
- Programming in Color

New UIS Writing Modes

Five new writing modes have been added since Version 2.0.

- UIS$C_MODE_BIC
- UIS$C_MODE_BICN
- UIS$C_MODE_BIS
- UIS$C_MODE_BISN
- UIS$C_MODE_COPYN

New Technical Character Set Fonts

Twelve new technical character set fonts have been added since Version 2.0.

New Text Attributes

The following new text attributes have been added to the programming interface.

- Character rotation
- Character scaling
- Character slant
- Text formatting
- Text margins
- Text path
- Text slope

Changes to Existing UIS Routines

UIS$BEGIN_SEGMENT

UIS$BEGIN_SEGMENT now returns segment identifier that can be referenced by other display list routines. For example, this allows traversing segments and segment paths.

UIS$MEASURE_TEXT and UIS$TEXT

You can now use control lists with UIS$TEXT and UIS$MEASURE_TEXT.

UIS$DISABLE_DISPLAY_LIST and UIS$ENABLE_DISPLAY_LIST

Additional arguments have been included that control display screen and display list updates.

UIS$SET_POINTER_PATTERN and UISDC$SET_POINTER_PATTERN

If you are using a color system, you can now specify a pointer pattern outline.

Display Lists and Segmentation

The chapter on display lists and segmentation has been expanded with more examples.

UIS Metafiles

You can create and store metafiles of generically encoded instructions as files and reexecute the file.

Shrinking Viewports and Expanding Icons

Applications can now shrink display viewports and expand icons.

Obsolete Version 2.0 UIS Routines

The following routines are obsolete.

- UIS$GET_LEFT_MARGIN
- UIS$SET_LEFT_MARGIN
- UISDC$GET_LEFT_MARGIN
- UISDC$SET_LEFT_MARGIN

Part I VMS Workstation Software Graphics Concepts

1 System Description

1.1 Overview

This chapter introduces the VMS Workstation Software graphics system. The chapter has two parts:

- A summary of typical workstation hardware
- A description of the graphics software

1.2 VAXstation Hardware

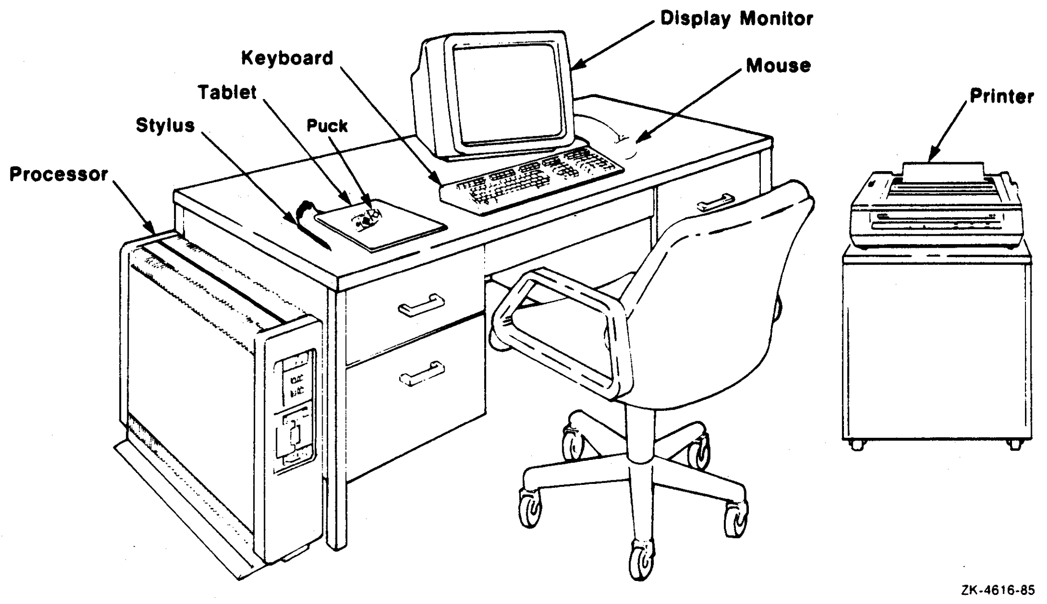
The VMS workstation can be used as a *standalone* system. It has all the components necessary to run programs and perform tasks without being connected to a host computer. It can also be connected to a host computer and used as a part of a network in a larger system.

The VMS workstation typically consists of a configuration of the following hardware:

- System cabinets or boxes
- Display monitor
- Keyboard
- Tablet with puck and stylus or three-button mouse
- Communications board
- Printer

Figure 1–1 shows typical VMS workstation hardware.

Figure 1-1 Typical VMS Workstation Hardware



1.2.1 System Cabinet or Box

The system cabinet (box) is the heart of the VMS workstation system. The system cabinet contains the CPU, disk drives, memory, any options, and communications hardware for the system. Usually, the cabinet or box houses both fixed and flexible disk drives. The amount of memory can vary, depending on the options installed.

1.2.2 Monitor

The workstation monitor is a high-resolution, bitmap device that displays text and graphics information. Depending on the model, you can use the workstation monitor to display black and white (bitonal), grey scale, or color graphics.

1.2.3 Keyboard

The workstation uses the DIGITAL LK201, a standard low-profile style keyboard that consists of:

- A top row of user-definable function keys
- A user-definable numeric keypad
- A special keypad with arrow keys and function keys
- A standard alphanumeric keypad

Some of the top-row function keys are control keys that allow you to:

- Hold the screen
- Display the operator window
- Switch the windowing system
- Change the active window

The top row also has editing keys and keys that call functions such as cancel, exit, and help.

You can program the function keys and numeric keypad keys to perform functions suited to a particular application. You can use the arrow keys to move the keyboard cursor within applications. The alphanumeric keypad is similar in function to a typewriter keyboard.

1.2.4 Mouse

The three-button mouse is a medium-resolution, relative pointing device. The mouse is the primary means for pointing to an object on the screen. When you roll the mouse on a flat surface, the pointer on the screen moves the same way. You use the buttons to make selections.

1.2.5 Tablet

The tablet is a high-resolution, absolute positioning device. It consists of a flat tablet, a puck with buttons, and a stylus with buttons. When you move the puck or stylus on the tablet, the pointer on the display screen moves the same way. You use the buttons to make selections.

1.2.6 Communications Board

The communications board connects the system to other computers.

1.2.7 Printer

The VMS workstation can have a printer connected to the processor console port or can access printers located at remote locations through the network. You can print any rectangular portion of display screen.

1.3 Software

The VMS workstation graphics software is a versatile graphics and windowing interface. It is designed to be used on any of the MicroVAX family of workstation products (such as VAXstations). This graphics interface allows you to write application programs in VAX MACRO, VAX BLISS, and many other high-level languages. Application programs written to use graphics software can create and manipulate windows, display multiple styles of text and sizes, receive input, and draw graphic objects in the windows.

1.3.1 Graphics Routine Types

The VMS workstation graphics software contains callable routines that can be accessed from a high-level programming language. An application program can perform graphics and windowing functions by making calls to appropriate routines. Routines create display windows, draw lines and text, and build graphic objects. This software contains the following types of routines:

- AST-enabling
- Attribute
- Color
- Display list
- Graphics and text
- Inquiry
- Keyboard
- Pointer
- Sound
- Windowing
- Device coordinate

1.3.2 Human Interface

The VMS workstation provides an interface between you and the graphics software. This feature is called the *human interface* because it helps you use the workstation.

This interface makes it easy to create new terminal windows on the screen. The VMS workstation provides you with the ability to have the equivalent of many terminals at your disposal. You can easily create emulated Digital VT200 series or Tektronix TEK4014 terminals simply by selecting a menu item that creates a window on the screen.

To control the placement of windows on the screen, you can move them anywhere on the screen (or even partially off it), hide them from view, push them behind or pop them in front of other windows, and so on. The following list shows some possible operations.

- Create a new VT200 series or TEK4014 terminal window
- Move a window to a different part of the screen
- Push a window behind other windows
- Pop a window in front of other windows
- Shrink a viewport to a icon
- Change the size of a window
- Delete a window
- Switch the keyboard from one window to another

- Suspend all screen activity (hold screen)
- Print any portion (or all) of a window or the screen
- Set workstation attributes
- Get online help

1.3.2.1 Terminal Emulation

You can create emulated terminals on the VMS workstation. The programming interface and the capabilities of emulated terminals are the same as the programming interface and capabilities of the corresponding real terminal. The appearance of an emulated terminal on the VMS workstation screen is similar to that of the corresponding real terminal. (It will not be completely identical because of hardware differences.)

If you have several terminal windows, you can start a job on one terminal window, leave it running, then start a job on another terminal window. Based on available resources, you create as many terminal windows as you need and switch back and forth among them at will.

VT200 Series/TEK4014

The VAXstation can emulate the Digital VT200 series or Tektronix TEK4014 terminal. Any number of VT200 series or TEK4014 windows can appear on the screen simultaneously. However, only one window can use the keyboard at any one time. You assign the keyboard to the window of your choice.

VT200 ANSI and Digital private escape sequences, and TEK4014 escape sequences, are interpreted and translated into the appropriate graphics routines.

Programs that run under the VAX/VMS operating system will operate in a VT100 or VT200 series workstation window without modification.

1.3.2.2 Communication Tools

You can communicate with the software interface through the mouse, tablet, or keyboard.

Mouse and Tablet

The mouse and tablet control a cursor called a *pointer* on the screen. When you manipulate the mouse or tablet, the pointer moves on the screen. Use the pointer to choose objects on the screen, such as an item in a menu. Use the buttons to make selections.

Use the pointer, in combination with buttons on the mouse, to perform the following tasks:

- Point to objects on the screen
- Select objects on the screen
- Move objects around on the screen
- Push and pop windows on the screen
- Call menus to the screen
- Switch the keyboard between emulated terminals or windows

- Perform application designated functions

Keyboard

Use the keyboard to perform the following functions:

- Respond to system prompts
- Provide control keys, such as HOLD SCREEN and CYCLE
- Provide special keys, such as HELP
- Enter data and information into a screen window
- Move a cursor in a window on the screen
- Perform application specific functions

1.3.3 Windowing Feature

The graphics software allows you to create and maintain many windows at the same time (based on available resources). Graphics routines create, delete, and manipulate overlapping windows. You can pop windows to the front of the screen, push them to the background, move them around the screen to a new position, or delete them from the screen. You can also control the amount and size of information that appears in a window.

1.3.4 Graphics Capabilities

Routines create new displays and draw graphics within created displays. A display list, which is an encoded description of routines that create the contents of a display, is kept in memory. The display list enables a program to pan and zoom portions of a display easily without redrawing the entire display. Graphics software automatically scales the display. A display, or a portion of a display, can be mapped into one or more windows on the screen.

2 Display Management Concepts

2.1 Overview

This chapter discusses basic concepts involved in creating a graphic object and displaying it on the workstation screen. This chapter covers:

- Virtual displays
- Display windows
- Display viewports
- World and device coordinates
- Display window and viewport scaling

2.1.1 Summary

VMS workstation graphics software enables application programs to build graphic objects and display them on the workstation screen.

An application program that takes full advantage of VMS workstation graphics capabilities can perform the following tasks:

- Create a virtual display
- Draw graphics and text into the virtual display
- Open windows into the virtual display for viewing on an output device
- Map windows into display viewports on the workstation screen
- Manipulate windows and viewports to display as much or as little of the virtual display as desired
- Pan, zoom in and out, resize, and duplicate display windows
- Manipulate display lists

The application program must first create a *virtual display* in which to build the object. Think of a virtual display as a conceptual display space that has no actual physical size or shape. This conceptual display space, called the *world coordinate system*, is defined by the application program in terms of *world coordinates*. World coordinates are arbitrary units of measure selected by the application program that specify locations (or points) in the world coordinate system using values convenient to the application.

The graphics software automatically translates world coordinates to *normalized coordinates* before it maps them to an output device. Normalized coordinates convert world coordinates into a single device-independent coordinate system so you do not have to deal with several coordinate systems. Normalized coordinates are automatically mapped to the device-dependent coordinates of the physical output device.

A graphic object constructed in a virtual display is not available for display on an output device until the application creates a *display window* and *display viewport*.

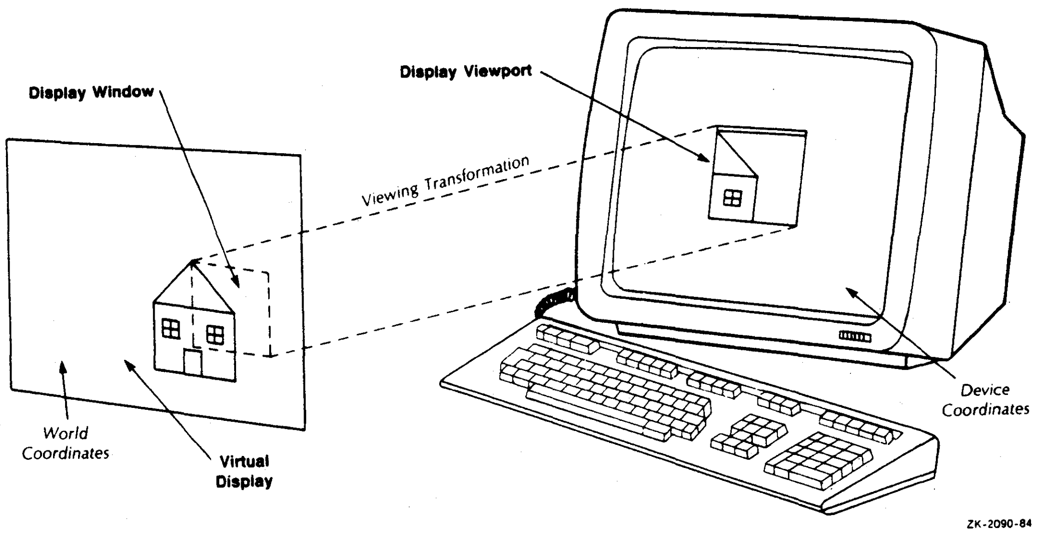
A display window defines what portion of the virtual display graphic object is visible. By creating the display window, the program makes information in the virtual display potentially visible to the user. Information in the display window is not actually visible until the display window is mapped to a display viewport.

A display viewport is the user-controlled, physical region on a screen created by VMS workstation software. The display viewport is the physical representation of the display window mapped to it. It enables you to view the graphic object inside the display window. Figure 2–1 illustrates the relationships among the virtual display, display window, and display viewport.

You use *physical device coordinates* to map a display window to a display viewport. Physical device coordinates are points on the display screen used to locate the graphic object. *Viewing transformation* is the process by which the system maps a graphic object from world coordinates of the display window to device coordinates of the display viewport. The graphics software automatically processes viewing transformations.

To *pan* and *zoom* the graphic object in the display viewport, you can manipulate the world coordinates of the display window in relation to the world coordinates of the virtual display.

Figure 2–1 Virtual Display, Display Window, and Display Viewport



2.2 Coordinate Systems

Think of the VMS workstation graphics environment as a two-dimensional plane. Within this environment, use the *Cartesian coordinate system* to describe points. Cartesian coordinates take the form of x,y, where *x* is the horizonal axis and *y* is the vertical axis. Use a coordinate pair to specify a point on this plane. *Coordinate space* is the area of this plane specified by coordinate pairs.

The VMS workstation graphics software uses four Cartesian coordinate systems: world, normalized, absolute, and viewport-relative device coordinates.

2.2.1 Device-Independent Coordinate Systems

Device-independent coordinate systems mediate between the requirements of the application program and graphics subsystem versus those of the output device.

2.2.1.1 World Coordinates

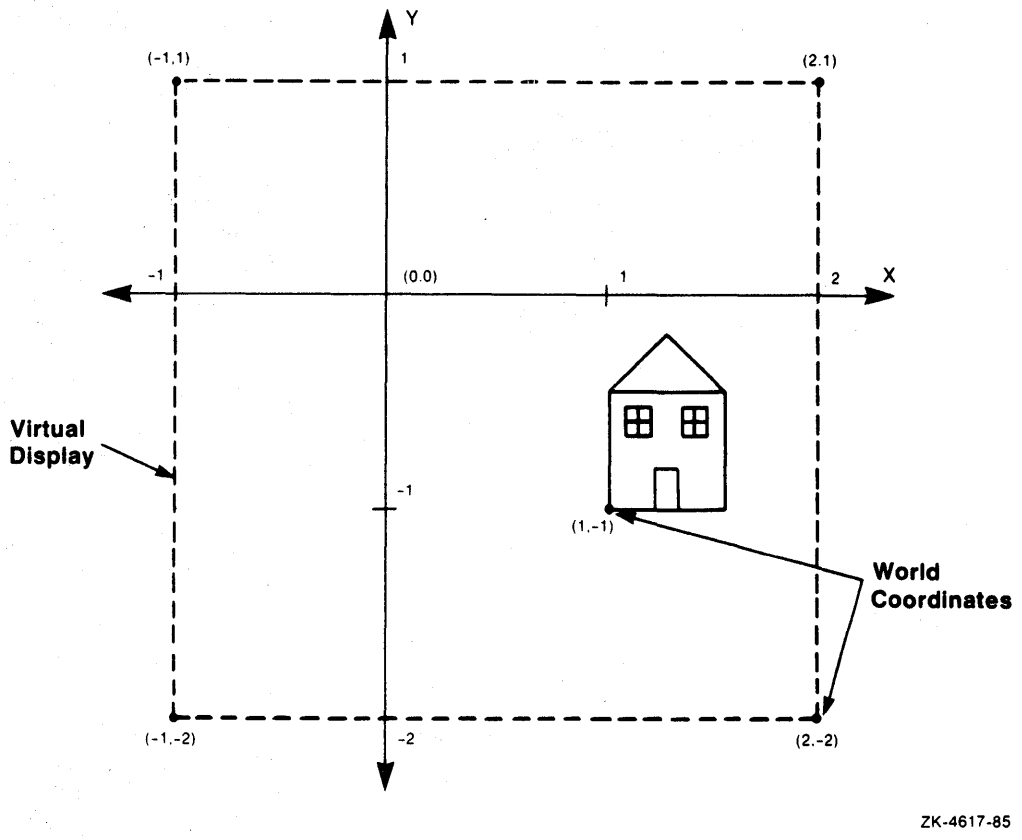
An application program uses world coordinates to describe a virtual display and to build a graphic object within it. Initially, the application program creates a virtual display and specifies a convenient world coordinate system to use when referring to the virtual display. Next, the program uses the same coordinates to specify size and location of objects to be created within the virtual display.

World coordinates are device-independent Cartesian coordinates specified by the application program. They provide a means of locating points in a virtual display. The range of world coordinate values is specified when the virtual display is created. Thus, the virtual display can be created to any proportions selected by the application program. World coordinate values are given as floating-point numbers.

The world coordinate system can represent any unit of measure. When application programs construct a graphic object, world coordinates enable them to use convenient increments of measurement. If the program accesses information from a data base, it could specify meaningful world coordinates for the data used. For instance, if an application draws a chart that shows holiday season product sales, the application could use convenient measurements that represent units sold in thousands versus time in weeks. Or, if the application program draws a graphic object, it could use measurements that make sense for the object. Logically, a virtual display with a map of the United States might use world coordinates that represent measurements in miles or kilometers. A floor plan of a house might use world coordinates that represent feet and inches or meters and centimeters.

Figure 2–2 shows a world coordinate system that describes a virtual display in which an object has been constructed.

Figure 2–2 World Coordinate System and Virtual Display



2.2.1.2 Normalized Coordinates

Normalized coordinates are device-independent coordinates defined by the graphics software. They describe the virtual display in physical terms that any output device can use. An output device cannot use the arbitrary world coordinates that an application program uses to describe a virtual display. Instead, each type of output device has its own device-specific coordinates to locate and build the graphic object. Normalized coordinates provide a means for the graphics software to normalize these different coordinate systems so that a graphic object can be mapped from a virtual display to any output device.

Application programs do not directly manipulate normalized coordinates. Rather, the graphics software internally uses normalized coordinates, mapping them into device-specific display coordinates.

Normalized coordinates provide a way to delay the mapping of application program world coordinates to device-specific coordinates until the actual output device is established.

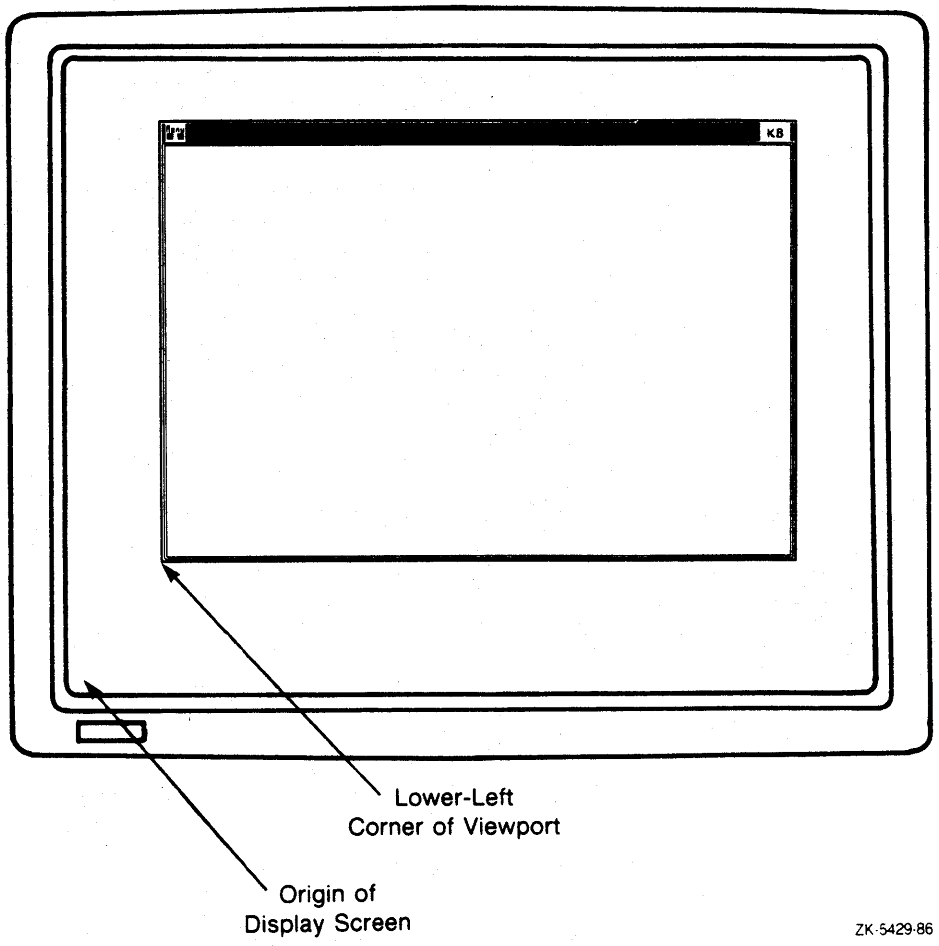
2.2.2 Device-Dependent Coordinate Systems

Output devices use device-dependent coordinate systems to map graphic objects on the display screen or to print objects on a printer. Device-dependent coordinates are physical device coordinates that denote some physical unit of measure such as pixels, centimeters, or inches. Such physical device coordinates reflect device-dependent mapping and drawing characteristics of the output device.

2.2.2.1 Absolute Device Coordinates

Absolute device coordinates are physical, device-dependent Cartesian coordinates that specify a position on the VMS workstation display screen. The position is specified in centimeters relative to the lower-left corner of the display screen. Typically, viewport placement, pointer position, and tablet placement use absolute coordinates. Figure 2–3 illustrates viewport placement on the VAXstation screen.

Figure 2–3 Absolute Device Coordinates



2.2.2.2 Viewport-Relative Device Coordinates

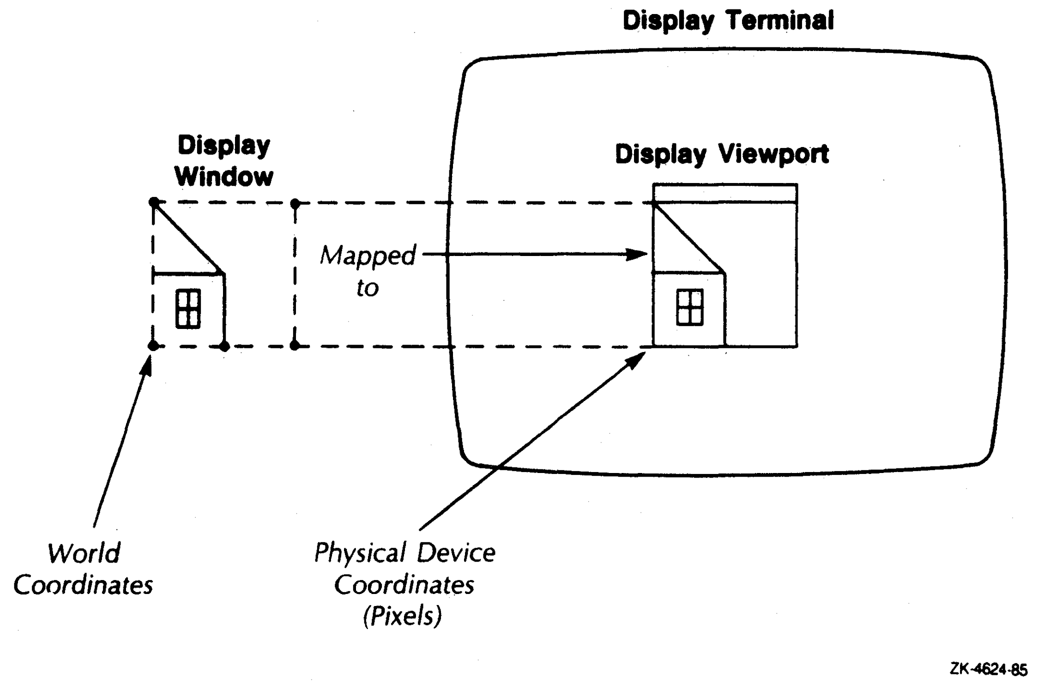
Many VMS workstation graphics software routines use a special type of physical device coordinates called *viewport-relative* device coordinates, which specify positions within a display viewport relative to the lower-left corner of the viewport. Viewport-relative device coordinates are always positive and specified in *pixel* units. A pixel is the smallest unit displayed on a screen. VMS workstation graphics software maps display windows to the display screen.

Viewport-relative device coordinates are used to map graphic objects from a display window to a display viewport on a physical display device.

Before you can display a graphic object in a display viewport on a screen, you must transform the world coordinates of the object to the viewport-relative device coordinates of the screen.

Figure 2–4 shows an object in a display window being mapped to a display viewport on a physical display device. In this figure, the world coordinates of the display window undergo a viewing transformation to the physical device coordinates of the display device.

Figure 2–4 Mapping a Display Window to a Display Viewport



2.3 Virtual Displays

A virtual display is a conceptual space an application program creates in which to construct graphic objects. The application program writes all text and graphics output to a virtual display.

A virtual display has no physical size (dimensions of length and width). Therefore, objects constructed in a virtual display also have no actual physical dimensions. You cannot measure a virtual display or the graphic objects within it. Rather, a virtual display and the objects within it have relative sizes and proportions. The *aspect ratio* of an object in a virtual display is a comparison of the relative proportions of the object's vertical and horizontal components. Use aspect ratio to refer to an object's relative size in a virtual display.

To create a virtual display, an application program specifies a coordinate range in the world coordinate system. The coordinate range establishes the relative size, or aspect ratio, of the virtual display. Objects constructed in the virtual display are specified in terms of world coordinates and have an aspect ratio. Later, the aspect ratio affects how the virtual display and the objects it contains map to the display window.

Refer to Figure 2–2, which shows a graphic object in a virtual display. Both the virtual display and the graphics object are specified in terms of world coordinates.

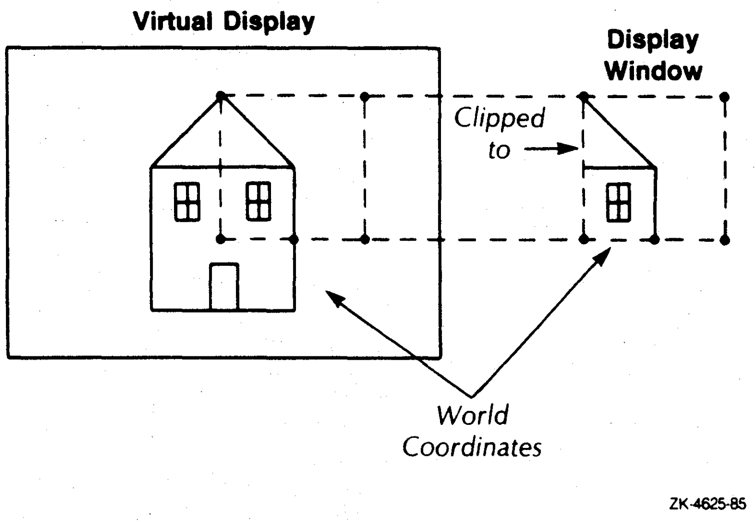
2.4 Display Windows

A display window shows all or part of the contents of a virtual display. Display windows are created by an application program to control how much of a virtual display is potentially available to view. A display window can be the size of an entire virtual display or just a small portion of it. One or several windows in a virtual display can be active at the same time.

An application uses world coordinates to specify the relative proportions and location of a display window. Therefore, the amount of virtual display encompassed by a display window is relative to the virtual display world coordinates. When it specifies the proportions and location of the display window, an application program determines what portion of the graphic object within a virtual display is visible.

World coordinate boundaries of a display window define a *clipping rectangle.* Any graphic object inside the clipping rectangle is potentially visible in the display viewport. Objects outside the clipping rectangle are not visible and are clipped from the window as illustrated in Figure 2–5.

Figure 2–5 Display Window in a Virtual Display



2.5 Display Viewports

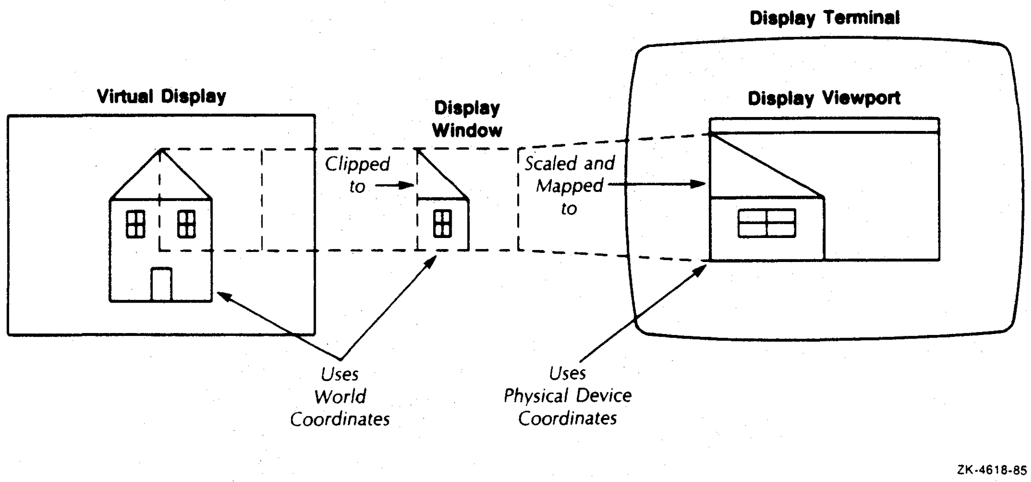
A display viewport is the area of the display screen where a display window is mapped. It can vary in size and shape and be anywhere on the display screen.

Based on available resources, you can have as many viewports as you want on the screen at a time. Viewports *occlude* in areas where they overlap. The last viewport created is on top and visible. However, you can modify which viewport is on top at any one time.

Normally, the graphics software automatically maps and scales the display window to the display viewport on a one-to-one basis. That is, the boundaries of the display viewport implicitly default to the same size and shape as the display window. However, the application program can explicitly set the display window (or display viewport) to a different size or shape than that of the display viewport (or display window). The effects of such manipulation are discussed in the following sections of this chapter.

Figure 2-6 illustrates the relationships among the virtual display, the display window, and the display viewport. This figure shows how a graphics object in a virtual display is clipped to the display window, scaled and mapped into a display viewport, and displayed on a display device such as a terminal screen.

Figure 2-6 Displaying a Graphic Object



2.6 Display Window and Viewport Scaling

You can manipulate the relative sizes of the display window and the display viewport to magnify or reduce graphic objects. The following list describes this manipulation.

Zooming

To zoom (magnify) the graphic object:

- Decrease display window size; do not alter viewport size
- Increase display viewport size; do not alter window size

Reducing

To reduce the graphic object:

- Increase display window size; do not alter viewport size
- Decrease display viewport size; do not alter window size

Panning

To pan the graphic object, move the display window within the virtual display; do not alter the display viewport.

Changing View Size

To change the area of the virtual display being viewed, without performing scaling:

- To increase the virtual display area being viewed, expand both the display window and the display viewport proportionately.
- To decrease the virtual display area being viewed, contract both the display window and the display viewport proportionately.

2.6.1 Distortion of Graphic Objects

Factors that determine whether a graphic object is distorted when it is mapped to the screen are:

- Virtual display aspect ratio
- Display window
- Display viewport

Width to height, the display viewport can have any specified proportions (within the limits of the display device). If the proportions of the display viewport do not match the proportions of the display window, the graphic object appears to be stretched or squeezed as the graphics software attempts to fit the display window to the display viewport. (The exact effect depends on proportional differences between the viewport and window.)

Transformation affects different objects in different ways.

- Straight lines remain straight, but can differ in length and slope, depending on window size and coordinate system.
- Curved lines can change shape, depending on the characteristics of the graphic object and the mapping (transformation) from display window to viewport.
- Arcs change shape and size. For instance, an ellipse can change its proportions.
- Graphics text (specifically character size and spacing) does not adjust to fit the required number of characters into the display viewport. The size and spacing of text characters is fixed and will not distort. However, the starting text position might change, depending on the transformation between window and viewport.

You can correct distortion. The application program can create a display viewport with proportions appropriate to a particular graphics window in world coordinate space. Because the display window can have any proportions in world coordinate space, you can create a properly-proportioned display viewport for a window that is square, tall and narrow, short and wide, or any other shape.

2.7 Display Lists

A display list is a device-independent encoding of the exact contents of a virtual display. The graphics software maintains and uses display lists as follows:

- Automatic management of panning, zooming, resizing, and duplicating display windows
- Structuring virtual display objects
- Simultaneous viewing of objects in a virtual display within several display viewports
- Storing and reexecuting UIS pictures
- Editing UIS pictures

2.8 Generic Encoding and UIS Metafiles

Whenever a graphic object is drawn in the virtual display or an attribute is modified, an encoded entry of the object or attribute modification is added to the display list.

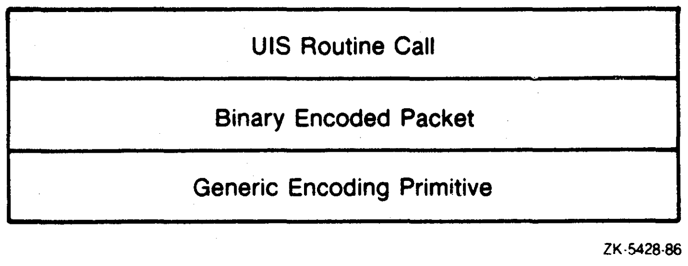
Because of these list entries, an application can extract output from a virtual display, transfer it to an intelligent application, or store it in a *metafile,* which is a *generically encoded* file or buffer, then later execute the generically-encoded binary stream into a new virtual display.

Generic encoding is device-independent.

When UIS routines execute, a binary-encoded packet of values is constructed and stored as display list entries. When the binary-encoded packet is extracted from the display list, it becomes a generically-encoded UIS metafile. Such metafiles can be reexecuted to invoke the appropriate generic encoding routines.

Figure 2-7 shows a display list extraction.

Figure 2-7 Display List Extraction



Many UIS routines have corresponding generic encoding primitives. However, this does not ensure a one-to-one mapping between UIS routines and generic encoding routines or between the UIS routine arguments and generic encoding routine arguments.

3 Graphic Objects and Attributes

3.1 Overview

This chapter discusses the basic building blocks used to construct graphic objects in a virtual display:

- Text and graphics routines
- Attributes and attribute modification routines
- Attribute blocks
- Segments

3.2 Summary

Text and graphics routines (sometimes called output routines) are the fundamental building blocks an application program uses to create graphic objects. These routines specify lines, circles, text, or other graphic objects.

Attributes are values that define various characteristics about the appearance of a text or graphic object. Attributes define how displayed text objects or graphic objects look.

An *attribute block* is a set of attributes. Every text and graphics routine used by an application program must specify an attribute block. The attribute block defines an object's attributes.

An application program uses *attribute routines* to specify or change the current value of an attribute. The changed attribute value affects subsequent text and graphics routines that use the attribute block. You must use an attribute routine to specify which attribute block is affected.

Application programs can group associated attribute, graphics, and text routines together into a *segment*. Segments give the program a convenient way to view several attribute, graphics, and text routines as a single unit.

An application program uses *application-specific data* to associate graphics and text routines or even entire segments. The application program can store application-specific data in the generic encoding stream. In this way, if a portion of a display screen is copied, stored, and later used (restored), the program will be able to associate internal information with the graphic object.

3.3 Graphics and Text Routines

Graphics and text routines map objects directly into the virtual display. You can use these routines to create new objects or modify existing ones. Application programs use graphics and text routines to draw lines, circles, text, and other graphic objects. You can combine these routines to form a desired graphic object.

Each graphics and text routine has two required arguments: one that specifies the virtual display where you draw a graphic object and another that specifies the attribute block to use when you draw the graphic object.

How a graphics or text routine draws a graphic object is strongly influenced by the attributes associated with it.

3.4 Attributes

Attributes define the appearance characteristics of graphic objects created by graphics and text routines. Attributes influence the way a graphic object appears on a display device. Attributes can determine color intensity, style, mode, and width, to name a few.

When you specify attribute values, they remain the same until you explicitly change them. For example, if the application program changes line width, all lines are drawn to the new thickness until the program changes the line width again.

Each type of graphic and text object has a set of unique attributes. For example, attributes that affect graphics do not affect text, and *vice versa*. Certain general attributes, however, affect all routines. For example, the background has an attribute you can set to determine background appearance. Think of the background as all parts of a display not covered by an object created by a graphics or text routine.

Attributes fall into the following general categories:

- General attributes
- Text attributes
- Graphics attributes
- Window attributes

3.4.1 General Attributes

All types of graphics and text routines have general attributes, which include:

- Writing color
- Background color
- Writing mode

Writing Color

This attribute assigns the writing color. It is used by all graphics and text routines (such as lines, text, and so on). To express this attribute, specify an index into a color map.

Background Color

This attribute assigns the background color. To express it, specify an index into a color map.

Writing Mode

This attribute assigns the mode of writing text or graphics. In particular, writing mode determines how a text or graphics routine will use the writing and background colors to display a graphic object.

3.4.2 Text Attributes

Font set

This attribute specifies the font set used to define text characters. *Fonts* express the size and shape of the characters in physical dimensions. This attribute uses display routines during text plotting to enable proper-size text to display. You can choose from a variety of multinational character set fonts and technical character set fonts.

Character spacing

This attribute defines character spacing for width and height of character sizes. It is the additional unit of increment beyond the normal character size for highly spaced characters. You specify this attribute as a floating-point number. Multiply it by the normal character size to produce the actual spacing distance. If you specify zeros, no additional spacing is performed. If you use negative values, the spacing is reduced instead of increased.

NOTE: **In some cases, negative values for this attribute cause the characters to overlap.**

Text Path

Text path is the direction of text drawing. The text path attribute consists of two parts—the major path and the minor path. *Major path* refers to the direction in which characters are drawn on a line. *Minor path* refers to the direction used for beginning a new line of text. The following table lists available major and minor paths.

- Left to right (default major text path)
- Right to left
- Bottom to top
- Top to bottom (default minor text path)

Text Slope

Text slope is the angle between the actual path of text drawing and the major text path. The actual path of text drawing connects the baseline points of each character cell.

Text Margins

This attribute specifies a starting margin and the *x* coordinate distance to the ending margin.

Text Formatting

This attribute and the text margins attribute position text as follows:

- Flush against either or both margins
- Centered
- No formatting at all

UIS supports four types of text formatting modes:

- Left justification
- Right justification
- Center justification
- Full justification

Character Rotation

Individual characters rotate counterclockwise from 0 to 360 degrees. The angle of rotation is the angle between the baseline vector of the character cell and the actual path of the text drawing.

Character Slant

This attribute specifies the angle between the up vector and baseline vector of the character cell. You can express the character slant angle as a negative or positive value.

Character Size

Character scaling allows you to increase the height and width of characters in the virtual display.

3.4.3 Graphics Attributes

Graphics or line attributes affect graphic objects such as lines, polylines, polygons, rectangles, arcs, and curves. These attritubes control filling of objects and determine line style and width.

Current Line Drawing Width

This attribute sets line width in terms of world or device coordinate units. You specify line width as a floating-point number, either interpreted as a world coordinate width or multiplied by the standard line width for a device to produce the desired line width.

Line Style

This attribute, a bit vector that indicates the color of each pixel drawn, sets the current line style of line routines. You can designate the color the same as either the foreground or the background. You repeat bit vector as often as necessary to draw all the pixels in the line.

Fill Pattern

This attribute specifies the fill character to be used for filling closed figures such as polygons, circles, and ellipses. Fill pattern is specified both as a font file and as the index of a character in that font file. You use the pattern defined by the character to fill the figure. Refer to Appendix D for further information about fill patterns.

Arc Type

This attribute specifies how to close an open arc of a circle or ellipse. This attribute can have the following values:

- Open—The arc is not closed off.
- Pie—Two radii are drawn from the endpoints of the arc to the centerpoint (forming a pie shape).
- Chord—A line is drawn between the two endpoints of the arc, connecting them.

3.4.4 Window Attribute

Clipping Rectangle

The clipping rectangle is the visible area of a virtual display. Define the clipping rectangle as the corners of a world coordinate rectangle to which all drawing operations are clipped. Objects or parts of objects outside the clipping rectangle are not visible.

3.5 Attribute Blocks

An attribute block is a set of attribute values that describe the appearance of any graphic object created by an application program. Each attribute block contains attributes for graphics, text, and general display characteristics.

You can address up to 256 different attribute blocks at a time. You address them with numbers from 0 to 255. Application programs assign and use attribute block numbers.

3.5.1 Attribute Block 0

Attribute block 0 is a special attribute block specified by the graphics software. This attribute block contains a standard set of text and graphics attributes. The application program cannot modify the attributes in this block.

Attribute block 0 is read only. There is no convention on the naming and usage of attribute blocks, with the exception of attribute block 0. The graphics software reserves it as a default attribute block.

Attribute block 0 provides default attribute values for an application program to use. Also, you can use it as an attribute block template to create alternate attribute blocks.

3.6 Segments

A segment consists of an attribute block and graphics and text objects. With a segment, an application program can use a special attribute without knowing if particular attribute blocks are being used by other parts of the program. Also, with a segment, an application program can implement transformations either on a per-segment basis or on the entire segment tree. Segments provide programming convenience and increased modularity.

Nested Segments

You can nest a segments. Each nested segment uses the current set of attribute blocks of higher level segments. This feature makes it easier to create segments without having to redefine attribute blocks. However, modifications of attribute blocks in a segment do not affect the attribute blocks of higher-level segments.

Extracting and Re-executing Segments

An application program can take the contents of a file that contains a display list of a virtual display and execute it into another virtual display as a segment. The attributes of the original virtual display should not affect the inserted virtual display segment.

3.7 Viewing Transformations

Viewing transformation is the mapping of the display window to the display viewport. It can affect the appearance of a graphic object on a screen. The shape of the display window and display viewport affect the appearance of displayed text and graphic objects.

3.8 Two-Dimensional Geometric Transformations

Geometric transformations can alter the appearance of graphic objects through scaling, translation, and rotation. These methods all involve manipulation of the object's angular orientation or shape in the virtual display.

Scaling

Scaling is proportional expansion or reduction of graphic objects on the screen. For example, if the display window and display viewport shapes are disproportional, the graphics software must squeeze or stretch the window to fit the viewport. Distortion of the graphics window causes distortion of the graphic objects in that window. Different graphic objects are affected in different ways. Chapter 2 provides further information about the distortion of graphic objects.

Translation

Points that define the position of graphic object in a coordinate system are *translated* when the object coordinates are changed but the following occur:

- The object does not change its angular relationship with other objects.
- The object does not change its implied angular relationship with the coordinate system.

For example, translation occurs when two lines move in the coordinate system but remain parallel.

Rotation

A graphic object *rotates* when it turns on a pivotal point or axis. The object can rotate with respect to some point on its surface, or it can *revolve* around some external point. To give the appearance of rotation on the display screen, you must first translate the axis of the object to the *origin* or center of the coordinate system.

4 Color Concepts

4.1 Overview

Depending on your VAXstation, you can display graphic objects in black and white (bitonal), grey scale, or color. The VAXstation offers a number of color options. This chapter discusses color concepts and color subsystem features in the following topics:

- Color hardware systems
- UIS virtual color maps
- Miscellaneous color concepts

See Chapter 16 for more information about programming in color.

4.2 Color Hardware Systems

UIS supports three types of VAXstation hardware systems:

- Monochrome or bitonal—Displays black and white only
- Intensity—Displays shades of grey or achromatic color
- Color—Displays shades, tints, hues, or chromatic colors

4.3 Raster Graphics Concepts

The VAXstation display screen consists of a set of picture elements called *pixels*. Pixels are the smallest displayable unit of a graphic object. The rectangular set of pixels on the VAXstation screen is a *raster*. To write graphic objects, you illuminate the necessary pixels along the path of points that geometrically describe the object. Each pixel has an address and a binary value associated with it. Pixel values determine graphic object color.

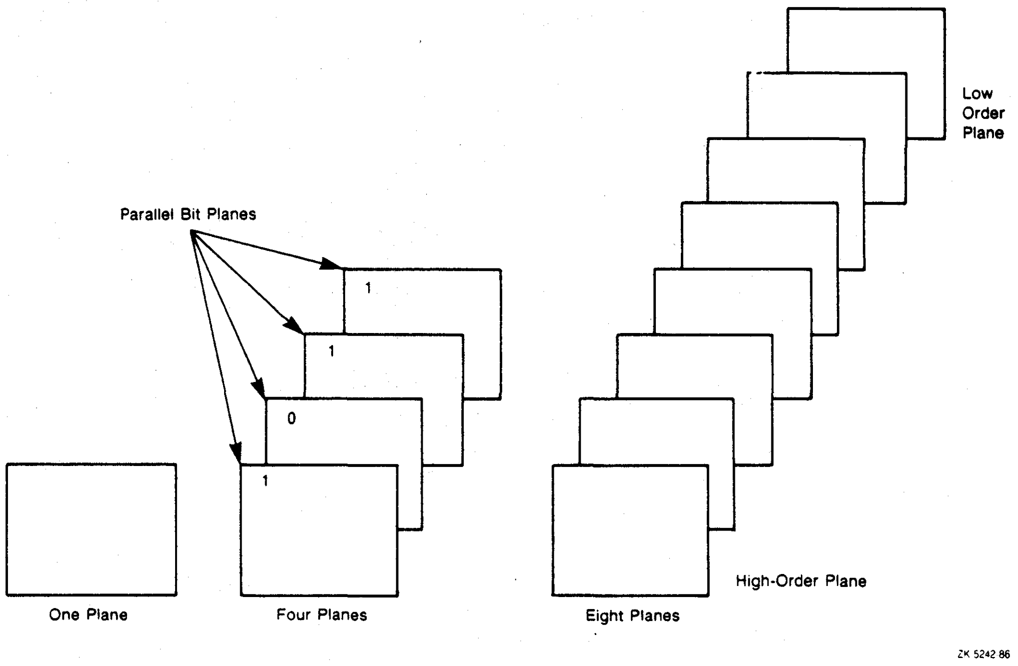
4.3.1 Hardware Interpretation of Pixel Values

The number of possible pixel values depends on the number of bit planes or planes of memory that the system hardware supports. A plane is an allocation of memory in which each bit maps to a pixel on the display screen. Conversely, each pixel has an address in memory. The following table shows the relationship between the number of hardware-supported planes and the number of possible pixel values.

Workstation	Number of Planes	Number of Possible Values
Monochrome	1	2
Intensity or color	4 or 8	16 or 256

Figure 4–1 shows how pixel values are represented in single and multiplane systems.

Figure 4–1 Bitplane Configuration in Single and Multiplane Systems



In Figure 4–1, a pixel on the VAXstation screen correlates to four corresponding bits in memory on each bit plane of a four-plane system. If the bit settings are arranged as a binary value corresponding to the high- and low-order planes, they appear in the following order: 1011_2.

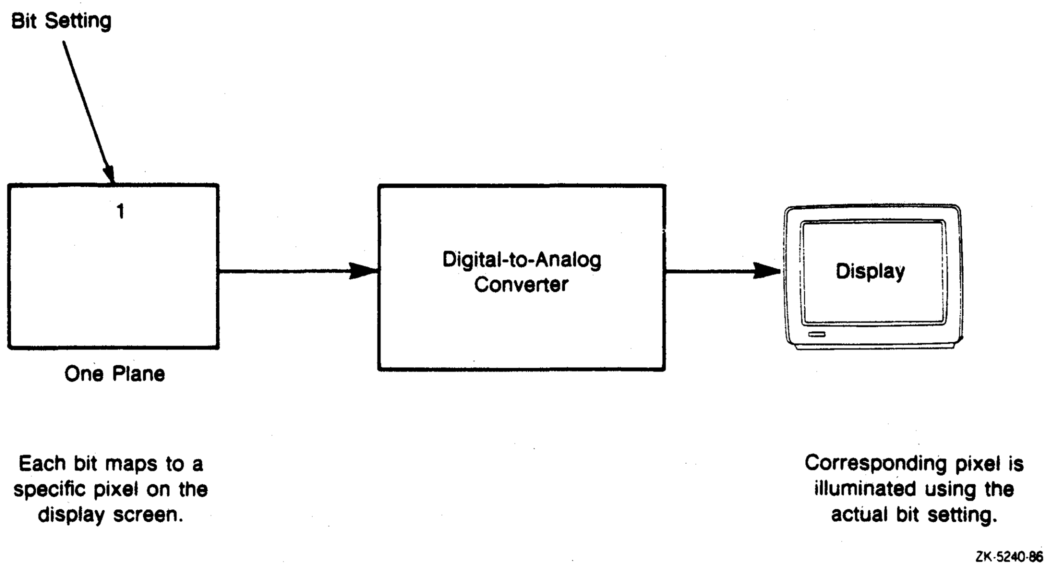
Therefore, the pixel value is 11_{10}. A pixel in a four-plane system can have a maximum of 16 values. You can use the pixel value in two different ways, as a *direct color value* or as a *mapped color value*.

Direct Color Value

If the pixel value is used as a direct color value, each possible pixel value directly specifies a color. In other words, the pixel value goes directly to system hardware (for example, a digital-to-analog converter), where it is used as the actual color value of the graphic object. For instance, the one-plane, VAXstation monochrome system interprets pixel values as direct color values where *0* is black and *1* is white.

Figure 4–2 shows direct color values.

Figure 4–2 Direct Color Values

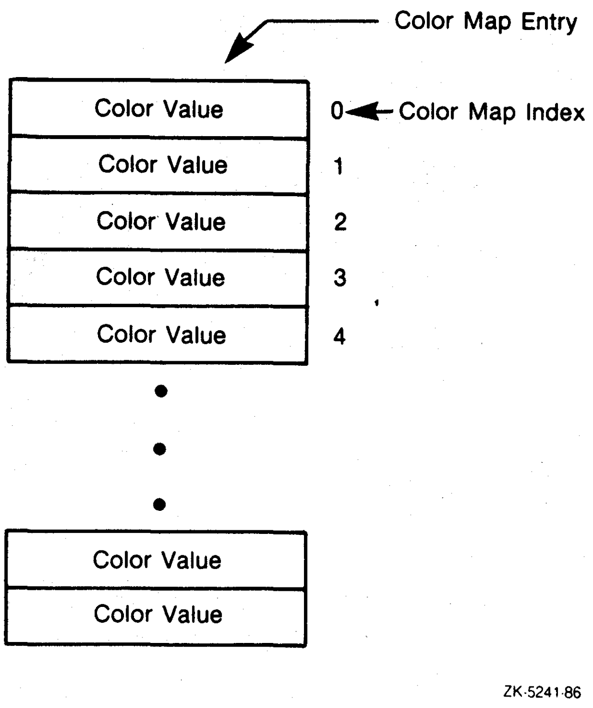

Bit Setting
1
One Plane
Digital-to-Analog Converter
Display
Each bit maps to a specific pixel on the display screen.
Corresponding pixel is illuminated using the actual bit setting.
ZK-5240-86

Mapped Color Value

When pixel values are interpreted as mapped color values, they indirectly specify an actual color value located in a hardware *color look-up* table or hardware color map. Figure 4–3 shows a hardware color map.

The pixel value is an *index* to an entry in the color map.

Figure 4–3 Hardware Color Map



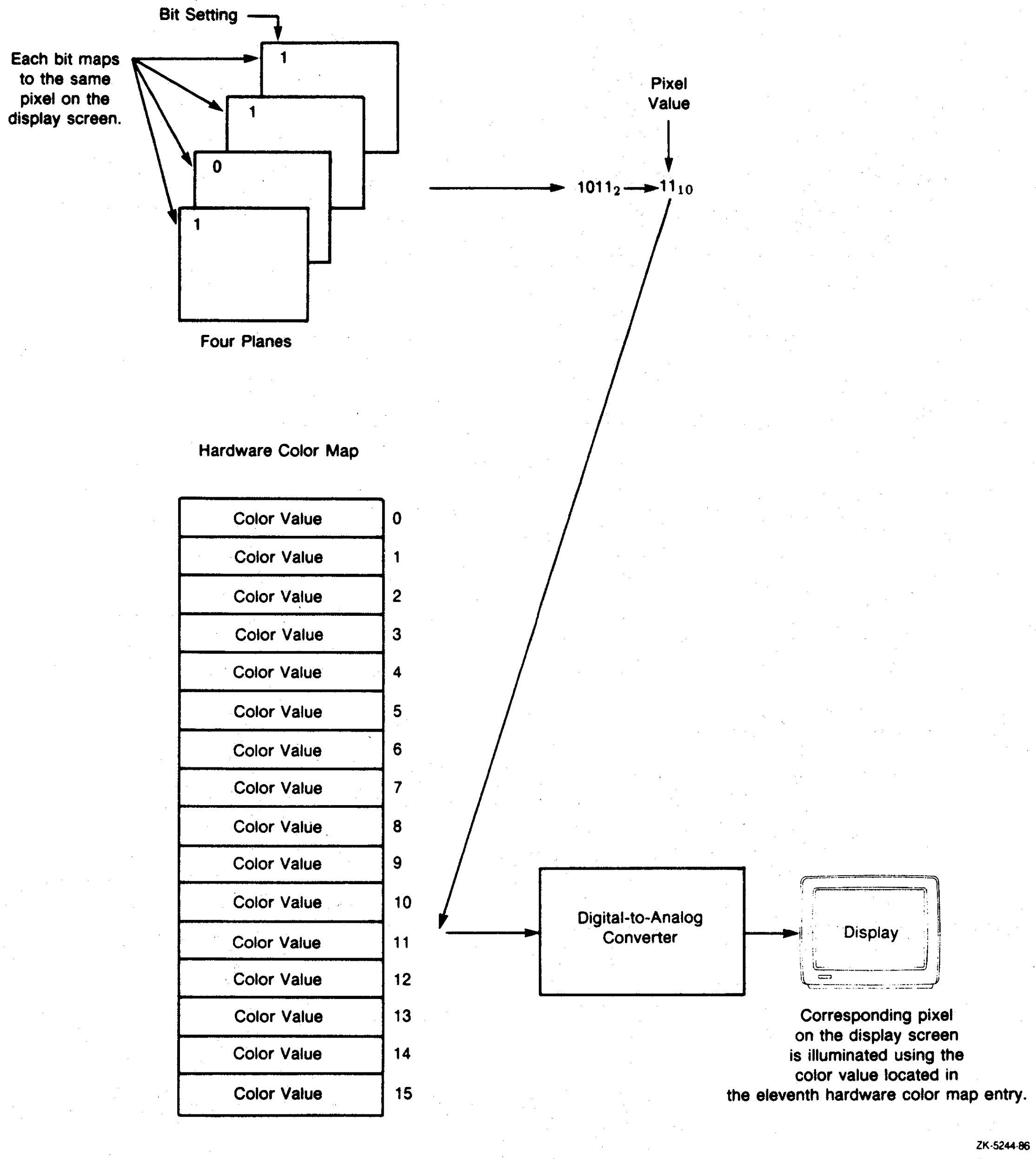
The hardware color map is the same size as the number of possible pixel values; it has the maximum number of colors that can be displayed simultaneously. Table 4–1 lists the size of the hardware color map in intensity and color systems.

Table 4–1 Hardware Color Map Characteristics

System	Number of Planes	Number of Entries
Intensity	Four	16
	Eight	256
Color	Four	16
	Eight	256

For example, an eight-plane VAXstation intensity (color) system has a hardware color map with 256 entries. Each color map entry contains color values that are *RGB color components* and that define the desired color. Each hardware color map entry contains a color value for each pixel. Conversely, the value of each pixel is the hardware color map index of a color map entry with the actual color value. Use this color value to illuminate the pixel on the VAXstation screen. Figure 4–4 shows mapped color values in a four-plane system.

Figure 4–4 Mapped Color Values in Four-Plane System



4.3.2 Color Representation Models

You express color values according to the requirements of the particular color representation model used. Three well-known color representation models are:

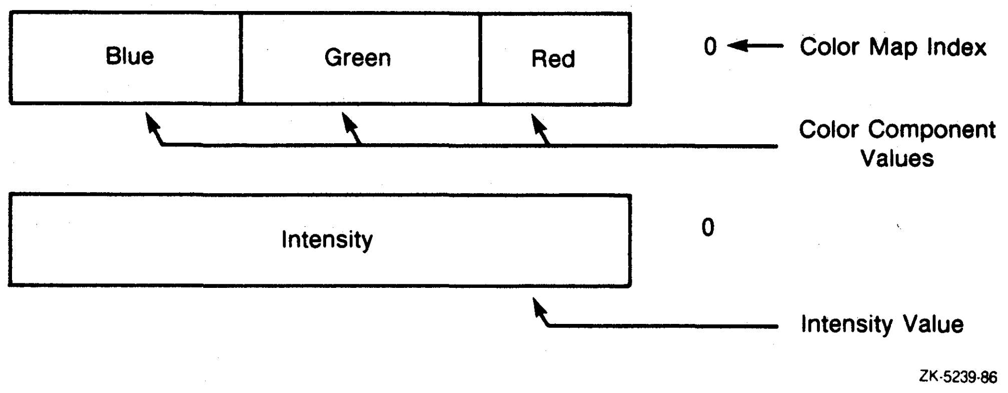
- Hue lightness saturation (HLS)
- Hue saturation value (HSV)

- Red green blue (RGB)

The UIS base color model is the RGB model. RGB color values range from *0.0* to *1.0*. Red, green, and blue color component values compose a single color value on a VAXstation color system.

Specify intensity values (the color values associated with shades of grey), as a single value in the range *0.0* to *1.0*. Figure 4–5 shows RGB and intensity color values as hardware color map entries.

Figure 4–5 RGB and Intensity Color Values as Hardware Color Map Entries



4.3.3 Color Palette

The *color palette* is the number of possible colors you can specify. Table 4–2 shows the color palette available on each color system.

Table 4–2 Color Palette

System	Possible Colors
Monochrome	black and white
Intensity	up to 2^{24} shades of gray
Color	up to 2^{24} chromatic colors

Color Palette Size and Direct Color Systems

On direct color systems, palette size is identical to the number of simultaneously displayable colors. For example, the size of the color palette of a VAXstation monochrome system is two. You can display only two possible colors, black and white, simultaneously on the screen.

Color Palette Size and Mapped Color Systems

On mapped color systems, the palette size is typically much greater than the number of simultaneously displayable colors. The palette size is determined by the precision of color component specification. For example, on a VAXstation color system, you can specify each color component with eight binary bits of precision for each red, green, and blue color component or 2^{24} (16,177,216) possible colors.

4.4 UIS Virtual Color Maps

When an application uses hardware color resources, the hardware color map must be aware of hardware system limitations and color characteristics. It must know the answers to the following questions:

- Is the system direct color or mapped color?
- What is the precision of the color representation values for each RGB color component?
- What is the range of possible pixel values?

The hardware color map contains a finite number of entries (for example, 16 entries in a four-plane system). Concurrent processes executing in the same display space must somehow share system color resources.

Purpose of Virtual Color Maps

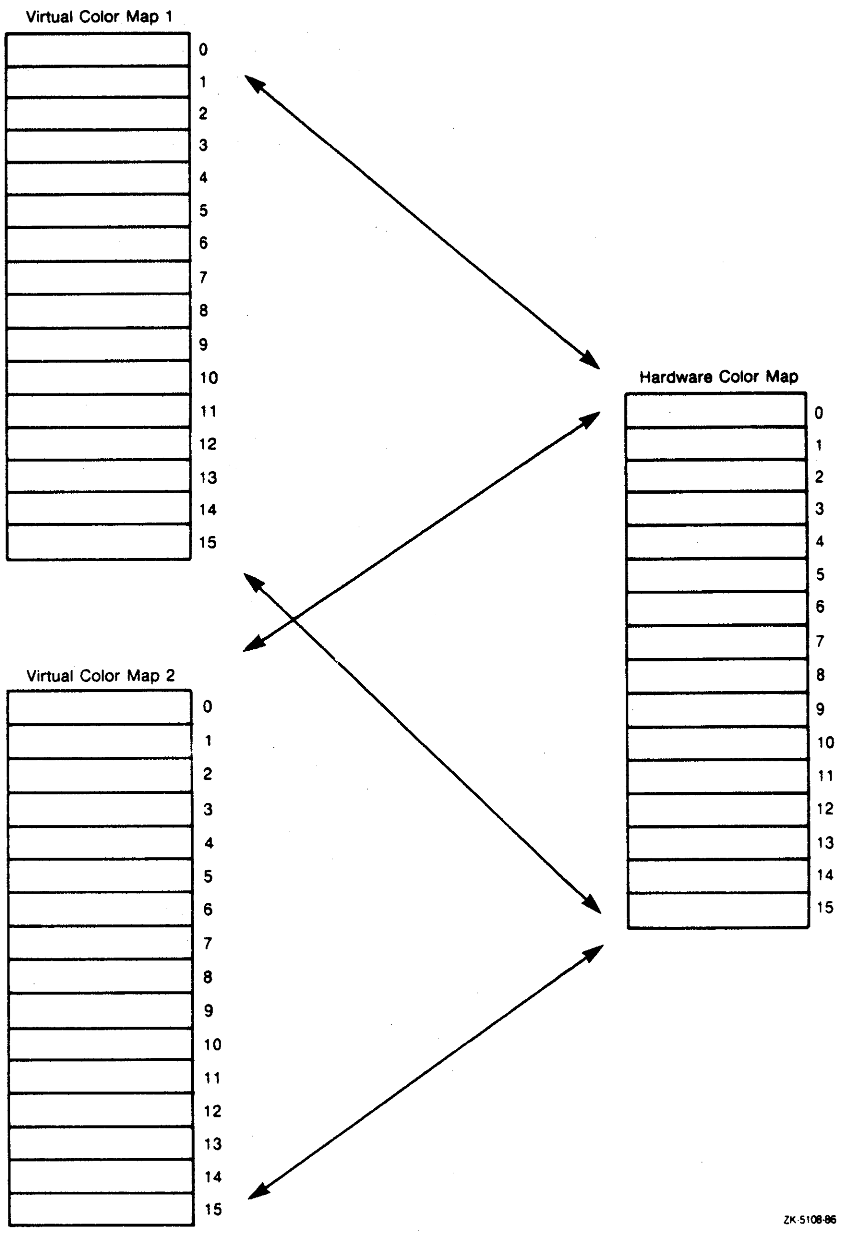
The *virtualization* of the hardware color map solves problems that occur when individual applications require abundant system resources. Virtualization also solves the problem that occurs when many processes compete for finite color resources. The use of virtual color maps is analogous to the use of virtual memory in a multiprogramming environment where many processes must access physical memory.

When concurrent processes collectively require more color map entries than exist in the hardware color map, the color values associated with each competing process are swapped in and out of the hardware color map as *virtual* color maps. Swapping virtual color maps in and out of the hardware color map is a means of arbitrating hardware color map use across applications.

The process of loading or writing values of the virtual color map into the hardware lookup table is transparent to the user. Applications see only a virtual color map, not the underlying hardware resources. Each virtual display has a virtual color map associated with it.

Figure 4–6 illustrates the swapping of two 16-entry virtual color maps into a 16-entry hardware color map.

Figure 4–6 Swapping Virtual Color Maps



Characteristics of Virtual Color Maps

A virtual color map is flexible enough to serve a wide range of applications. Virtual color map size can range from two to 32,768 entries. If you do not specify a virtual color map, a two-entry virtual color map is created by default. The virtual color map size does not have to match that of the hardware color map. Although virtual color maps are potentially shareable among applications, they are private by default. Virtual color maps are *resident*; that is, you cannot swap them in the hardware color map. The following table shows how virtual color map entries are initialized.

Virtual Color Map Entry	Color Value
0	Default window background color
1	Default window foreground color

All other entries are undefined.

UIS transparently reconciles differences between the virtual color map model and the hardware color resources. UIS manages the concurrent use of these resources across applications.

For information about creating and using virtual color maps, see Chapter 16.

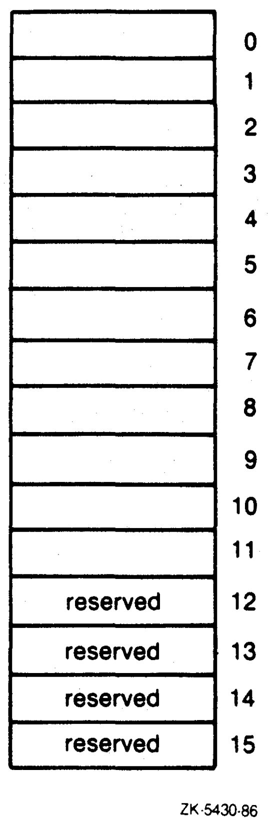
4.4.1 Reserved Hardware Color Map Entries

Because of hardware limitations on mapped color systems, the hardware color system or the UIS window management software preallocates some of the hardware color map entries for special purposes. For example, pointer colors, window background and foreground colors, and display screen color are allocated reserved entries in the hardware color map.

Whenever a virtual color map exceeds the size of the hardware color map less the reserved entries, the results are unpredictable. For more information about how to use the programming interface to obtain the hardware color map characteristics, see Chapter 16.

Figure 4–7 describes reserved entries in a hardware color map in a four-plane system.

Figure 4–7 Reserved Hardware Color Map Entries in a Four-Plane Color System



4.5 UIS Color Map Segments

The use of color map segments represents a device-specific binding of a virtual color map to the underlying hardware color resources, that is, the hardware color map. In a color-mapped color system, color map segments are bound to specific hardware color map entries and swapped in and out of the hardware color map based on system and user events. Usually, applications need not worry about color map segments. UIS handles the device-specific binding automatically. Applications might use color map segments for the following reasons:

- Applications can explicitly control the binding of the virtual color map and the hardware color map.
- Applications are not transported to different hardware configurations, for example, four-plane to eight-plane systems or VAXstation color and intensity systems to VAXstation monochrome systems.

4.6 Shareable Virtual Color Maps

By default, virtual color maps are private. Yet, they can be shared among cooperating application programs to define a uniform color regime and to conserve hardware color map entries. Shared virtual color maps have names, an ASCII string from 1 to 15 characters, and a name space (UIC group or system). For example, UIS uses a system-wide, shared color map to display terminal emulator windows and window and screen menus.

4.7 Miscellaneous UIS Color Concepts

The following sections contain additional information about the UIS color subsystem.

4.7.1 Standard and Preferred Colors

VAXstation color and intensity systems support two sets of symbolically defined colors. Workstation *standard* colors and intensity values are a set of colors used for specific purposes within the workstation environment. For example, the default window background and foreground, cursor background and foreground colors, and the display screen color are the workstation standard colors.

Workstation *preferred* colors are a set of colors that represent user preference for the eight combinations of the RGB primary colors. For example, workstation preferred colors can define a particular shade of red rather than a full intensity red. In an intensity system, preferred colors can define a base white level from which preferred shades of grey are derived. Preferred values are simply a mechanism to conveniently maintain and communicate color preferences to an application.

Use the workstation setup mechanism to set values for standard and preferred colors. Use UIS$GET_WS_COLOR and UIS$GET_WS_INTENSITY to return standard and preferred color and intensity values.

4.7.2 Monochrome (Bitonal), Intensity, and Color Compatibility Features

Use UIS$SET_COLOR or UIS$SET_INTENSITY to change or retrieve color map entries. Both load a single color value in a color map entry and can be used in any of the three hardware color environments—monochrome (bitonal), intensity, or color.

Color System	Compatibility Feature
Monochrome (bitonal)	UIS chooses the color (black or white) closest to the color specified by the application.
Intensity[1]	UIS$SET_COLOR converts the specified RGB values to an equivalent gray level using an equation. UIS$SET_INTENSITY sets the requested gray level directly.
Color[2]	UIS$SET_COLOR sets the requested RGB color values directly. UIS$SET_INTENSITY converts the specified intensity value to an equivalent RGB value using an equation.

[1]The color-to-intensity equation is $I = 0.30R + 0.59G + 0.11B$. Color television broadcasts transmitted for reception by noncolor television sets are processed in this manner.

[2]The intensity-to-color equation is $R = I$, $G = I$, $B = I$.

4.7.3 Color Value Conversion

UIS provides routines to convert color values in applications that use other color representation models.

- Hue lightness saturation (HLS)
- Hue saturation value (HSV)

Hue values range from *0.0* to *360.0*, where red = *0.0*. Values for lightness and saturation range from *0.0* to *1.0*.

4.7.4 Set Colors and Realized Colors

UIS routines that *set* (load) color map entries in the virtual color map accept F_floating point values between *0.0* and *1.0*. The precision of the F_floating point data type is approximately seven decimal places.

The precision for the color representation for a particular device might not be accurate enough to represent the requested F_floating point value. In this case, the *set* color value (F_floating) differs from the *realized* color value (device precision). An application can determine realized color values using UIS$GET_COLOR(S) and including the optional parameter. See Chapter 16 for details.

4.7.5 Color Regeneration Characteristics

The color regeneration hardware characteristic specifies whether changing a color map entry affects the color of existing graphic objects (retroactive regeneration) or only graphic objects drawn after the color map is changed (sequential regeneration).

The following table summarizes regeneration characteristics of direct and mapped color systems.

System	Regeneration Characteristics
Direct color	Usually sequential
Mapped color	Usually retroactive

An application can determine the hardware color regeneration characteristics by calling UIS$GET_HW_INFO.

5 Input Devices

5.1 Overview

This chapter discusses the devices that enable user and application program interaction. Some of the topics covered in this chapter are:

- Pointing devices
- Virtual keyboards
- Physical keyboards

5.1.1 VAXstation Input Devices

Application programs and users interact through input devices. Typical VAXstation input devices are:

- Keyboard
- Mouse
- Tablet

With the keyboard, you can initiate program interaction and respond to application program prompts by pressing a key or entering data. With the mouse and tablet, you can communicate with an application program by pointing to objects or items with a *pointer* and by making selections with buttons.

5.2 Pointers

You can use two types of pointing devices with the workstation, a mouse and a tablet. You can use only one type of pointing device at a time.

Application programs receive input from a pointing device by polling or soliciting interrupts from pointer input routines. Because only one pointer input device can be used at a time, applications use the same set of pointer input routines to receive input from either the mouse or the tablet. The actual pointer input device used is transparent to an application.

The programming interface lets you set the pattern or the position of the cursor that is synchronized with the pointing device.

5.2.1 Mouse

The mouse is a small, hand-held device with three buttons on the top and a roller-ball on the bottom. Associated with the mouse, on the display screen, is an arrow-shaped cursor (or pointer).

You manipulate items on the display screen by using the pointer and buttons. When you move the mouse in any direction on a flat surface, the ball on the bottom turns and the pointer on the screen moves in any direction you choose. You can position the pointer anywhere on the display screen. When you press the buttons on the mouse, you can select items in a menu and perform a variety of other functions.

The mouse is a *relative pointing device*. The mouse reports only its relative movement to the workstation. You can pick up the mouse and place it in a different position without changing the position of the pointer on the screen. Consequently, the workstation keeps track of the current mouse position only when the mouse is moved on a surface.

Application programs can use the mouse pointer in the following ways:

- To create menus from which the user selects items
- To read the position of the pointer and the state of the mouse buttons

The workstation human interface implements menus that allow you to create, select, move, and delete objects on the display screen. Application programs can create menus that do the same things. To select a menu item, move the pointer to the region of the desired item and press one of the mouse buttons. The application program predefines items and specifies the action to be taken when you select an item.

Application programs can detect when the pointer is moved across the boundary of a window or a mouse button is pressed within a window. Programs can also read the current pointer location and current button state. When you move the pointer to the border, or outside, of a screen viewport, the human interface detects interrupts from the mouse. If you position the pointer inside a viewport that is mapped to an application-created window, the application program can receive these interrupts.

5.2.2 Tablet

The tablet is an optional input device that can be used with the workstation. A tablet operates in much the same way as a mouse. An application program uses the same routines to receive information from a tablet as it does for the mouse. This is possible because the actual physical input device being used is transparent to an application program.

The tablet is an *absolute pointing device*. That is, it reports all movement to the workstation. For example, if the pen or stylus is picked up and moved to another position on the tablet, the pointer changes its position on the screen to match the movement.

A tablet is composed of the following parts:

- Tablet
- Puck

- Stylus

Tablet

The tablet is a flat square device with a surface similar to a table top. It is used in conjunction with a puck and/or stylus to locate points on the display screen. When the puck and/or stylus are moved on the surface of the tablet, the pointer on the display screen moves in an identical fashion. If you pick up the puck and place it in different region of the tablet, the pointer on the display screen reflects this change. The tablet has a grid that senses a change in the position of the pen or stylus.

Puck

The puck is a hand-held device that you move on the tablet to locate points on the display screen. The puck has cross-hair markings used for precision in positioning it on the tablet. It also has four buttons that you can use for various purposes, depending upon the application.

Stylus

The stylus is a hand-held device that resembles a pen. You move it on the tablet to locate points on the display screen. The stylus has greater precision than the puck in locating positions. The stylus can also have buttons: usually one is located on the outside of the barrel and one on the tip. The functions of these buttons are application-specific.

5.3 Keyboards

You should be able to distinguish between a physical keyboard (the workstation keyboard) and a *virtual keyboard* (a simulated keyboard).

The physical keyboard is the workstation keyboard. You can press its keys to respond to prompts from the application program, or you can type and enter data into the currently active display window. A workstation can have only one physical keyboard attached to it at any one time.

A virtual keyboard is a conceptual keyboard that does not have an actual physical existence. Rather, a virtual keyboard is a simulated keyboard that exists in software and is associated with a display window. Each application can have one or more virtual keyboards attached to it. Virtual keyboards provide the means for applications to share the single physical keyboard.

5.3.1 Virtual Keyboards

A *virtual keyboard* is a simulated rather than an actual physical keyboard. Virtual keyboards are conceptual in nature and exist only in software. A virtual keyboard has the same relationship to the physical keyboard as a virtual display has to the physical display screen.

By using routines that establish one or more virtual keyboards, application programs can read from the physical (workstation) keyboard, assign the physical keyboard to a display window, and modify the characteristics of a physical keyboard associated with a window. To manipulate the workstation keyboard, applications refer to the established virtual keyboards.

The VAXstation supports multiple windows with multiple processes running simultaneously. At various times, these windows and processes require keyboard input. Consequently, each window needs a keyboard. Because there is only one physical keyboard, applications use *virtual keyboard routines* to share the physical keyboard among several windows.

With virtual keyboards, each window can have its own keyboard. One or more display windows and virtual keyboards can be active on the display screen at a time. However, the physical keyboard can be connected to only one virtual keyboard at a time. A virtual keyboard can be attached to more than one display window at a time. However, each display window can have only one virtual keyboard attached to it.

You control the association between the physical keyboard and the various virtual keyboards that exist at any point in time. To connect the workstation keyboard to different windows, manipulate the display viewports to which the virtual keyboards are connected. When you determine which window the workstation keyboard is attached to, you know which process is receiving keyboard input and thus, which window on the screen is currently active.

The workstation places a small KB icon in the upper right corner of all windows that use the keyboard. This icon is highlighted in the currently active window. An application can restrict windows from receiving keyboard input. Display windows that do not interact with the keyboard do not have the KB icon.

Part II How to Program with VMS Workstation Software Graphics

6 Programming Considerations

6.1 Overview

The User Interface Services (UIS) graphics software package allows you to create application programs that call system routines. With UIS system routines, you can create virtual displays, display windows, viewports, graphic images, and text. You can access these *callable* routines through high-level programming languages, VAX MACRO, and VAX BLISS. Note that the programming examples included in succeeding chapters to illustrate the capabilities of the UIS graphics software are written in VAX FORTRAN.

This chapter discusses the following topics:

- UIS routine calls
- Argument characteristics
- Constants
- Condition values
- Additional program components
- Program execution

Refer to the *VMS Programming Support Manual* for additional information about other callable routines.

6.2 Calling UIS Routines

To draw and manipulate graphic images and text, application programs must contain references or *calls* to specific UIS system routines. Call statements and language-specific function declarations invoke the UIS system routines using the VAX Procedure Calling Standard.

6.2.1 Calling Sequences

The format of a call to UIS, or the *calling sequence*, consists of:

- The elements that make up the statement
- Their positional order

Tables A–1 and B–1 summarize UIS and UISDC calling sequences.

6.2.1.1 Call Type

Typically, application program calls to UIS system routines specify the function name and an argument list as follows:

```
vd_id=UIS$CREATE_DISPLAY(-1.0,-1.0,+1.0,+1.0,width,height)
```

However, some UIS routines are *functions* and return values to the calling program. The preceding example shows such a call from a VAX FORTRAN program. It also returns a value, the virtual display identifier, to the **vd_id** argument. Such return values are stored in variables that are often arguments (where applicable) in subsequent routine calls.

UIS routines that are not functions must be called using an explicit VAX FORTRAN CALL statement.

```
CALL UIS$PLOT(vd_id,1,-1.0,-1.0)
```

Programming languages have no standard call type to invoke UIS system routines. This manual does not describe the syntax of each high-level programming language call. It uses examples of VAX FORTRAN to describe representative call syntax. For information about other language call syntax, refer to the specific language user's guide.

6.2.1.2 Routine Name

When you call a system routine, you must identify it by specifying its routine (or *entry point*) name, for example, UIS$MOVE_AREA. The routine name consists of a symbol prefix that identifies the system facility (UIS$) and a symbol name that indicates what operation it performs (MOVE_AREA).

6.2.1.3 Argument List and Argument Characteristics

The argument list contains parameters to be passed to the UIS routine. This list follows the routine name as a parenthetical expression containing arguments separated by commas. You can substitute your own argument names in place of the formal parameter names. However, whenever you invoke a UIS routine, you must maintain the positional order of the parameters in the argument list, as follows:

```
CALL UIS$CIRCLE(VD_ID,ATB,CENTER_X,CENTER_Y,XRADIUS,START_DEG,END_DEG)
```

You pass data to the called routine via the routine arguments. Keep in mind the characteristics of arguments—VMS Usage, type, access, mechanism.

6.2.2 VMS Usage

The *VMS Usage* entry contains the name of a VMS data type that has special meaning in the VMS operating system environment.

The VMS Usage entry is not a traditional data type such as the VAX standard data types byte, word, longword, and so on. It is significant only within the context of the VMS operating system environment and is intended solely to expedite data declarations within application programs.

Appendix F contains a complete listing of VMS usage entries and implementation charts for each UIS-supported VAX language. The implementation charts describe how to code the VMS usage entry for each programming language.

6.2.3 Type

The *type* characteristic refers to the standard argument data type, that is, whether the argument is a word, longword, floating point number, and so forth. Depending on the programming language, you might have to declare certain data types locally within your program. These structures provide data type definitions for the arguments in subsequent calls to UIS routines.

6.2.3.1 VAX Standard Data Types

When a calling program passes an argument to a system routine, the routine expects the argument to be a particular data type. The routine descriptions in Part III of this manual indicate the expected data types for each argument.

Properly speaking, an argument does not have a data type; rather, the data specified by an argument has a data type. The argument is merely the vehicle to pass data to the called routine.

Nevertheless, the term "argument data type" is frequently used to describe the type of data specified by the argument. This terminology is simpler and more straightforward than the strictly accurate phrase "data type of the data specified by the argument."

Table 6-1 lists data types allowed by the VAX Procedure Calling Standard.

Table 6-1 VAX Standard Data Types

Data Type	Symbolic Code
Absolute date and time	DSC$K_DTYPE_ADT
Byte integer (signed)	DSC$K_DTYPE_B
Bound label value	DSC$K_DTYPE_BLV
Bound procedure value	DSC$K_DTYPE_BPV
Byte (unsigned)	DSC$K_DTYPE_BU
COBOL intermediate temporary	DSC$K_DTYPE_CIT
D_floating	DSC$K_DTYPE_D
D_floating complex	DSC$K_DTYPE_DC
Descriptor	DSC$K_DTYPE_DSC
F_floating	DSC$K_DTYPE_F
F_floating complex	DSC$K_DTYPE_FC
G_floating	DSC$K_DTYPE_G
G_floating complex	DSC$K_DTYPE_GC
H_floating	DSC$K_DTYPE_H
H_floating complex	DSC$K_DTYPE_HC
Longword integer (signed)	DSC$K_DTYPE_L
Longword (unsigned)	DSC$K_DTYPE_LU
Numeric string, left separate sign	DSC$K_DTYPE_NL
Numeric string, left overpunched sign	DSC$K_DTYPE_NLO
Numeric string, right separate sign	DSC$K_DTYPE_NR

Table 6-1 (Cont.) VAX Standard Data Types

Data Type	Symbolic Code
Numeric string, right overpunched sign	DSC$K_DTYPE_NRO
Numeric string, unsigned	DSC$K_DTYPE_NU
Numeric string, zoned sign	DSC$K_DTYPE_NZ
Octaword integer (signed)	DSC$K_DTYPE_O
Octaword (unsigned)	DSC$K_DTYPE_OU
Packed decimal string	DSC$K_DTYPE_P
Quadword integer (signed)	DSC$K_DTYPE_Q
Quadword (unsigned)	DSC$K_DTYPE_QU
Character string	DSC$K_DTYPE_T
Aligned bit string	DSC$K_DTYPE_V
Varying character string	DSC$K_DTYPE_VT
Unaligned bit string	DSC$K_DTYPE_VU
Word integer (signed)	DSC$K_DTYPE_W
Word (unsigned)	DSC$K_DTYPE_WU
Unspecified	DSC$K_DTYPE_Z
Procedure entry mask	DSC$K_DTYPE_ZEM
Sequence of instruction	DSC$K_DTYPE_ZI

Refer to the *VMS Programming Support Manual* for more information about VAX standard data types.

6.2.4 Access

The *access* characteristic describes how a calling routine uses argument-specified data. A list of the most common types of argument access follows.

- Read only access—The UIS routine uses the data specified by the argument as input only.
- Write only access—The UIS routine uses the argument as a location to return data only.
- Modify access—The UIS routine uses the data specified by the argument as input for its operation and then writes data to that argument.

6.2.5 Mechanism

VAX language extensions provide the means to reconcile the various argument-passing mechanisms within a programming language. The VAX Procedure Calling Standard provides three ways for application programs to pass arguments to a system routine.

- By value—The argument contains the actual data to be used by the routine; the actual data is said to be passed to the routine *by value.*
- By reference—The argument contains the address of the location in memory of the actual data to be used by the routine; the actual data is said to be passed to the routine *by reference.*
- By descriptor—The argument contains the address of a descriptor; the actual data is said to be passed *by descriptor.*

  Depending on its type, a descriptor consists of two or more longwords that describe the location, length, and data type of the data to be used by the called routine.

All language processors (except VAX MACRO and VAX BLISS) pass arguments by default by reference or by descriptor. Some high-level languages, including VAX FORTRAN, set up the descriptors and arrays individually.

The following table lists VAX Procedure Calling Standard passing mechanisms.

Passing Mechanism	**Descriptor Code**
By value	
By reference	
By reference, array reference	
By descriptor	
By descriptor, fixed-length	DSC$K_CLASS_S
By descriptor, dynamic string	DSC$K_CLASS_D
By descriptor, array	DSC$K_CLASS_A
By descriptor, procedure	DSC$K_CLASS_P
By descriptor, decimal string	DSC$K_CLASS_SD
By descriptor, noncontiguous array	DSC$K_CLASS_NCA
By descriptor, varying string	DSC$K_CLASS_VS
By descriptor, varying string array	DSC$K_CLASS_VSA
By descriptor, unaligned bit string	DSC$K_CLASS_UBS
By descriptor, unaligned bit array	DSC$K_CLASS_UBA
By descriptor, string with bounds	DSC$K_CLASS_SB
By descriptor, unaligned bit string with bounds	DSC$K_CLASS_UBSB

Refer to the *VMS Programming Support Manual* for more information about passing mechanisms.

6.2.5.1 VAX FORTRAN Built-In Functions

VAX FORTRAN also supports explicit argument-passing mechanisms, or *built-in* functions, that do not require formal data declarations. Specify built-in functions only in the argument list of the call (with one exception)[1] and use them to pass data to subroutines written languages other than VAX FORTRAN. The VAX FORTRAN built-in functions are:

- %VAL—Specifies that the argument must be passed as a value.
- %REF—Specifies that the argument must be passed as the address of the actual data.
- %DESCR—Specifies that the argument must be passed as the address of a descriptor that points to the actual data.
- %LOC—Returns the virtual address of the actual data.

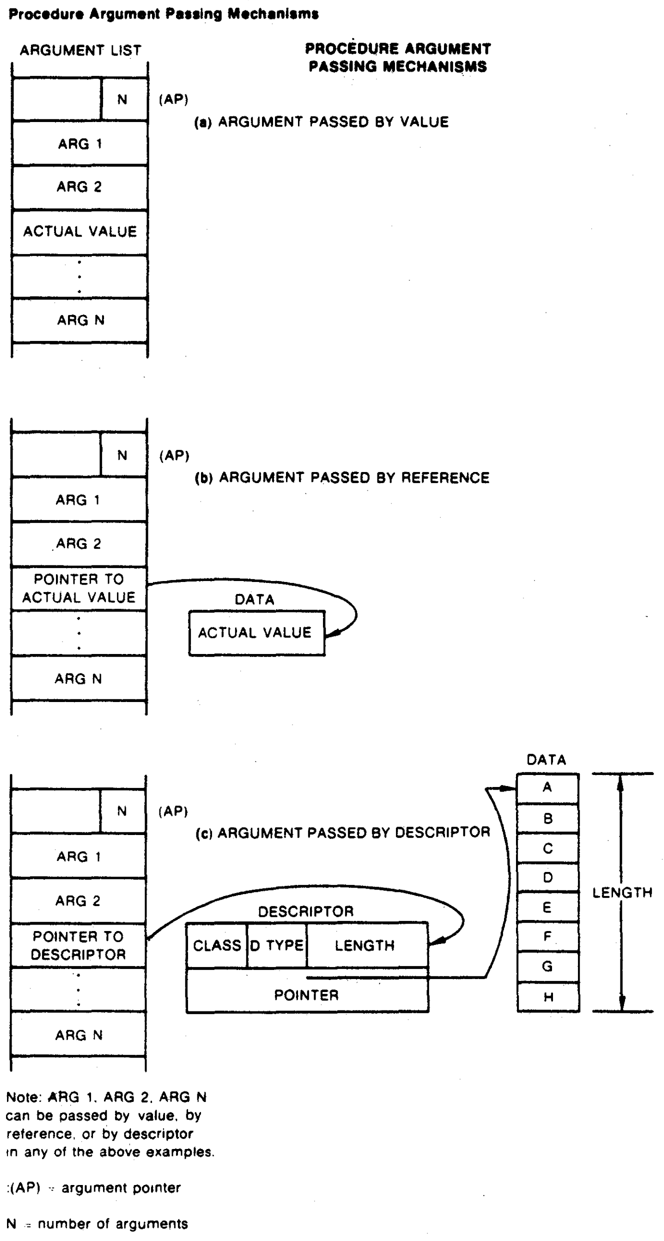
By default, VAX FORTRAN passes numeric data by reference and character string data by descriptor. The built-in functions override default argument-passing mechanisms. You might occasionally encounter an external procedure that passes data differently from the VAX FORTRAN default. In that case, use the built-in functions in VAX FORTRAN code.

For specific information about similar procedure argument-passing mechanisms for other high-level programming languages, refer to the specific language user's guide.

Figure 6-1 illustrates how arguments are placed on the stack and shows how arguments are passed to the called routine.

[1] You can use the built-in function %LOC outside an argument list to obtain the address of a variable. For example, use %LOC in an assignment statement where a longword in a character string descriptor is assigned the address of the actual character string

Figure 6–1 Passing Arguments


Procedure Argument Passing Mechanisms
ARGUMENT LIST
PROCEDURE ARGUMENT PASSING MECHANISMS
N
(AP)
(a) ARGUMENT PASSED BY VALUE
ARG 1
ARG 2
ACTUAL VALUE
ARG N
(b) ARGUMENT PASSED BY REFERENCE
POINTER TO ACTUAL VALUE
DATA
ACTUAL VALUE
(c) ARGUMENT PASSED BY DESCRIPTOR
POINTER TO DESCRIPTOR
DESCRIPTOR
CLASS
D TYPE
LENGTH
POINTER
A
B
C
D
E
F
G
H
Note: ARG 1, ARG 2, ARG N can be passed by value, by reference, or by descriptor in any of the above examples.
(AP) = argument pointer
N = number of arguments

6.3 UIS Constants

UIS constants are symbolic names for values that can be passed to, or returned from, UIS routines. UIS constants are syntactically equivalent to literal integer constants. Use them as follows:

- As arguments to UIS functions

- As indices into array arguments passed to, or received from, the UIS subsystem
- As literals to compare to a returned value from an inquiry routine

Refer to Section 6.5 for information about UIS symbol definition files.

6.4 Condition Values Signaled

Occasionally hardware- or software-related events occur, causing errors that could jeopardize successful program execution. Instead of returning condition values to R0 (as in VAX MACRO) or to a status variable (as in high-level languages), the UIS routines signal a condition. In such cases, unless you explicitly arrange to handle the signaled condition, program execution halts by setting up condition handlers.

6.5 Additional Program Components

In addition to the usual program entities, some UIS-specific and language-specific program components affect program execution.

Subroutines and Functions

If it uses a subroutine name as an argument to other subprograms, a VAX FORTRAN application program must use the EXTERNAL statement to declare the subroutine an external procedure. The subprogram can then use the corresponding dummy argument in a function reference or a CALL statement.

Entry Point and Symbol Definition Files

All UIS and UISDC routines are declared in an entry point file supplied with the graphics software. In addition, depending on the programming language, you might have to include a *data description* file of UIS symbol definitions. See the specific language user manual to determine whether you must include data description files in your program data declarations.

Table 6–2 contains a list of entry point files and symbol definition files for each VAX programming language. All files are in SYS$LIBRARY.

Table 6–2 Entry Point and Symbol Definition Files

VAX Language	Entry Point File	Symbol Definition File
BLISS	UISENTRY.R32	UISUSRDEF.R32
C	UISENTRY.H	UISUSRDEF.H
FORTRAN	UISENTRY.FOR	UISUSRDEF.FOR
MACRO		UISUSRDEF.MAR
PASCAL	UISENTRY.PAS	UISUSRDEF.PAS
PL/I	UISENTRY.PLI	UISUSRDEF.PLI
ADA	UISENTRY.ADA	UISUSRDEF.ADA

Message Definition File

A language-specific message definition file called UISMSG is included in the directory SYS$LIBRARY. This file, which is similar to the entry point file UISENTRY, defines all possible UIS error codes. For instance, to define message symbols in a VAX FORTRAN condition handler, you add the following line to your program.

```
INCLUDE 'SYS$LIBRARY:UISMSG'
```

Depending on the programming language options you choose, the appropriate version of UISMSG is copied to your disk during the installation procedure.

All messages symbols use the prefix UIS$_.

6.6 Notes to Programmers

The following sections describe language-specific issues that might affect program execution.

6.6.1 VAX ADA Programmers

Creating a Workable LIBRARY for VAX ADA To Use

Before you run VAX ADA application programs, you must perform the following procedures:

1 Set your default directory as follows:

```
SET DEFAULT SYS$LIBRARY
```

2 Request a directory of .ADA files.

```
DIRECTORY SYS$SYSROOT:[SYSLIB]*.ADA

UISENTRY.ADA;1  UISUSRDEF.ADA;1  UISMSG.ADA;1  VWSSYSDEF.ADA;1

Total of 4 files.
```

3 Copy the four files into one file as follows:

```
$COPY UISENTRY.ADA,UISUSRDEF.ADA,UISMSG.ADA,VWSSYSDEF.ADA UIS_.ADA
```

4 Edit the UIS_.ADA file.

```
$ EDIT UIS_.ADA
```

Insert the following four lines at the top of the file in the leftmost column:

```
with STARLET; use STARLET;
with SYSTEM; use SYSTEM;
with CONDITION_HANDLING; use CONDITION_HANDLING;
package UIS is
```

Place the body of the four entry-point files here.

Insert the last line in the UIS_.ADA file as follows:

```
end UIS;
```

5 To create a library that your VAX ADA programs can use, run the command file ADD$ADA_PREDEFINED_UNIT.COM as follows:

```
@ADD$ADA_PREDEFINED_UNIT.COM UIS_.ADA UIS
```

The compiled unit is placed in the library of predefined units for ADA in a file called UIS.ADA.

If you create the new library, it will be available to you automatically.

6 If you have not created the new library, use the following command to enter it into your own ADA library:

```
$ ACS ENTER UNIT ADA$PREDEFINED UIS
```

7 To use the UIS entry points in your program, add the following command to the beginning of your ADA program:

```
with UIS;
```

6.6.2 VAX C Programmers

Entry Point and Symbol Definition Files

The file UISENTRY.H defines all routine entry points in lowercase characters, and UISUSRDEF.H defines all constants in uppercase characters.

Floating-Point Constants

When you are programming UIS in C, it is recommended that you do not use floating-point constants in your C programs. UIS expects all values passed to it to be F_floating, or single precision. In VAX C, all floating-point constants are of type double (see *Programming in VAX C*, section 5.3.5).

6.6.3 VAX PASCAL Programmers

Entry Point Files

Because VAX PASCAL references arguments as formal parameters, your calls to UIS must specify the same parameter names as those in the entry point file UISENTRY.PAS. Therefore, specify **obj_id** as the argument whenever the routine descriptions in Parts III and IV allow a choice between the **obj_id** and **seg_id** arguments. Refer to Tables A-1 and B-1 for a summary of UIS and UISDC calling sequences.

Creating Environment Files

Before you run VAX PASCAL application programs, you must perform the following procedure.

1 Set your default directory as follows:

```
$ SET DEFAULT SYS$LIBRARY
```

2 Invoke the VAX PASCAL compiler with the /ENVIRONMENT and /NOOBJECT qualifiers to produce an environment file of symbolic definitions and type declarations.

NOTE: **In Version 3.4 of the VAX PASCAL compiler, a bug in a parameter declaration checking was fixed. This bug uncovered an invalid parameter declaration in the UISENTRY.PAS file shipped with VWS Version 3.0 and later. To maintain compatibility with all other versions of VMS Workstation Software and PASCAL, you must add the /NOWARNING qualifier when you build the PASCAL environment file.**

```
$ PASCAL/ENVIRONMENT/NOOBJECT/NOWARNING UISENTRY
```

The result of the compilation is UISENTRY.PEN, an environment file.

3 Include the INHERIT attribute in the first line of the application program or program module that specifies UISENTRY.PEN.

```
[INHERIT('UISENTRY.PEN')]
```

4 Repeat this procedure for the symbol definition file UISUSRDEF.PAS.

Refer to *Programming in VAX PASCAL* for more information about the /ENVIRONMENT and /NOOBJECT qualifiers and the INHERIT attribute.

Drawing Lines and Polygons

When you draw lines and polygons, use UIS$PLOT_ARRAY instead of UIS$PLOT and UIS$LINE_ARRAY instead of UIS$LINE.

6.6.4 VAX PL/I Programmers

Entry Point Files

Because VAX PL/I references arguments as formal parameters, your calls to UIS must specify the same parameter names as those in the entry point file UISENTRY.PLI. Therefore, specify **obj_id** as the argument whenever the routine descriptions in Parts III and IV allow a choice between the **obj_id** and **seg_id** arguments. Refer to Tables A-1 and B-1 for a summary of UIS and UISDC calling sequences.

6.7 Programming Examples

The programming examples in Parts II and III of this manual use VAX FORTRAN Version 4.4. In addition, some examples—particularly in Part III—include ellipses to indicate omitted portions of code and to point out places in the program where you can add code.

Many of the examples include the VAX FORTRAN PAUSE statement. PAUSE suspends program execution and displays the DCL prompt ($). A default message—FORTRAN PAUSE—is returned to the display screen. The graphic images remain on the screen. Respond to the DCL prompt ($) by typing one of the following commands:

- CONTINUE—Resume program execution at the next executable statement.
- EXIT—Terminate program execution.

- DEBUG—Resume program execution under the control of the VAX/VMS Symbolic Debugger.

NOTE: **If your program is running in batch mode, program execution is not suspended. All messages are written to the system output file.**

6.7.1 Structure of Programming Tutorial

Each chapter in Part II uses a tutorial approach to explain UIS graphics features and programming. After discussion of the main topics, each chapter includes:

- Programming options—Lists available features. The addition of each new group of programming options lets you progress from simple to complex programming tasks.
- Program development—Lists current programming objective and tasks needed to implement the objective successfully.
  - Program—Contains the source module with embedded callouts. Each callout refers to a programming feature.
  - Program output—Displays and explains the output from the program.

Each programming example uses some or all of the programming options listed. Not all routines are illustrated in the accompanying example.

6.8 Program Execution

Your program can run in batch mode with predefined data or interactively, accepting input as needed. However, to execute your application program successfully, you must first store it as a file using a text editor.

Invoke the text editor on your workstation as follows:

```
$ EDIT MYPROG.FOR
```

Please note that in this example you must supply a file name, for example, MYPROG. In addition, a VAX FORTRAN file type (FOR) is added to the file name to identify the file as a VAX FORTRAN source file. Enter your program according to the rules of your programming language. Refer to the specific language reference manual for detailed information.

6.8.1 Compiling Your Program

You must compile the newly created source file MYPROG.FOR before execution. The language compiler (in this case the VAX FORTRAN compiler) checks for proper syntax and initiates code optimization where appropriate. Invoke the language compiler as follows:

```
$ FORTRAN/LIST MYPROG
```

You need not include file type. By default, the system searches for the latest version of the file, MYPROG, with a file type of FOR. If the application source file contains syntax errors, you receive *compile-time* error messages called *diagnostics*. These diagnostic messages indicate the portion of code in error as well as an explanation. The /LIST qualifier specifies the creation of a listing file of accounting information and diagnostics (if present).

Some language compilers return a predetermined maximum number of diagnostics before terminating compilation. You must correct these errors and resubmit the source program for a successful compilation. Successful compilation produces an object module with file type of OBJ.

6.8.2 Linking the Object Module

The Linker resolves references to subroutines and allocates memory to variables within your program. Invoke the Linker as follows:

```
$ LINK MYPROG
```

You need not specify the file type of the program, MYPROG. By default, the system searches for the latest version of the file MYPROG with the file type OBJ.

In addition, you can link object modules of programs written in different source code.

6.8.3 Running the Executable Image

The Linker produces an executable image with a file type of EXE. At this point, you can run your program. However, if you receive run-time errors, you must correct the errors in your source code, recompile the source module, and relink the object modules. After you receive the $ prompt, run the executable image as follows:

```
$ RUN MYPROG
```

7 Creating Basic Graphic Objects

7.1 Overview

This chapter describes how to create basic graphic objects — lines, circles, ellipses, and text. It discusses the following topics:

- Creating a virtual display
- Creating graphics and text
- Creating a display window

You construct an interactive program to create graphic objects. You use other windowing routines to manipulate these objects.

Refer to Section 6.7 for more information about the programming examples in this manual.

7.2 Step 1—Creating a Virtual Display

When you use UIS to create graphic objects, you use a frame of reference called a *virtual display* to establish the environment in which the graphic objects exist.

Calls to UIS routines must reference points within the virtual display. When you specify coordinates, the UIS subsystem generates a coordinate system to create the virtual display and subsequent windows. You use this coordinate system, or grid, to reference points as *world coordinates* along two perpendicular axes labelled x and y. The virtual display is infinite and you can draw graphic objects anywhere in it.

7.2.1 Specifying Coordinate Values

Many routines documented in this manual require coordinates to define virtual displays, display windows, and extent rectangles. Table 7–1 lists information about coordinate values.

Table 7–1 Coordinate Types and Values

Coordinate	Units	Data Type	Origin
Absolute	cm	F_floating[1]	Lower-left corner of display screen or tablet
Normalized	Gutenbergs	F_floating[1]	Lower-left corner of virtual display
Viewport-relative	Pixels	Longword (unsigned)	Lower-left corner of display viewport
World	User-specified	F_floating[1]	Lower-left corner of virtual display

[1]F_floating point numbers can have up to approximately seven decimal digits of precision.

7.2.2 Creating and Deleting a Virtual Display

You use UIS$CREATE_DISPLAY to specify the world coordinate space in which you will draw graphic objects. The world coordinate values you specify establish mapping and scaling factors that the system can use later in viewport creation. Do not think of the coordinate values as the absolute boundaries of the virtual display.

You can create an unlimited number of virtual displays, subject to system and process resources.

You can use UIS$DELETE_DISPLAY anywhere in your program to delete a virtual display. However, you should remember that when you delete a virtual display you are throwing out the medium on which you have drawn graphic objects.

7.2.3 Program Development

Programming Objective

To create an executable program using the VAX FORTRAN programming language.

Programming Tasks

To create and delete a virtual display.

```
PROGRAM IMAGES_1
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY' ❶
INCLUDE 'SYS$LIBRARY:UISUSRDEF' ❷
   .
   .
   .
VD_ID=UIS$CREATE_DISPLAY(+1.0,+1.0,+20.0,+20.0,10.0,10.0) ❸
   .
   .
   .
PAUSE ❹
CALL UIS$DELETE_DISPLAY(VD_ID) ❺
END
```

At this point the program contains UIS entry points ❶ and definitions ❷. It also includes a call ❸ to UIS$CREATE_DISPLAY. The plus sign (+) is optional for positive coordinates. The minus sign (-) is required for negative coordinates.

Because world coordinates are f_floating numbers, you must use a decimal point when you specify world coordinate pairs.

See Section 6.7 for information about the VAX FORTRAN PAUSE statement ❹.

Call UIS$DELETE_DISPLAY ❺ to remove the virtual display before the program ends. You need not use UIS$DELETE_DISPLAY to terminate an application program.

Not only does UIS$CREATE_DISPLAY specify the world coordinate range of the virtual display, but also, in **vd_id**, it returns the value of the virtual display identifier. This value, used in subsequent windowing routines, uniquely identifies the newly created virtual display. Typically, UIS$CREATE_DISPLAY is the first UIS routine called in an application program.

If your application program invokes the UIS$CREATE_DISPLAY only, the workstation screen does not change.

7.3 Step 2—Creating Graphics and Text

You can now draw any of the following graphic objects anywhere on the virtual display.

Graphic Object	Example
Geometric shapes	Point, line, polygon, circle, and ellipse
Text	Characters
Raster images	Any object constructed with a bitmap of varying size

7.3.1 Graphics Drawing Operation Specifications

- All line drawing operations are symmetrical and include both end points.
- In the case of fill patterns, images, ellipses, moving windows, and so forth, all region specifications include the region borders.

7.3.2 Programming Options

Creating Points, Lines, and Polygons

Depending on the number of times you repeat coordinate pairs in UIS$PLOT or UIS$PLOT_ARRAY, you can draw a point, connected lines, or a polygon.

You can draw more than one unconnected line in a single call to UIS$LINE or UIS$LINE_ARRAY. Each specified *pair* of world coordinate pairs represents the end points of a line.

NOTE: **VAX PASCAL application programs should use UIS$PLOT_ARRAY or UIS$LINE_ARRAY to draw all lines, disconnected lines, and polygons.**

Creating Circles

Use UIS$CIRCLE to create circles or circular arcs.

Creating Ellipses

Use UIS$ELLIPSE to create ellipses or elliptical arcs.

Drawing Images

Use the following procedure to create a bitmap image of a graphic object, then draw the raster to the display screen with UIS$IMAGE.

1. Create a data structure such as an array or record in your program to define the bitmap.
2. Set the bits in the structure to create the bitmap image by assigning values to the elements of the structure.
3. Use UIS$IMAGE to specify pixel width and height of the raster image.
4. Use UIS$IMAGE to specify the name of the data structure.

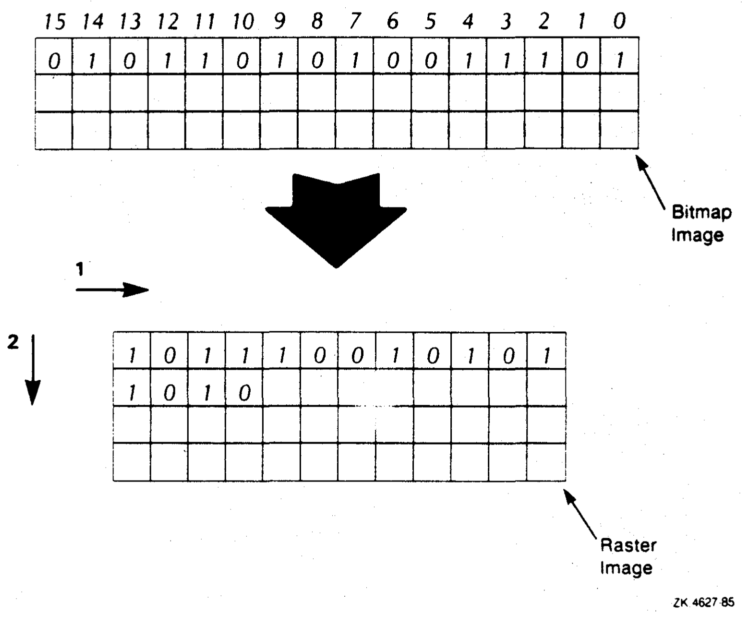
Figure 7-1 illustrates how bitmap settings are mapped to raster images.

Raster image mapping occurs from left to right and from top to bottom. See the UIS$IMAGE routine description for more information.

Text

Use UIS$TEXT to set the current position and create text anywhere within a virtual display. You can use the text within a virtual display to label an accompanying graphic object within the window. Only UIS$TEXT can write characters in a virtual display.

Figure 7-1 Mapping a Bitmap to a Raster



7.3.3 Program Development

Programming Objective

To create an executable program using the VAX FORTRAN programming language.

Programming Tasks

1 Create a virtual display.

2 Draw four graphic objects in the virtual display.

3 Delete the virtual display.

```
PROGRAM IMAGES_2
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
REAL WIDTH,HEIGHT

VD_ID=UIS$CREATE_DISPLAY(1.0,1.0,20.0,20.0,10.0,10.0)

CALL UIS$CIRCLE(VD_ID,0,10.0,10.0,1.0) 1
CALL UIS$PLOT(VD_ID,0,4.0,3.0,5.0,7.0) 2
CALL UIS$ELLIPSE(VD_ID,0,15.0,15.0,1.0,2.0) 3
CALL UIS$TEXT(VD_ID,0,'This is a test.',1.0,12.0) 4
   .
 .
 .
PAUSE
CALL UIS$DELETE_DISPLAY(VD_ID)
END
```

In the preceding example, you specify world coordinate pairs that describe the exact locations of the graphic objects (circle, line, ellipse, and text) in the virtual display, explicitly to the UIS graphics routines ❶ ❷ ❸ ❹.

If you execute the program in its present form, the workstation display screen shows no objects. Although your calls to the UIS graphics and text routines are processed, you must create a window to view what is drawn.

7.4 Step 3—Creating a Display Window

You must now create a display window to define the world coordinate range of the viewable portion of the virtual display. When you create a display window, you also create a display viewport, an area on the physical screen where the display window is mapped.

7.4.1 Programming Options

At this point, all the available programming options are provided through UIS$CREATE_WINDOW. The full capabilities of UIS$CREATE_WINDOW are discussed in more detail in Chapter 8.

Creating a Display Window and Viewport

Use UIS$CREATE_WINDOW to create a display viewport and its associated viewport.

7.4.2 Program Development

Programming Objective

To create an executable program that draws and displays graphic objects on the VAXstation screen.

Programming Tasks

1 Create a virtual display.

2 Draw four graphic objects in the virtual display.

3 Create a display window and viewport.

4 Delete the virtual display.

```
PROGRAM IMAGES_2A
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
REAL*4 WIDTH,HEIGHT

TYPE *,'ENTER DESIRED VIEWPORT WIDTH AND HEIGHT'
ACCEPT *,WIDTH,HEIGHT

VD_ID=UIS$CREATE_DISPLAY(1.0,1.0,20.0,20.0,WIDTH,HEIGHT)  ❶

CALL UIS$CIRCLE(VD_ID,0,10.0,10.0,1.0)  ❷
CALL UIS$PLOT(VD_ID,0,4.0,3.0,5.0,7.0)  ❸
CALL UIS$ELLIPSE(VD_ID,0,15.0,15.0,1.0,2.0)  ❹
CALL UIS$TEXT(VD_ID,0,'This is a test.',1.0,12.0)  ❺

WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION') ❻
```

```
PAUSE
CALL UIS$DELETE_DISPLAY(VD_ID)
END
```

Specify the world coordinate range of the virtual display and the default dimensions of the display viewport in a call to UIS$CREATE_DISPLAY ❶.

NOTE: The display viewport is not mapped until a display window is created.

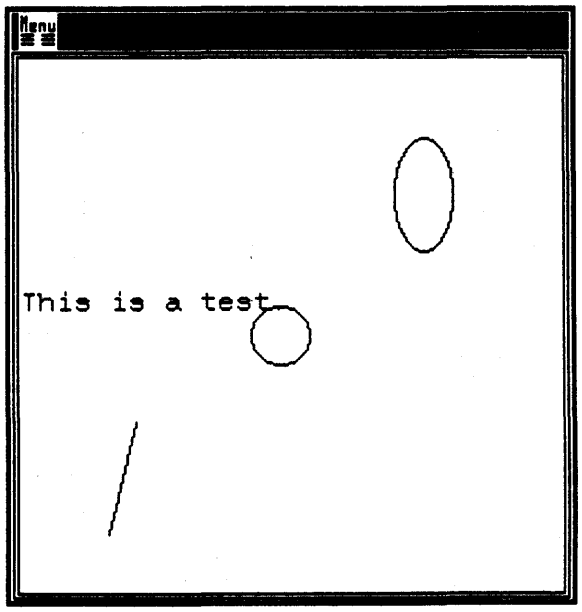
Next, call the graphics and text routines ❷ ❸ ❹ ❺ to draw the graphic objects.

Create a display window and viewport in a call to UIS$CREATE_WINDOW ❻. The world coordinate range of the window and the viewport width and height are not specified. Therefore, the world coordinate space of the display window (that is, the viewable portion of the virtual display) defaults to the entire virtual display. You see all objects drawn in the virtual display.

7.4.3 Calling UIS$CIRCLE, UIS$ELLIPSE, UIS$PLOT, UIS$TEXT, and UIS$CREATE_WINDOW

When you run the program IMAGES_2A, you should get a single, untitled display viewport containing text, a circle, a line, and an ellipse as shown in Figure 7-2.

Figure 7-2 Display Viewport and Graphic Objects



8 Display Windows and Viewports

8.1 Overview

Before you manipulate graphic objects, you should know about display windows and viewports. These features allow you to see graphic objects drawn in the virtual display. The UIS *windowing* routines perform the following operations:

- Create display windows and viewports
- Move display windows
- Manipulate display viewports
- Delete display windows
- Erase the virtual display
- Create transformations

8.2 Windowing Routines

You use windowing routines to create and delete virtual displays, display windows, and display viewports. Table 8-1 lists windowing routines and their functions.

Table 8-1 UIS Windowing Routines

Routine	Description
UIS$CREATE_DISPLAY	Creates a virtual display and defines default viewport dimensions
UIS$CREATE_WINDOW	Creates display window and viewport
UIS$EXPAND_ICON	Substitutes an associated viewport for an icon
UIS$MOVE_AREA	Moves a specified rectangle and its contents in the virtual display to another part of the virtual display
UIS$MOVE_WINDOW	Pans the display window across the virtual display
UIS$POP_VIEWPORT	Allows an occluded viewport to be fully displayed
UIS$PUSH_VIEWPORT	Places a viewport behind another viewport
UIS$SHRINK_TO_ICON	Substitutes an icon for a display viewport
UIS$CREATE_ TRANSFORMATION	Alters the world coordinate space of the virtual display
UIS$ERASE	Erases objects that lie completely within a specified rectangle in the virtual display

Table 8-1 (Cont.) UIS Windowing Routines

Routine	Description
UIS$DELETE_DISPLAY	Deletes a virtual display
UIS$DELETE_WINDOW	Deletes a display window and viewport

These routines allow you to create and manage the display screen environment and to perform certain housekeeping functions such as erasing and deleting virtual displays and windows.

8.3 Step 1—Creating Many Display Windows

For every display window you create, you also create a display viewport. A one-to-one relationship exists between each display window and its associated viewport. An application program can create an unlimited number of display windows and viewports, subject to system and process resources.

8.3.1 Programming Options

Each display window can be unique with regard to world coordinate range. Therefore, you can create display viewports that are also unique with respect to dimensions and position on the display screen.

Display Window Size

By default, a newly created display window displays the full world coordinate space specified when you create the virtual display. You can specify world coordinate pairs in UIS$CREATE_WINDOW to produce different size display windows within the virtual display.

Display Viewport Size

Similarly, the default display viewport dimensions equal the values you specify in the **width** and **height** arguments in the UIS$CREATE_ DISPLAY call. However, you can specify different dimensions to scale the contents of the window. Maximum display viewport size depends on the dimensions of the display screen. If you specify viewport dimensions that exceed the size of the display screen, UIS scales the viewport to the size of the display screen.

Graphic Object Magnification

You can manipulate the world coordinate range of the display window or the dimensions of the display viewport to increase or decrease magnification of the object in the viewport. Magnification occurs when the display window area is increased or decreased while the viewport size remains the same, or when the viewport is increased or decreased while dimensions of the window remain the same.

Distortion

Distortion occurs whenever the aspect ratios of the display viewport and display window are not equal.

The aspect ratio of the display window is the absolute value of the difference between *y* world coordinates of the upper-right and the lower-right corners of the window divided by the absolute value of the difference between the *x* world coordinates of the lower-right and lower-left corners. Figure 8–1 illustrates how to calculate the aspect ratios of the display window and viewport.

Figure 8–1 Aspect Ratios of the Display Window and Display Viewport

$$\frac{|y1 - y0|}{|x1 - x0|} = \frac{\text{viewport height}}{\text{viewport width}}$$

ZK-4579-85

Number of Windows and Viewports

You can create an unlimited number of display windows and, as a result, an unlimited number of display viewports, subject to system and process resources. In addition, you can specify the dimensions of each display viewport.

Display Banner

The display banner appears along the top border of the display viewport and contains the menu and keyboard icons as well as the viewport title. The maximum length of the viewport title is 63 characters.

You can suppress display banner generation with the **attributes** argument in UIS$CREATE_WINDOW. When the display banner is suppressed, only the viewport border displays.

Display Viewport Placement

You can either explicitly place a display viewport on the workstation screen or allow UIS to choose a location for you. By default, display viewport placement is random.

8.3.2 Program Development

Programming Objective

To create four display windows and display viewports.

Programming Tasks

1 Create a virtual display.

2 Draw four graphic objects in the virtual display.

3 Create four display windows and viewports, omitting the display window coordinates in the calls to UIS$CREATE_WINDOW.

4 Delete the virtual display.

```
PROGRAM IMAGES_3
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'

VD_ID=UIS$CREATE_DISPLAY(1.0,1.0,20.0,20.0,10.0,10.0)

CALL UIS$CIRCLE(VD_ID,0,10.0,10.0,1.0)
CALL UIS$PLOT(VD_ID,0,4.0,3.0,5.0,7.0)
CALL UIS$ELLIPSE(VD_ID,0,15.0,15.0,1.0,2.0)
CALL UIS$TEXT(VD_ID,0,'This is a test.',1.0,12.0)
WD_ID1=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION') ❶
PAUSE
WD_ID2=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION') ❷
WD_ID3=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION') ❸
WD_ID4=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION') ❹

PAUSE
CALL UIS$DELETE_DISPLAY(VD_ID)

END
```

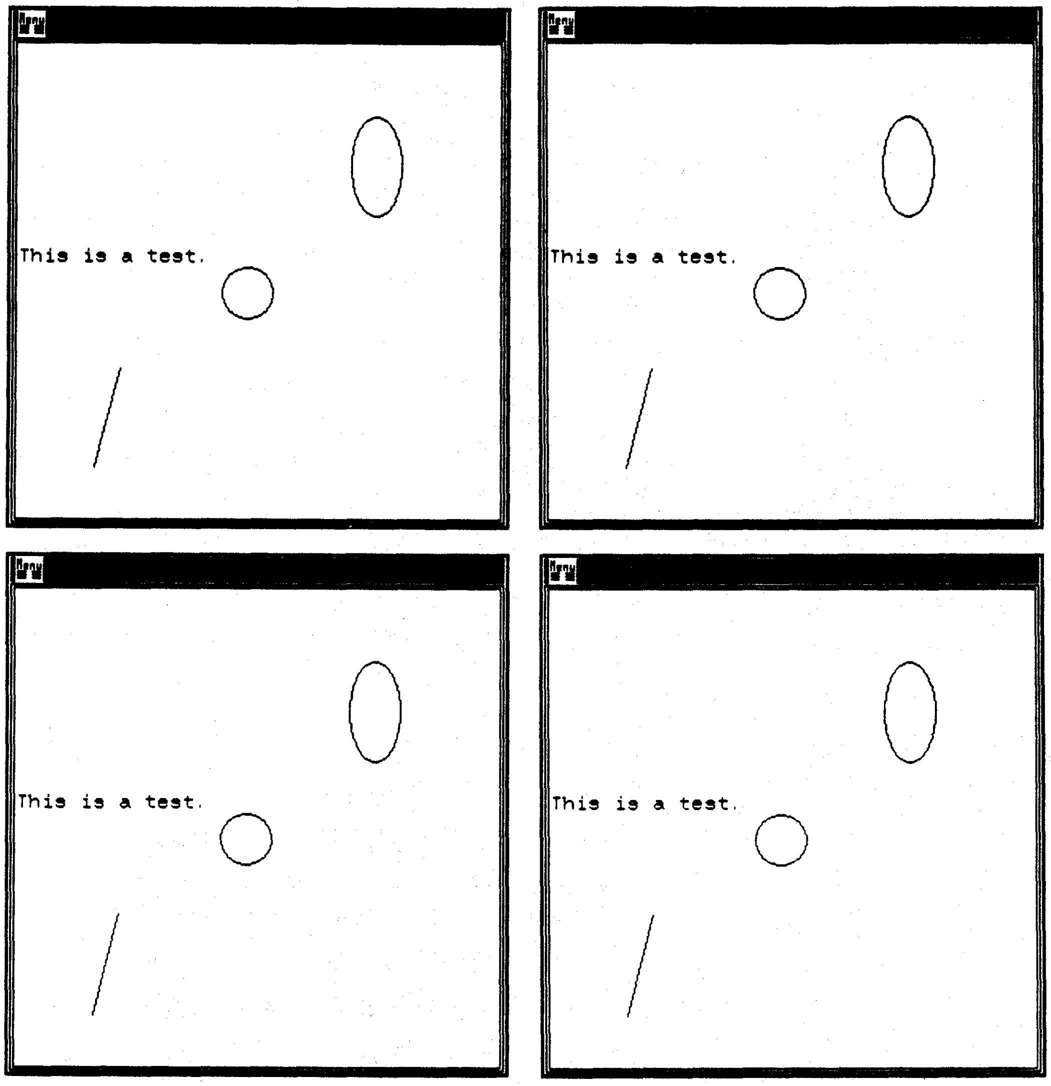
Four calls to UIS$CREATE_WINDOW ❶ ❷ ❸ ❹ are inserted to create four windows. The world coordinate range of each window defaults to the world coordinate range of the entire virtual display.

8.3.3 Calling UIS$CREATE_WINDOW

If you run this program now, your workstation screen displays the graphic objects shown in Figure 8–2.

As you can see, four display windows are created and mapped to the display screen as four viewports. Each of the viewports contains four objects. Because display window world coordinate pairs were not explicitly specified in UIS$CREATE_WINDOW, the viewports allow you to see the entire area of the virtual display by default. In addition, because the display viewport width and height were not explicitly specified in the UIS$CREATE_WINDOW call, each display viewport is, by default, 10 cm square as specified in the **width** and **height** arguments of the UIS$CREATE_DISPLAY call.

Figure 8–2 Four Display Viewports



8.4 Step 2—Deleting and Erasing Display Windows

Some windowing routines perform housekeeping functions; that is, they delete unused display windows or erase graphic objects from the virtual displays. When you run complicated applications, such routines are important to manage display environment.

8.4.1 Programming Options

You can call certain UIS routines that cause your application program to delete unwanted windows, viewports, and virtual displays.

Display Window Deletion

You can delete any display window without affecting other windows or viewports. Deletion of the display window does not affect the graphic objects in the virtual display. If you delete a display window, you also delete the associated display viewport. To delete a display window and its associated viewport, specify the appropriate display window identifier in UIS$DELETE_WINDOW.

Erasing the Virtual Display

Use UIS$ERASE at any time to delete graphic objects that lie completely within a specified rectangle in the virtual display. If you do not specify a rectangle, the entire virtual display is used.

8.4.2 Program Development

Programming Objectives

- To enclose each graphic object in its own display window.
- To delete a window and its viewport.

Programming Tasks

1 Create a virtual display.

2 Draw four graphic objects in the virtual display.

3 Create four display windows and viewports that specify display window regions to enclose each of the graphic objects.

  - Specify display window regions that enclose the graphic objects.
  - Specify viewport titles that identify the graphic objects.

4 Delete one of the display windows and its viewport.

```
PROGRAM IMAGES_4
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
REAL WIDTH,HEIGHT
TYPE *,'ENTER DISPLAY SIZE' 1
ACCEPT *,WIDTH,HEIGHT

VD_ID=UIS$CREATE_DISPLAY(1.0,1.0,20.0,20.0,WIDTH,HEIGHT)

CALL UIS$CIRCLE(VD_ID,0,12.0,12.0,1.0)
CALL UIS$PLOT(VD_ID,0,4.0,3.0,5.0,7.0)
CALL UIS$ELLIPSE(VD_ID,0,15.0,15.0,1.0,2.0)
CALL UIS$TEXT(VD_ID,0,'This is a test.',1.0,12.0)
```

```
     WD_ID1=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','CIRCLE',
    2        10.0,10.0,14.0,14.0,WIDTH,HEIGHT) ❷
     WD_ID2=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','LINE',
    2        3.0,2.0,6.0,8.0,WIDTH,HEIGHT) ❸
     WD_ID3=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','TEXT',
    2        1.0,12.0,10.0,10.0,WIDTH,HEIGHT) ❹
     WD_ID4=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','ELLIPSE',
    2        13.0,13.0,17.0,18.0,WIDTH,HEIGHT) ❺

     PAUSE
         CALL UIS$DELETE_WINDOW(WD_ID2)  ❻

     PAUSE

     END
```

The program now accepts interactive input for the display viewport dimensions. ❶

To define each display window in the UIS$CREATE_WINDOW calls ❷ ❸ ❹ ❺, you explicitly specify world coordinate space.

UIS$CREATE_WINDOW returns the variable *wd_id2*, the display window identifier ❸, to identify the LINE window uniquely. Note that the call to delete the LINE window ❻ references this variable.

8.4.3 Calling UIS$DELETE_WINDOW

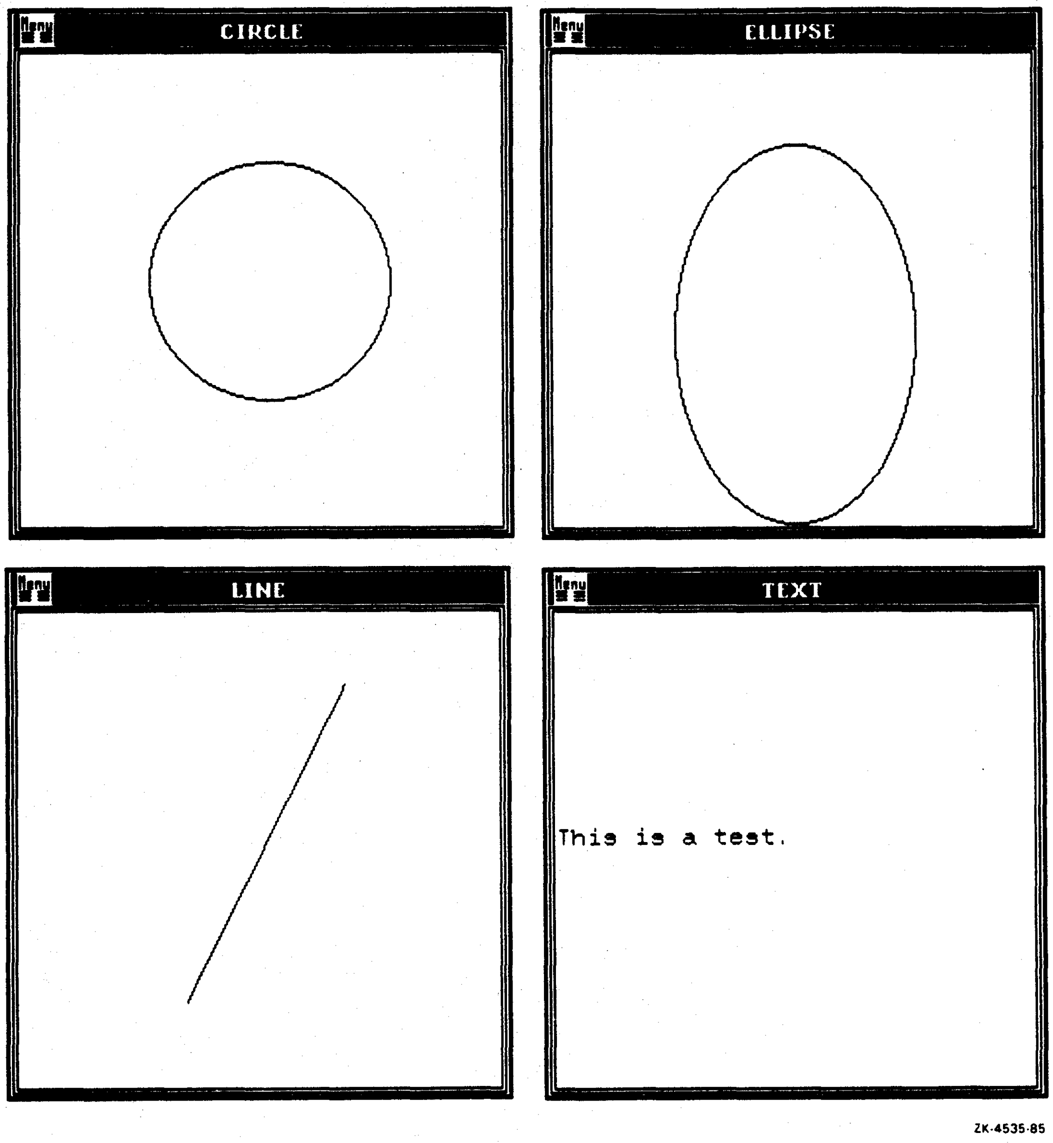
If you run this program until the first PAUSE statement, the workstation screen displays the graphic objects shown in Figure 8-3.

When you explicitly specify a particular world coordinate range within the virtual display for each display window, each graphic object lies within a separate window that maps to the physical display screen as a separate display viewport.

To continue program execution, type CONTINUE at the DCL prompt ($). The program continues to execute and the screen changes, as shown in Figure 8-4.

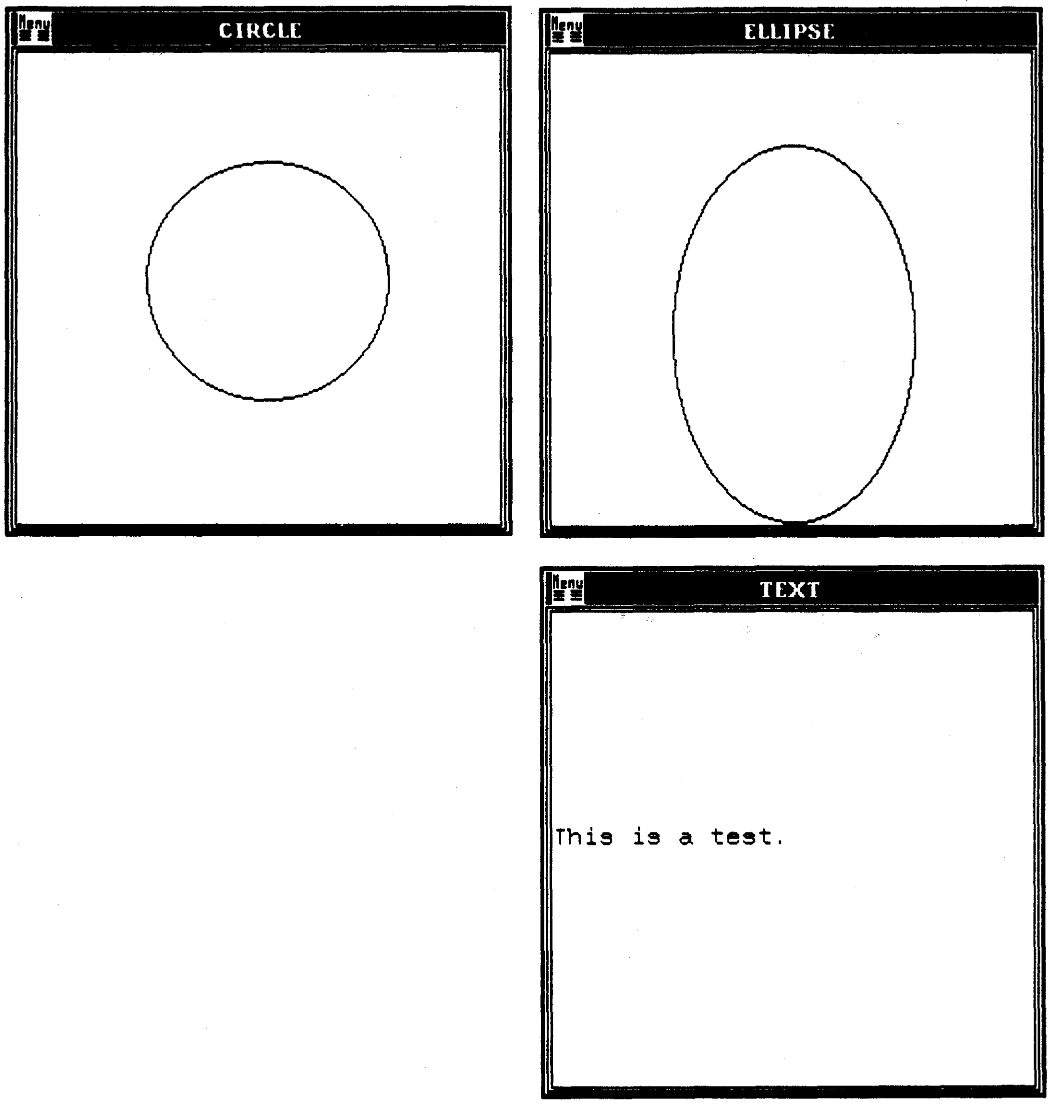
Although the viewport LINE and its window are deleted, the actual graphic object still exists. You have simply deleted the display window that allowed you to view the line portion of the virtual display. If you call UIS$CREATE_WINDOW again and specify the appropriate world coordinate space in the virtual display, the object reappears.

Figure 8–3 Objects Within Different Windows



This is a test.

Figure 8–4 Display Window Deletion



8.5 Step 3—Manipulating Display Windows and Viewports

Display viewports and windows do not have to remain as static objects on the screen. You can manipulate the newly created display windows and viewports in many ways.

8.5.1 Programming Options

Use the optional **attributes** argument of UIS$CREATE_WINDOW to implement viewport placement features and window attributes.

NOTE: **When you include the attributes argument in UIS$CREATE_WINDOW, you do not modify attribute block 0.**

Attributes and attribute block 0 are discussed in detail in Chapter 9.

General and Exact Placement of Viewports

Unless you specify otherwise, display viewports are placed randomly throughout the screen. You can move a display viewport to any position on the screen. When you create the window, you can specify general viewport placement, that is, within a certain vicinity on the screen—top, left, right, or bottom.

If you specify exact placement, the display viewport is positioned anywhere you want it on the screen. This placement saves space by allowing occlusion of other viewports.

Panning and Zooming the Virtual Display

You can pan across the virtual display to include either the entire virtual display or any discrete area within it.

Pushing and Popping Display Viewports

Pushing and popping display viewports is useful when you create display windows with the exact placement attribute. In this case, your application might create two windows and purposely occlude one of the viewports. Since you know which viewport is occluded, you can use UIS$POP_VIEWPORT.

Otherwise, by default, the UIS subsystem places newly created windows randomly on the screen. As a result, since you do not know where the viewports will be placed, you should not use UIS$POP_VIEWPORT or UIS$PUSH_VIEWPORT.

Moving a Display Viewport

You can use UIS$MOVE_VIEWPORT to move an existing display viewport anywhere on the screen.

Moving a Portion of the Virtual Display

Use UIS$MOVE_AREA to draw a graphic object in a portion of the virtual display, then move that coordinate space to another part of the same virtual display.

8.5.2 Program Development I

Programming Objectives

To delete three display windows and viewports, then use the remaining display window to pan the virtual display.

Programming Tasks

1 Create a virtual display.

2 Draw four graphic objects in the virtual display.

3 Create four display windows and viewports, each containing a graphic object.

4 Specify a title for each viewport.

5 Delete three of the four display windows.

6 Use UIS$MOVE_WINDOW to pan the virtual display with the remaining display window.

```
      PROGRAM IMAGES_5
      IMPLICIT INTEGER(A-Z)
      INCLUDE 'SYS$LIBRARY:UISENTRY'
      INCLUDE 'SYS$LIBRARY:UISUSRDEF'
      REAL WIDTH,HEIGHT

      TYPE *,'ENTER VIEWPORT WIDTH AND HEIGHT'
      ACCEPT *,WIDTH,HEIGHT

      VD_ID=UIS$CREATE_DISPLAY(1.0,1.0,20.0,20.0,10.0,10.0)

      CALL UIS$CIRCLE(VD_ID,0,12.0,12.0,1.0) ❶
      CALL UIS$PLOT(VD_ID,0,4.0,3.0,5.0,7.0)  ❷
      CALL UIS$ELLIPSE(VD_ID,0,15.0,15.0,1.0,2.0) ❸
      CALL UIS$TEXT(VD_ID,0,'This is a test.',1.0,12.0)  ❹
      WD_ID1=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','CIRCLE',
     2        10.0,10.0,14.0,14.0,WIDTH,HEIGHT) ❺
      WD_ID2=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','LINE',
     2        3.0,2.0,6.0,8.0,WIDTH,HEIGHT)  ❻
      WD_ID3=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','TEXT',
     2        1.0,12.0,10.0,10.0,WIDTH,HEIGHT) ❼
      WD_ID4=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','ELLIPSE',
     2        13.0,13.0,17.0,18.0,WIDTH,HEIGHT)  ❽

      PAUSE  ❾
      CALL UIS$DELETE_WINDOW(WD_ID1)  ❿
      CALL UIS$DELETE_WINDOW(WD_ID3)  ⓫
      CALL UIS$DELETE_WINDOW(WD_ID4)  ⓬

      PAUSE    ⓭

      CALL UIS$MOVE_WINDOW(VD_ID,WD_ID2,6.0,8.0,18.0,18.0)  ⓮

      PAUSE  ⓯

      CALL UIS$DELETE_DISPLAY(VD_ID)

      END
```

The program IMAGE_5 creates four graphic objects ❶ ❷ ❸ ❹ in the virtual display.

The program prompts for the viewport width and height to override the values specified in UIS$CREATE_DISPLAY.

Each newly created display window ❺ ❻ ❼ ❽ contains a graphic object. Each display window is mapped to the physical screen as a display viewport with an appropriate title that describes the graphic object within the window.

Program execution is suspended ❾. The display screen contains the four viewports previously described.

Three calls to UIS$DELETE_WINDOW ❿ ⓫ ⓬ remove the windows and their viewports CIRCLE, ELLIPSE, and TEXT from the display screen.

Program is suspended ⓭. The display screen contains one display viewport LINE.

A call to UIS$MOVE_WINDOW ⓮ has been inserted. Thus, the display window LINE pans the virtual display.

8.5.3 Calling UIS$MOVE_WINDOW

The display screen initially contains all four windows as shown in Figure 8-5.

Three of the display windows and viewports are deleted.

The display viewport LINE remains. Originally, the viewport contained a line; now it contains the circle and the ellipse. The display window goes to the location you specify in the virtual display. You can include any number of calls to UIS$MOVE_WINDOW. Your workstation screen displays the objects shown in Figure 8-6. The circle and the ellipse still exist in the virtual display.

Figure 8–5 Before Panning the Virtual Display

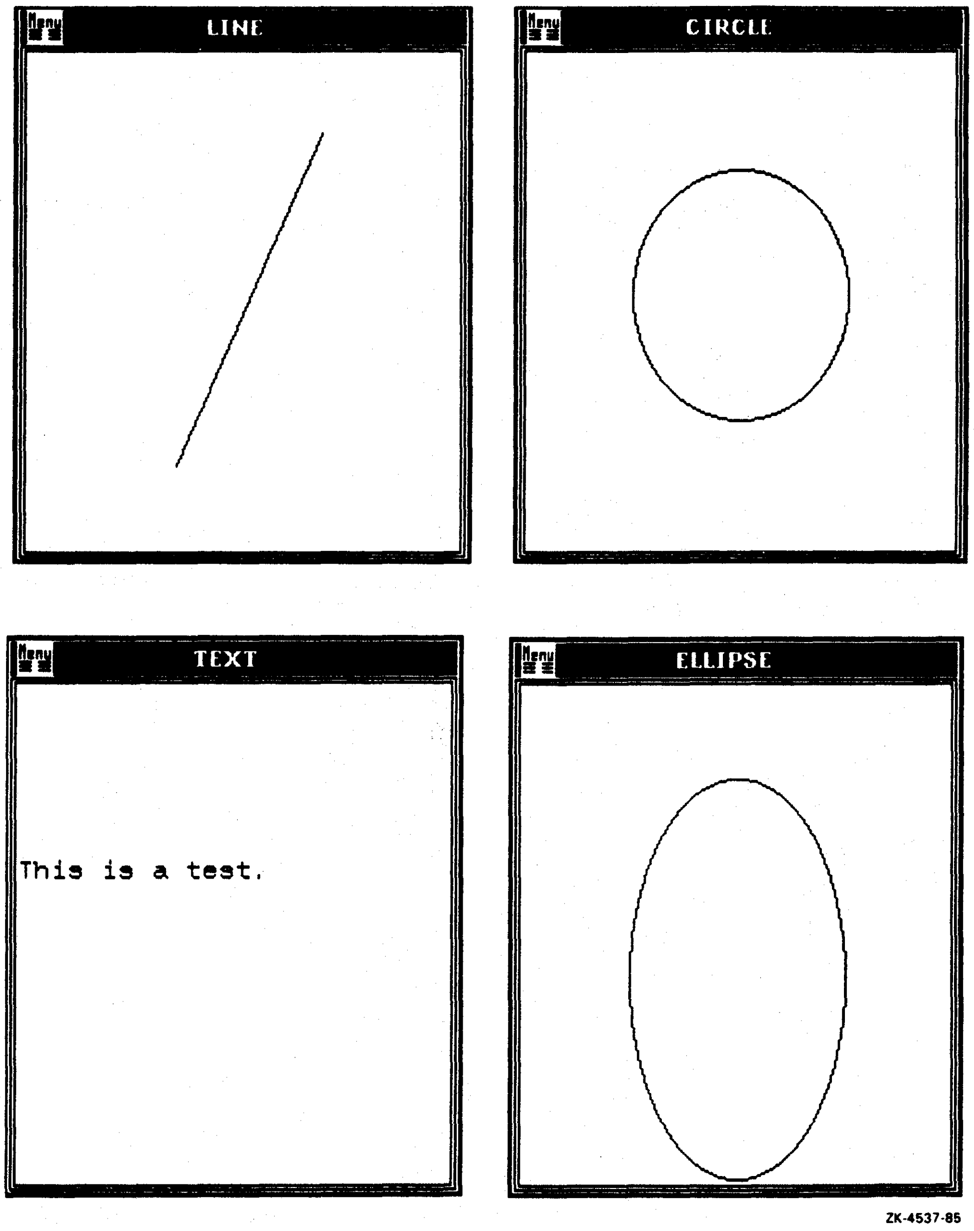
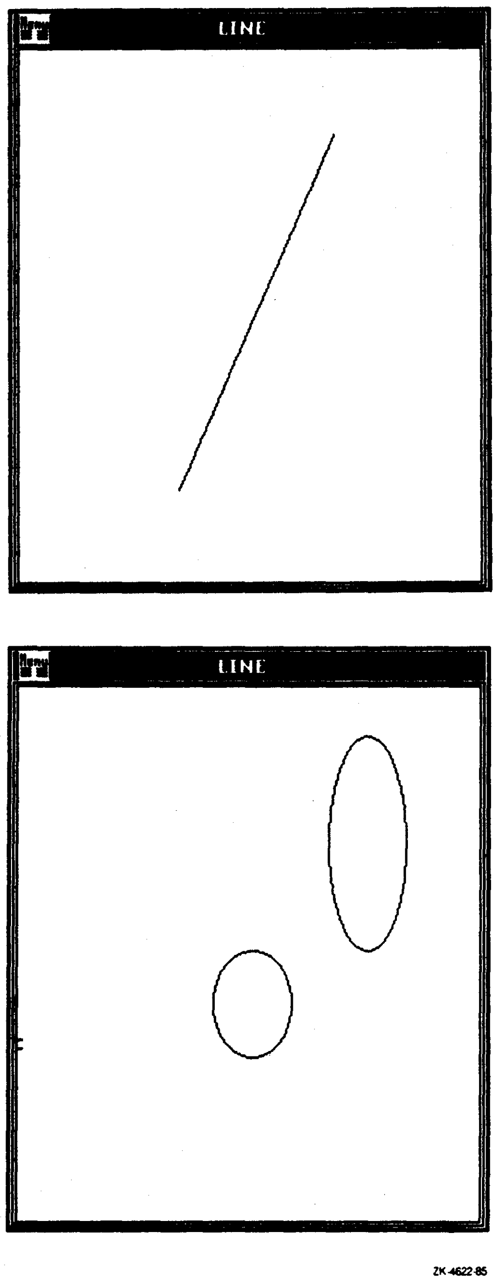


Figure 8–6 Panning the Virtual Display



8.5.4 Program Development II

Programming Objective

To demonstrate exact placement of the display viewport on the display screen to pop and push viewports.

Programming Tasks

1 Specify viewport placement data to create a viewport attributes data structure.

2 Create a virtual display.

3 Draw two graphic objects in the virtual display in separate viewports.

4 One viewport initially occludes the other.

```
      PROGRAM IMAGES_6
      IMPLICIT INTEGER(A-Z)
      INCLUDE 'SYS$LIBRARY:UISENTRY'
      INCLUDE 'SYS$LIBRARY:UISUSRDEF'
      REAL WIDTH,HEIGHT
      STRUCTURE/PLACE/ ❶
        INTEGER*4   CODE_1
        REAL*4      ABS_POS_X
        INTEGER*4   CODE_2
        REAL*4      ABS_POS_Y
        INTEGER*4   END_OF_LIST
      END STRUCTURE
      RECORD /PLACE/PLACE_LIST,ON_TOP ❷

      PLACE_LIST.CODE_1=WDPL$C_ABS_POS_X
      PLACE_LIST.ABS_POS_X=8 ❸
      PLACE_LIST.CODE_2=WDPL$C_ABS_POS_Y
      PLACE_LIST.ABS_POS_Y=8 ❹
      PLACE_LIST.END_OF_LIST=WDPL$C_END_OF_LIST

      ON_TOP.CODE_1=WDPL$C_ABS_POS_X
      ON_TOP.ABS_POS_X=8.5 ❺
      ON_TOP.CODE_2=WDPL$C_ABS_POS_Y
      ON_TOP.ABS_POS_Y=8.5 ❻
      ON_TOP.END_OF_LIST=WDPL$C_END_OF_LIST

      TYPE *,'ENTER DISPLAY SIZE'
      ACCEPT *,WIDTH,HEIGHT

      VD_ID=UIS$CREATE_DISPLAY(1.0,1.0,20.0,20.0,10.0,10.0)

      CALL UIS$CIRCLE(VD_ID,0,10.0,10.0,1.0)
      CALL UIS$PLOT(VD_ID,0,4.0,3.0,5.0,7.0)
      WD_ID1=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','CIRCLE',
     2        8.0,8.0,12.0,12.0,WIDTH,HEIGHT,PLACE_LIST) ❼
      WD_ID2=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','LINE',
     2        3.0,2.0,6.0,8.0,WIDTH,HEIGHT,ON_TOP)  ❽

      PAUSE      ❾

      CALL UIS$POP_VIEWPORT(WD_ID1) ❿
        .
        .
        .

      PAUSE

      CALL UIS$PUSH_VIEWPORT(WD_ID1) ⓫

      PAUSE

      CALL UIS$DELETE_DISPLAY(VD_ID)

      END
```

The program IMAGES_6 creates a data structure argument ❶, which it gives the symbolic name PLACE with the STRUCTURE statement. The program arbitrarily chooses symbolic names for the fields.

The program creates two type PLACE variables, PLACE_LIST and ON_TOP, ❷, which contain five longwords.

Actual values are assigned to the different fields of the record PLACE_LIST. In this case, the absolute coordinates of the lower-left corner 3 4 of the display viewport LINE are assigned to the fields ON_TOP.ABS_POS_X and ON_TOP.ABS_POS_Y 5 6. The absolute coordinates of the display viewport CIRCLE are assigned to the fields PLACEMENT.ABS_POS_X and PLACEMENT.ABS_POS_Y as well.

Also, the position of calls to UIS$CREATE_WINDOW 7 8 within the program is important. You must execute the call to create the display viewport CIRCLE before LINE.

At the first PAUSE statement 9, viewport LINE occludes viewport CIRCLE.

The program calls UIS$POP_VIEWPORT 10. The display viewport CIRCLE is placed over the viewport LINE.

A call to UIS$PUSH_VIEWPORT 11 returns the viewports to their orginal position.

8.5.5 Calling UIS$POP_VIEWPORT and UIS$PUSH_VIEWPORT

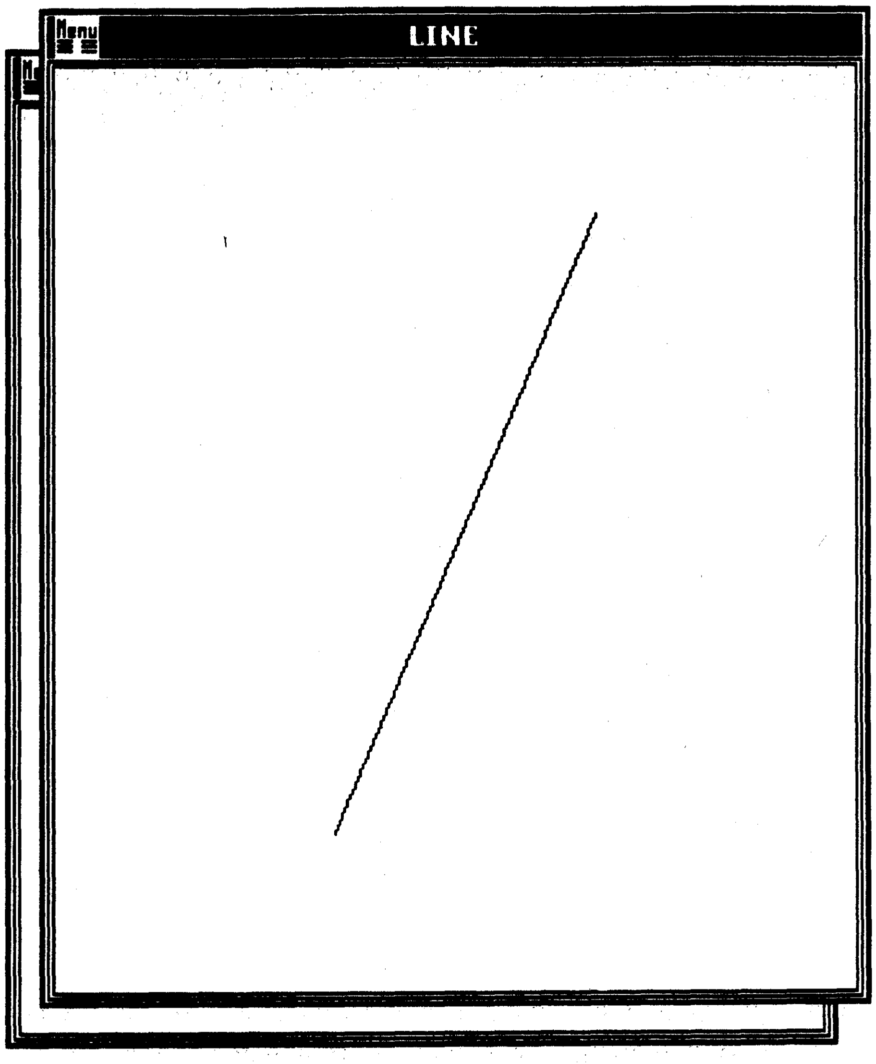
Initially, the viewport LINE is placed over CIRCLE. Note that display viewports are placed on the display screen with absolute coordinates. The lower-left corner of any viewport is the origin of the viewport rectangle. When you request exact placement of a viewport, you are specifying the location on display screen where the origin of the viewport rectangle is to be placed relative to the lower-left corner of the display screen.

Program execution is suspended at the first PAUSE statement. The display screen contains the graphic objects shown in Figure 8-7.

The display viewports LINE and CIRCLE change positions when the call to UIS$POP_VIEWPORT is executed. The viewport CIRCLE now occludes LINE as shown in Figure 8-8.

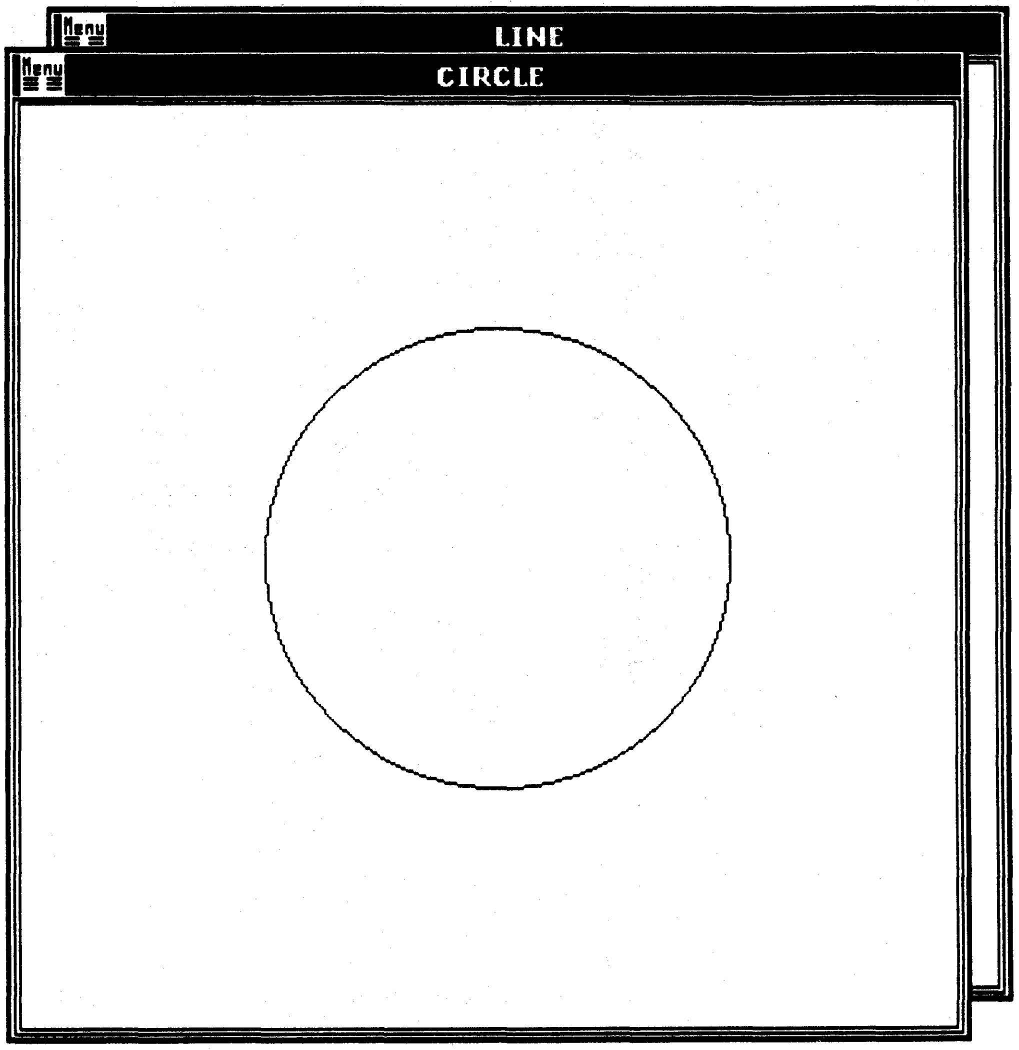
To return the viewports to their original positions, call UIS$PUSH_VIEWPORT. This pushes viewport CIRCLE behind viewport LINE as shown in Figure 8-9.

Figure 8–7 Occluding a Display Viewport



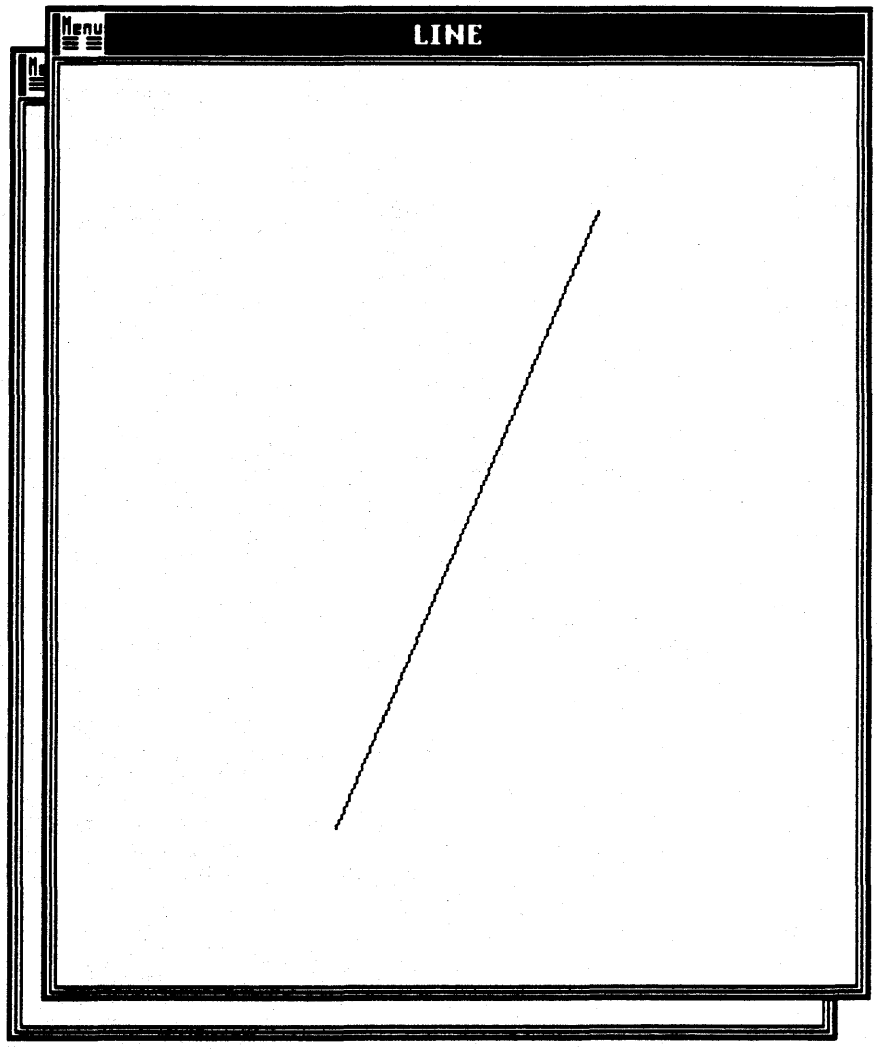
ZK-4539-85

Figure 8–8 Popping a Display Viewport



ZK-4540-85

Figure 8–9 Pushing a Display Viewport



8.5.6 Program Development III

Programming Objectives

To place a viewport in a general vicinity on the display screen and to create a display viewport with no border.

Programming Tasks

1 Create a viewport attributes list to hold the appropriate viewport placement and attributes data.

2 Create a virtual display.

3 Draw two graphic objects in the virtual display.

4 Create two display windows and associated viewports each with a graphic object.

5 Delete the virtual display.

```
PROGRAM IMAGES_7
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
REAL WIDTH,HEIGHT
STRUCTURE/PLACE/   ❶
INTEGER*4    CODE_5
INTEGER*4    REL_POS
INTEGER*4    CODE_6
INTEGER*4    ATTR
INTEGER*4    END_OF_LIST
END STRUCTURE
RECORD /PLACE/LOCATION(2)    ❷

LOCATION(1).CODE_5=WDPL$C_PLACEMENT
LOCATION(1).REL_POS=WDPL$M_TOP .OR. WDPL$M_LEFT ❸
LOCATION(1).CODE_6=WDPL$C_ATTRIBUTES
LOCATION(1).ATTR=WDPL$M_NOMENU_ICON
LOCATION(1).END_OF_LIST=WDPL$C_END_OF_LIST

LOCATION(2).CODE_5=WDPL$C_PLACEMENT
LOCATION(2).REL_POS=WDPL$M_RIGHT .OR. WDPL$M_BOTTOM  ❹
LOCATION(2).CODE_6=WDPL$C_ATTRIBUTES
LOCATION(2).ATTR=WDPL$M_NOBORDER
LOCATION(2).END_OF_LIST=WDPL$C_END_OF_LIST
TYPE *,'ENTER VIEWPORT WIDTH AND HEIGHT'
ACCEPT *,WIDTH,HEIGHT

VD_ID=UIS$CREATE_DISPLAY(1.0,1.0,20.0,20.0,10.0,10.0)

CALL UIS$CIRCLE(VD_ID,0,12.0,12.0,1.0)
CALL UIS$ELLIPSE(VD_ID,0,15.0,15.0,1.0,2.0)
WD_ID1=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','CIRCLE',
2       10.0,10.0,14.0,14.0,WIDTH,HEIGHT,LOCATION(1))
WD_ID4=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','ELLIPSE',
2       13.0,13.0,17.0,18.0,WIDTH,HEIGHT,LOCATION(2))

PAUSE

CALL UIS$DELETE_DISPLAY(VD_ID)

PAUSE

END
```

The program defines the name of the data structure argument PLACE with the STRUCTURE statement ❶. It defines an array LOCATION with two elements that are records; these records have a structure defined by the structure PLACE ❷. Each record LOCATION(1) and LOCATION(2) consists of two pairs of longwords terminated by a longword that equals the constant WDPL$C_END_OF_LIST.

To place the display viewport CIRCLE in the upper-left corner of the display screen and the borderless viewport ELLIPSE in the lower-right corner, specify in each assignment two preference masks for each viewport 3 4.

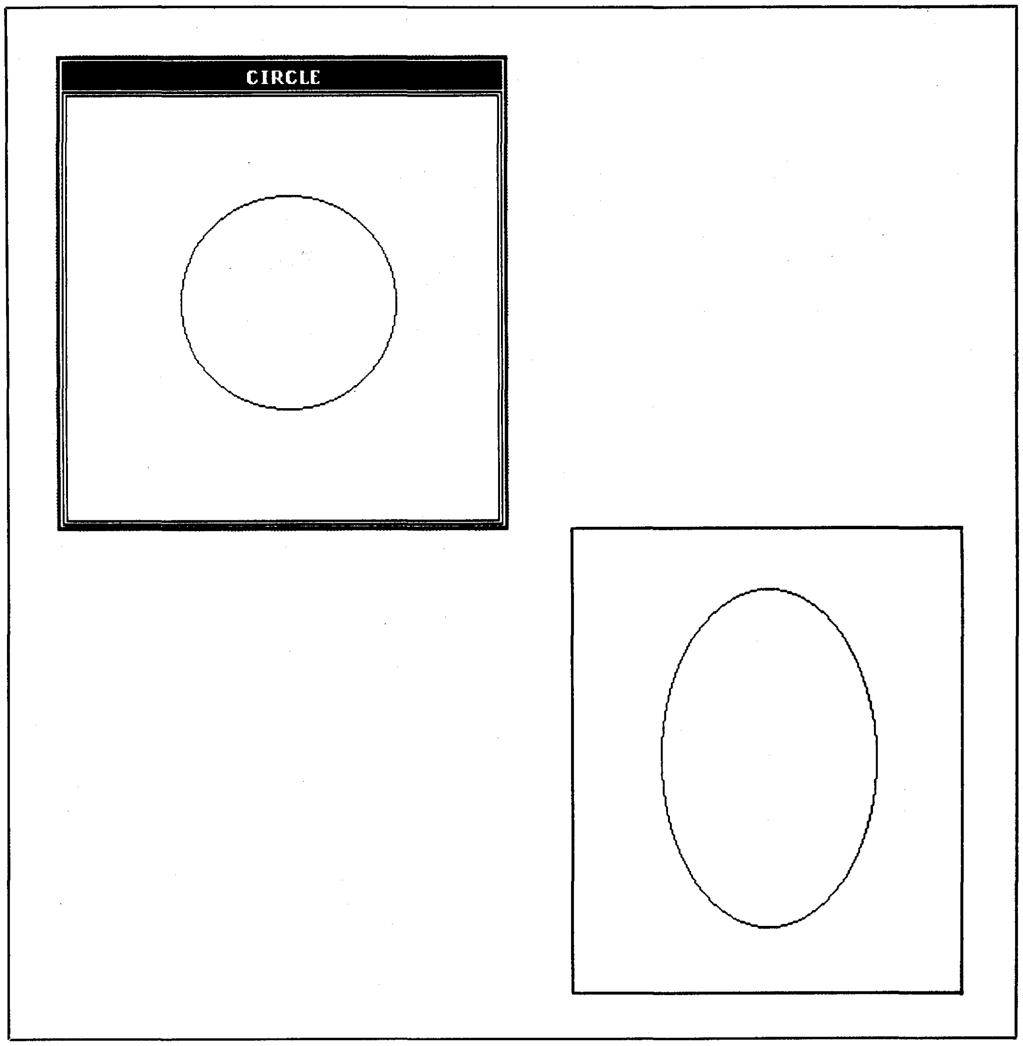
NOTE: **Note that you must use the logical operator .OR. when you specify more than one preference mask.**

The array name LOCATION is added to the argument lists of the viewport CIRCLE and ELLIPSE to invoke the optional attribute list.

8.5.7 Requesting General Placement and No Border

General display viewport placement works best on an uncluttered screen. Your workstation screen displays the objects shown in Figure 8–10.

Figure 8–10 General Placement and No Border



8.5.8 Program Development IV

Programming Objective

To move graphic objects within the virtual display.

Programming Tasks

1 Create a virtual display.
2 Create a display window and viewport.
3 Draw two graphic objects in the virtual display.

4 Use UIS$MOVE_AREA to move the coordinate space that contains each graphic object to another portion of the virtual display.

```
PROGRAM AREA
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
VD_ID=UIS$CREATE_DISPLAY(0.0,0.0,50.0,50.0,15.0,15.0)
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','MOVE AREA')

CALL UIS$PLOT(VD_ID,0,1.0,25.0,16.0,25.0,9.0,42.0,1.0,25.0)   ❶
CALL UIS$CIRCLE(VD_ID,0,35.0,35.0,10.0)       ❷
PAUSE
CALL UIS$MOVE_AREA(VD_ID,0.0,22.0,20.0,42.0,30.0,1.0)    ❸
CALL UIS$MOVE_AREA(VD_ID,25.0,25.0,50.0,50.0,1.0,1.0)    ❹

PAUSE

END
```

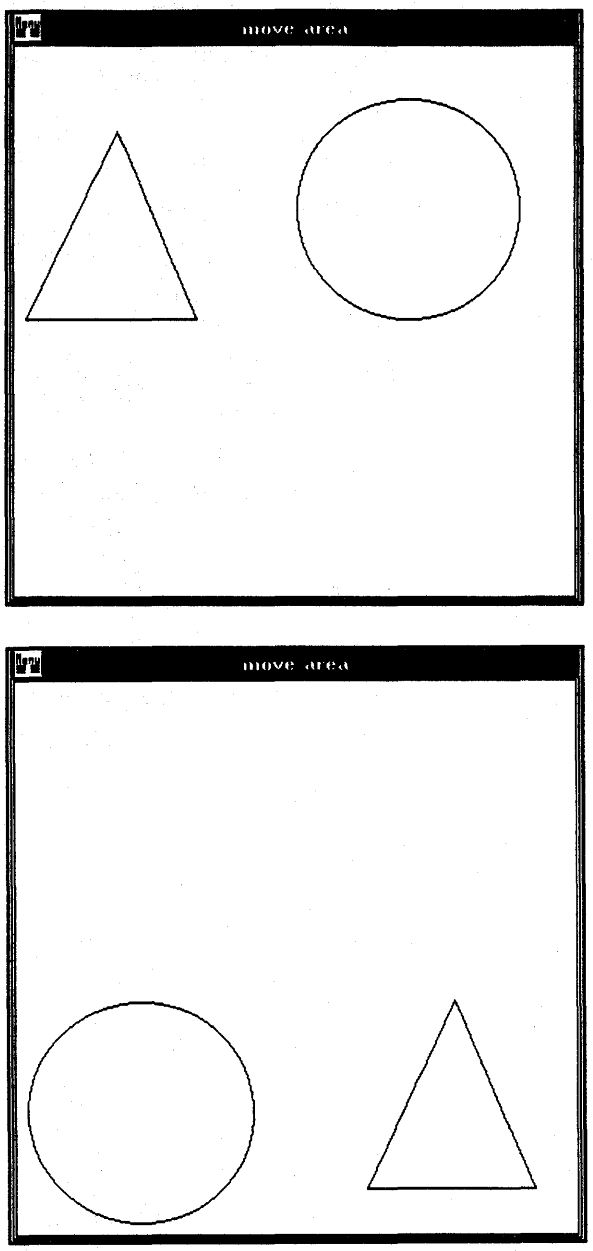
The program uses UIS$PLOT and UIS$CIRCLE ❶ ❷ to draw a triangle and a circle in the upper half of the virtual display.

A rectangular area containing the triangle is moved to the lower-right area of the virtual display ❸. A rectangular area containing the circle is moved to the lower-left region in the virtual display ❹.

8.5.9 Calling UIS$MOVE_AREA

Figure 8-11 illustrates how graphic objects in areas within the virtual display can be moved to other parts of the same virtual display.

Figure 8–11 Moving Graphic Objects Within the Virtual Display



ZK 4623 85

8.6 World Coordinate Transformations

Certain applications require that you create more than one virtual display, or *world coordinate space*. Depending on the requirements of the program, you might have to map graphic objects in one virtual display to another virtual display.

8.6.1 Programming Options

To see the advantages of world coordinate transformations, construct a program that creates a virtual display. Then create a circle in a virtual display. The circle is written to new world coordinate space or transformation space.

Two-Dimensional Transformation and Scaling

Depending on the values supplied to UIS$CREATE_TRANSFORMATION, you can scale graphic objects that are mapped to other coordinate spaces. If the coordinates of the new transformation space are the same as those of the original virtual display, no scaling occurs.

8.6.2 Program Development

Programming Objectives

To transform a world coordinate space by altering its mapping and scaling factors.

Programming Tasks

1. Create a virtual display.
2. Create a display window and viewport.
3. Draw a graphic object in the virtual display.
4. Use UIS$CREATE_TRANSFORMATION to create a new coordinate space.
5. Redraw the graphic object: substitute the transformation identifier of the new coordinate space returned by UIS$CREATE_TRANSFORMATION for the virtual display identifier of the old coordinate space.

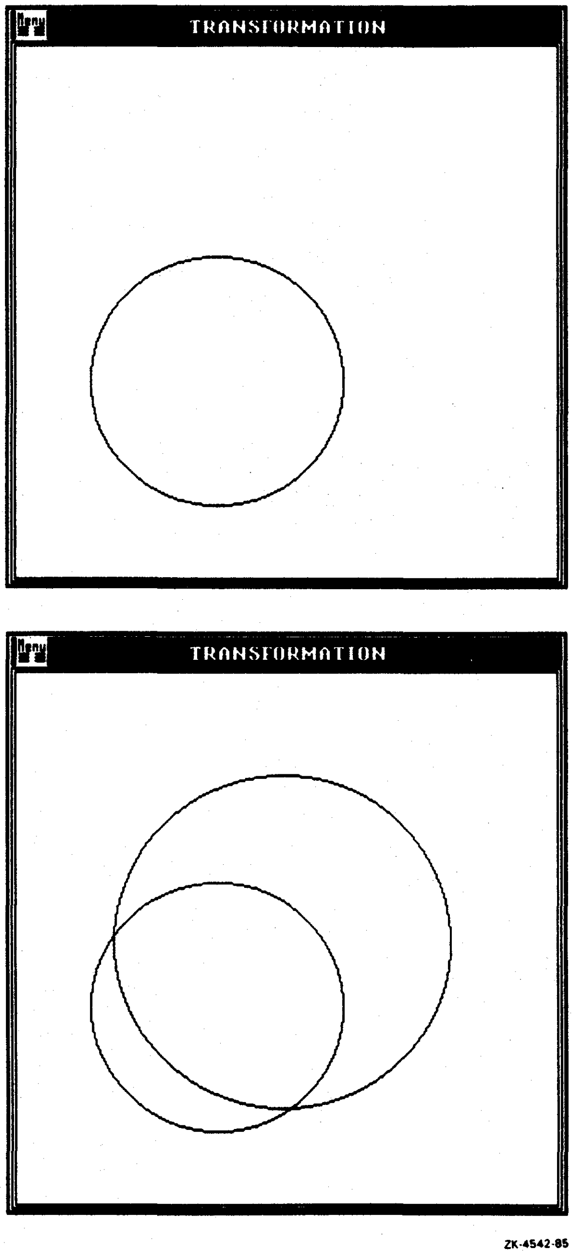
```
        PROGRAM TRANS
        IMPLICIT INTEGER(A-Z)
        INCLUDE 'SYS$LIBRARY:UISENTRY'
        INCLUDE 'SYS$LIBRARY:UISUSRDEF'

        VD_ID=UIS$CREATE_DISPLAY(-5.0,-5.0,25.0,25.0,10.0,10.0) ❶
        WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','TRANSFORMATION')
        CALL UIS$CIRCLE(VD_ID,0,6.0,6.0,7.0) ❷

        PAUSE

        TR_ID=UIS$CREATE_TRANSFORMATION(VD_ID,-5.0,-5.0,
     2          17.5,17.5) ❸
        CALL UIS$CIRCLE(TR_ID,0,6.0,6.0,7.0) ❹

        PAUSE

        END
```

The virtual display ❶ and the new transformation space ❸ specify different coordinate ranges. You create the circles with calls to UIS$CIRCLE ❷ ❹; you substitute the **tr_id** argument for the **vd_id** argument in the second call. The same circle is redrawn with the same world coordinates in the new transformation space.

8.6.3 Calling UIS$CREATE_TRANSFORMATION

The graphic objects appear to be superimposed one over the other. If you manipulate the **vdx1** and **vdy1** arguments, the size of the arc can increase or decrease relative to the size of the first circle. In any case, the arc is mapped to the transformation space, which eliminates the need for additional computation and coding. Figure 8–12 shows world coordinate transformation.

Figure 8–12 World Coordinate Transformations



9 General Attributes

9.1 Overview

Chapters 1 through 8 describe UIS output routines that create the basic structures you use to produce graphic objects. Other types of routines establish attributes that allow you to enhance the quality of graphic objects and text. This chapter discusses general attribute routines.

9.2 Attributes

Attributes enhance the appearance of graphic objects and text on the display screen. You can modify attributes in your program at any time.

9.2.1 Attribute Blocks

All UIS attributes are grouped in a data structure called an *attribute block*. You can modify attributes within a given attribute block. Default attribute settings reside in *attribute block* 0. Table 9-1 lists the categories of attributes within attribute block 0.

Table 9-1 Attribute Block 0

Type	Attribute
General	Writing mode
	Writing color index
	Background color
Text	Character rotation
	Character spacing
	Character slant
	Character size
	Text path
	Text slope
	Text formatting
	Left margin
	Right margin
	Font
Graphics	Line width
	Line style
	Fill pattern
	Arc type

Table 9–1 (Cont.) Attribute Block 0

Type	Attribute
Windowing	Clipping rectangle

9.2.2 Modifying General Attributes

When you modify general attributes, you cannot change the default attribute settings within attribute block 0 itself. Think of attribute block 0 as a template of default settings, a copy of which you modify for use within your program.

Attribute modification routines contain two arguments:

iatb—Input attribute block number

oatb—Output attribute block number

Table 9–2 lists the default settings of general attributes.

Table 9–2 Default Settings of General Attributes

General Attribute	Default Setting	Modification Routine
Background index[1]	Index 0	UIS$SET_BACKGROUND_INDEX
Writing index[2]	Index 1	UIS$SET_WRITING_INDEX
Writing mode	Overlay	UIS$SET_WRITING_MODE

[1]Background color index in the virtual color map.

[2]Foreground color index in the virtual color map.

Use the following procedure to modify attributes:

1. Choose an appropriate attribute modification routine.
2. Specify *0* as the **iatb** argument to obtain a copy of attribute block 0.
3. Specify a number from *1* to *255* as the **oatb** argument. You can then reference the attribute block in subsequent UIS graphics and text routines or in any other attribute modification routine.

The following routines reference attribute blocks in the **atb** argument:

- Graphics and text routines
- UIS$MEASURE_TEXT
- UIS$NEW_TEXT_LINE
- UIS$SET_ALIGNED_POSITION

9.3 Structure of Graphic Objects

There are three types of graphic objects:

- Geometric shapes, including:
  - Circles
  - Ellipses
  - Points
  - Lines
  - Polygons
- Text
- Raster images

Graphic objects are made from a *pattern*. In memory, the pattern represents one or more bit settings to *0* or *1* that constitute the actual graphic object.

The UIS writing modes translate the bit settings that constitute these objects and write them in the virtual display.

Text

In the case of text, a standard character within the default font displayed on the workstation screen represents the bitmap image of a *cell* in memory. The size of the cell depends on the type of font:

- Monospaced fonts—Use a standard cell size for all letters within the font; however, the standard cell size varies according to the font you use.
- Proportionally spaced fonts—Use character cells that vary in width according to the letter used; character cell height remains constant for all characters within the font.

The character cell contains the pattern. The remaining bits in the cell are set to *0*. All bits within the character cell are significant to UIS writing modes.

Geometric Shapes

In the case of geometric shapes, only the bit settings that actually compose the pattern are significant. Bit settings in the pattern can be *0* or *1*. For example, a dotted line represents bit settings of *0* and *1* in a pattern. All bit settings, both *0* and *1* within this pattern, are significant to UIS writing modes.

Raster Images

When you draw a raster image, set bits in a bitmap to create text characters or geometric shapes. For example, UIS$IMAGE and UIS$SET_POINTER_ PATTERN use bitmaps to map rasters to the display screen. All bits in the bitmap are significant to UIS writing modes. The following table shows the underlying structures from which graphic objects are created.

Graphic Object	Structure
Text	Character cell
Geometric shapes	Pattern
Raster Image	Bitmap image of varying size

For a given graphic object, the current writing mode determines how bit settings in the appropriate structure are displayed. All bit settings of a particular structure are significant to UIS writing modes. Figure 9–1 shows graphic objects as structures that UIS writing modes recognize:

- The letter E within a character cell
- A square as a pattern
- A bitmap that contains the letter E, a square, and a vertical dashed line of double thickness

9.4 UIS Writing Modes

There are 14 UIS writing modes: transparent, complement, copy, copy negate, overlay, overlay negate, erase, erase negate, replace, replace negate, bit set, bit set negate, bit clear, and bit clear negate.

Writing modes control how graphics and text routines use foreground and background colors to display graphic objects. The default writing mode is overlay.

Table 9–3 lists each writing mode and its functions.

Figure 9–1 Structure of Graphic Objects

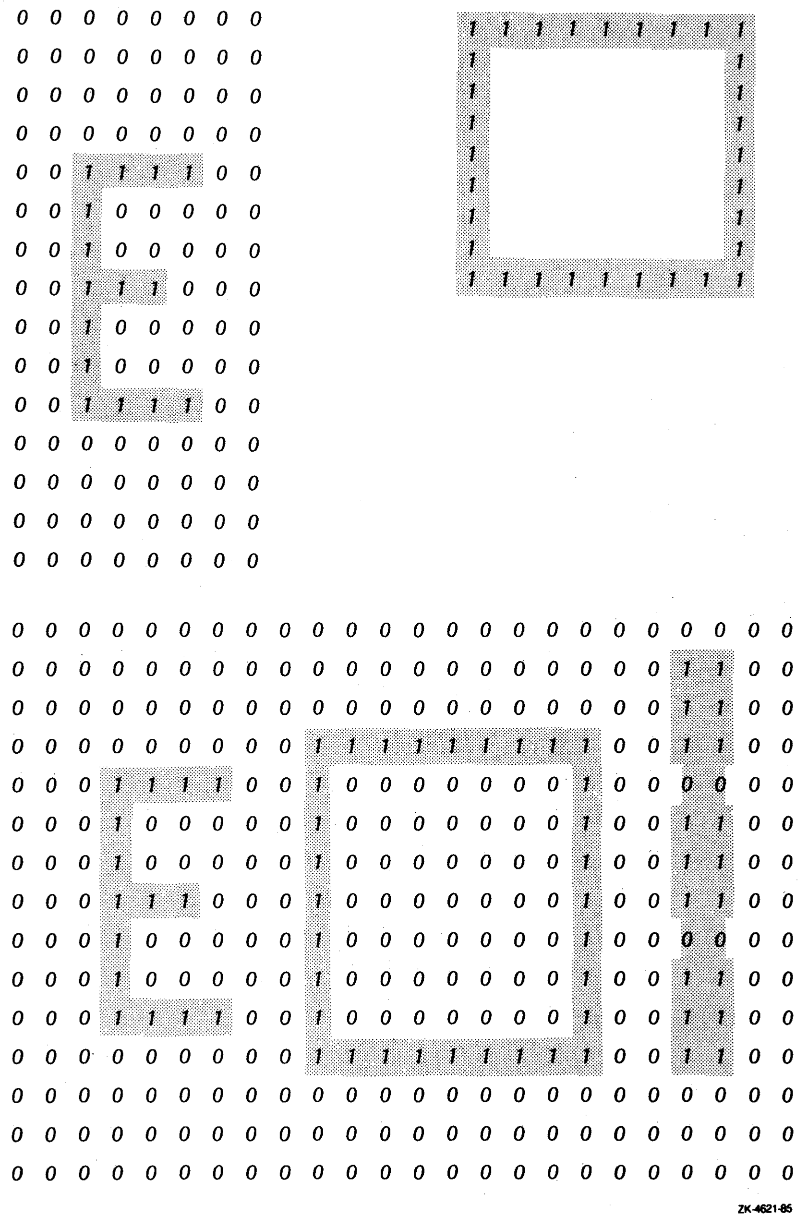


Table 9–3 UIS Writing Modes

UIS Writing Modes	Function
Device-Independent	
UIS$C_MODE_ERAS	Displays the current background color for each bit position no matter what the bit settings are in the character cell, pattern, or bitmap image.
UIS$C_MODE_ERASN	Displays the current writing color for each bit position no matter what the bit settings are in the character cell, pattern, or bitmap image.
UIS$C_MODE_OVER	Displays the current writing color for bits set to *1* in the character cell, pattern, or bitmap image. All bits set to *0* have no effect on the existing graphic object. This is the default writing mode attribute setting.
UIS$C_MODE_OVERN	Bitwise complements the character cell, pattern, or bitmap image that is, bits originally set to *0* are now set to *1* and vice versa. The bits now set to *1* in the character cell, pattern, or bitmap image display the current writing color. The bits that are now set to *0* in the character cell have no effect on any existing graphic object.
UIS$C_MODE_REPL	Displays the current writing color for bits set to *1* in the character cell, pattern, or bitmap image. Bits set to *0* in the character cell, pattern, or bitmap image display the current background color.
UIS$C_MODE_REPLN	Bitwise complements the character cell, pattern, or bitmap image. The bits now set to *1* in the character cell, pattern, or bitmap image now display the current writing color. Bits now set to *0* in the character cell, pattern, or bitmap image now display the current background color.
UIS$C_MODE_COMP	Where the two graphic objects intersect, the bits in the character cell, pattern, or bitmap image are exclusive .OR.ed with the existing graphic object.
UIS$C_MODE_TRAN	Does not alter the display screen.
Device-Dependent[1]	
UIS$C_MODE_BIC	The bitwise complement of the character cell, pattern, or bitmap image is logically .AND.ed with the existing graphic object and background. On mapped color systems, where the two graphic objects intersect, the bitwise complement of the writing index of the character cell, pattern, or bitmap image is logically .AND.ed with the pixel values of the existing graphic object and background.

[1]These UIS writing modes produce device-dependent results. Depending on the specific operation, graphic objects drawn using these writing modes may appear differently on VAXstation monochrome and color systems.

Table 9-3 (Cont.) UIS Writing Modes

UIS Writing Modes	Function
Device-Dependent[1]	
UIS$C_MODE_BICN	On monochrome systems, the bits in the character cell, pattern, or bitmap image are logically .AND.ed with the existing graphic object and background. On mapped color systems, the writing index of the character cell, pattern, or bitmap image is logically .AND.ed with the pixel values of the existing graphic object and background.
UIS$C_MODE_BIS	The bits in the character cell, pattern, or bitmap image are logically .OR.ed with the existing graphic object and background. On mapped color systems, the writing index of the character cell, pattern, or bitmap image is logically .OR.ed with the pixel values of the existing graphic object and background.
UIS$C_MODE_BISN	On monochrome systems, the bitwise complement of the character cell, pattern, or bitmap image is logically .OR.ed with the existing graphic object and background. On color systems, the bitwise complement of the writing index of the character cell, pattern, or bitmap image is logically .OR.ed with the pixel values of the existing graphic object and background.
UIS$C_MODE_COPY	Displays the character cell, pattern, or bitmap image without regard to current background and writing color. On a VAXstation monochrome system, bits set to *0* are black, and bits set to *1* are white. On mapped color systems, the writing index of the character cell, pattern, or bitmap is used directly as an index.
UIS$C_MODE_COPYN	Displays the character cell, pattern, or bitmap image without regard to current background and writing color. On monochrome systems, bits set to *0* are white and bits set to *1* are black. On mapped color systems, the bitwise complement of the writing index of the character cell, pattern, or bitmap image is used directly as an index.

[1]These UIS writing modes produce device-dependent results. Depending on the specific operation, graphic objects drawn using these writing modes may appear differently on VAXstation monochrome and color systems.

9.4.1 Using General Attributes

General attributes (background color, writing color or foreground, and writing mode) affect all graphic images on the screen.

9.4.1.1 Programming Options

For application-specific reasons or simply for variety, a program can set different background and writing colors for different display viewports.

Setting the Background Color

Modifying the background color attribute sets the value of an index into the color map. Modifying the background color affects how the current writing mode interprets the bits that compose the graphic object background color. Set the background color attribute with UIS$SET_BACKGROUND_INDEX.

Setting the Writing Color

Modifying the writing color attribute sets the value of an index into the color map. Writing color affects the color of the graphic object. Set the writing color with UIS$SET_WRITING_INDEX.

Setting the Writing Mode

Writing mode controls how background and foreground colors are used to draw graphic objects in the virtual display. Use UIS$SET_WRITING_MODE to specify writing mode.

9.4.1.2 Program Development I
Programming Objective

To use the default background and writing color attribute settings to draw a graphic object in each of the UIS device-independent writing modes.

Programming Tasks

1 Create a virtual display.

2 Create a display window and associated viewport.

3 Draw a line using the default overlay writing mode in the virtual display.

4 Draw a character at the same location in each of the UIS writing modes.

5 Use UIS$ERASE to erase graphic objects in the virtual display and use UIS$DELETE_WINDOW to delete the window.

6 Repeat steps 3 through 5.

The font name MY_FONT_5 is a logical name.

```
        PROGRAM MODE
        IMPLICIT INTEGER(A-Z)
        INCLUDE 'SYS$LIBRARY:UISENTRY'
        INCLUDE 'SYS$LIBRARY:UISUSRDEF'
        VD_ID=UIS$CREATE_DISPLAY(0.0,0.0,3.0,3.0,6.0,5.0)
        WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION')

        CALL UIS$PLOT(VD_ID,0,0.5,1.0,2.0,2.5)

        PAUSE

C       Erase the object in the virtual display and delete the window
C       Display window is deleted in order to change viewport title

        CALL UIS$ERASE(VD_ID,0.0,0.0,3.0,3.0)
        CALL UIS$DELETE_WINDOW(WD_ID)

        PAUSE
```

```
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','OVERLAY')
CALL UIS$SET_FONT(VD_ID,0,1,'MY_FONT_5')
CALL UIS$PLOT(VD_ID,0,0.5,1.0,2.0,2.5)
CALL UIS$TEXT(VD_ID,1,'D',1.0,2.0)

PAUSE

CALL UIS$ERASE(VD_ID,0.0,0.0,3.0,3.0)
CALL UIS$DELETE_WINDOW(WD_ID)

PAUSE
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','OVERLAY NEGATE')
CALL UIS$SET_WRITING_MODE(VD_ID,1,2,UIS$C_MODE_OVERN)
CALL UIS$PLOT(VD_ID,0,0.5,1.0,2.0,2.5)
CALL UIS$TEXT(VD_ID,2,'D',1.0,2.0)

PAUSE

CALL UIS$ERASE(VD_ID,0.0,0.0,3.0,3.0)
CALL UIS$DELETE_WINDOW(WD_ID)

PAUSE
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','REPLACE')
CALL UIS$SET_WRITING_MODE(VD_ID,2,3,UIS$C_MODE_REPL)
CALL UIS$PLOT(VD_ID,0,0.5,1.0,2.0,2.5)
CALL UIS$TEXT(VD_ID,3,'D',1.0,2.0)
PAUSE

CALL UIS$ERASE(VD_ID,0.0,0.0,3.0,3.0)
CALL UIS$DELETE_WINDOW(WD_ID)

PAUSE
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','REPLACE NEGATE')
CALL UIS$SET_WRITING_MODE(VD_ID,3,4,UIS$C_MODE_REPLN)
CALL UIS$PLOT(VD_ID,0,0.5,1.0,2.0,2.5)
CALL UIS$TEXT(VD_ID,4,'D',1.0,2.0)

PAUSE
CALL UIS$ERASE(VD_ID,0.0,0.0,3.0,3.0)
CALL UIS$DELETE_WINDOW(WD_ID)

PAUSE

WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','ERASE')
CALL UIS$SET_WRITING_MODE(VD_ID,4,5,UIS$C_MODE_ERAS)
CALL UIS$PLOT(VD_ID,0,0.5,1.0,2.0,2.5)
CALL UIS$TEXT(VD_ID,5,'D',1.0,2.0)

PAUSE

CALL UIS$ERASE(VD_ID,0.0,0.0,3.0,3.0)
CALL UIS$DELETE_WINDOW(WD_ID)

PAUSE
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','ERASE NEGATE')
CALL UIS$SET_WRITING_MODE(VD_ID,5,6,UIS$C_MODE_ERASN)

CALL UIS$PLOT(VD_ID,0,0.5,1.0,2.0,2.5)
CALL UIS$TEXT(VD_ID,6,'D',1.0,2.0)

PAUSE

CALL UIS$ERASE(VD_ID,0.0,0.0,3.0,3.0)
CALL UIS$DELETE_WINDOW(WD_ID)

PAUSE
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','TRANSPARENT')
CALL UIS$SET_WRITING_MODE(VD_ID,6,7,UIS$C_MODE_TRAN)

CALL UIS$PLOT(VD_ID,0,0.5,1.0,2.0,2.5)
CALL UIS$TEXT(VD_ID,7,'D',1.0,2.0)

PAUSE
CALL UIS$ERASE(VD_ID,0.0,0.0,3.0,3.0)
CALL UIS$DELETE_WINDOW(WD_ID)

PAUSE
```

```
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','COMPLEMENT')
CALL UIS$SET_WRITING_MODE(VD_ID,7,8,UIS$C_MODE_COMP)
CALL UIS$PLOT(VD_ID,0,0.5,1.0,2.0,2.5)
CALL UIS$TEXT(VD_ID,8,'D',1.0,2.0)

PAUSE

END
```

The program MODE sets the writing mode attribute ten times. The letter D is placed over the line. Table 9-3 describes the behavior of the UIS writing modes when text or geometric shapes such as circles are placed on top of an existing graphic object. Remember, character cells refer to text and patterns refer to geometric shapes.

NOTE: **Before you run the MODE demonstration program, define fonts by invoking the file DEFFONT.COM in the directory SYS$EXAMPLES: as follows: @SYS$EXAMPLES:DEFFONT**

If the documentation pictures do not look the same as those produced with the demonstration program, adjust the brightness on the terminal screen.

9.4.1.3 Calling UIS$SET_BACKGROUND_INDEX, UIS$SET_WRITING_INDEX, and UIS$SET_WRITING_MODE

To illustrate the effects of writing modes, imagine that the character cell is slowly lowered onto the virtual display containing an existing graphic object drawn in Overlay mode—a line. As the character cell approaches the plane of the virtual display, the writing mode of the character cell determines the final appearance of the graphic object. See Table 9-3 for a description of each writing mode.

The default background and writing color are in effect as shown in Figure 9-2.

Figure 9–2 UIS Device-Independent Writing Modes

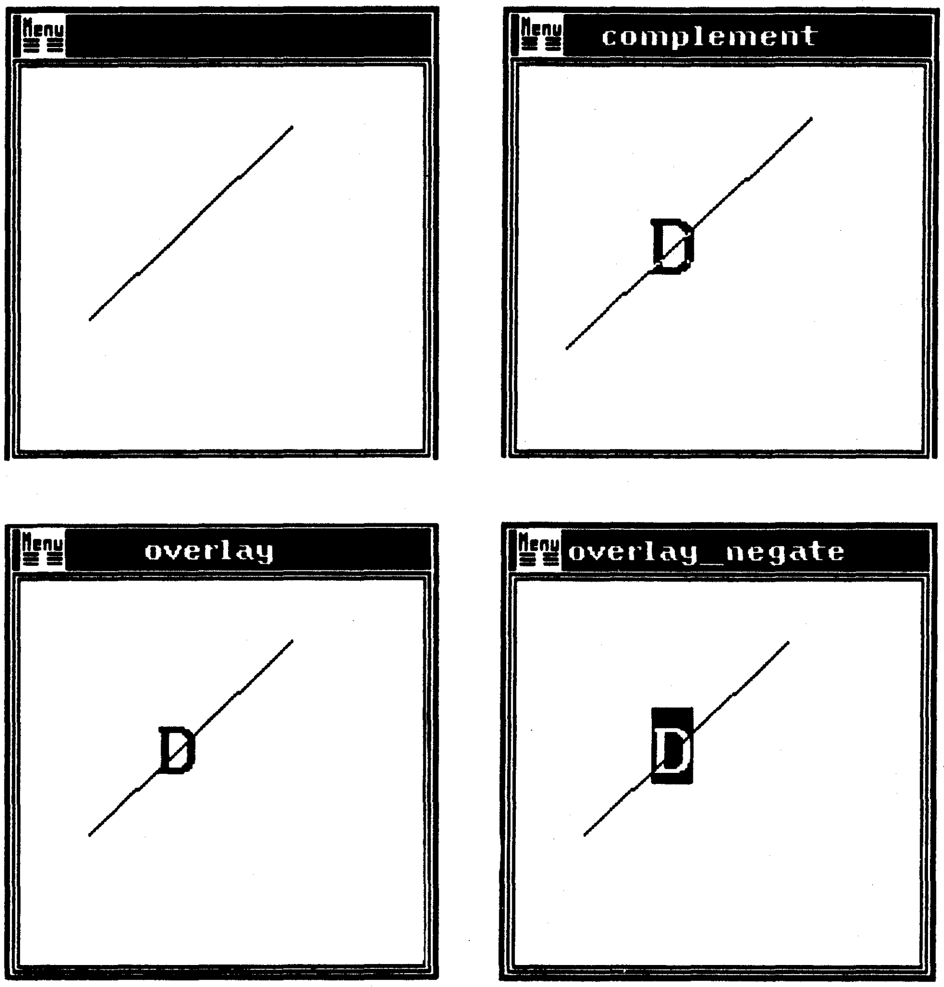


Figure 9–2 Cont'd. on next page

Figure 9–2 (Cont.) UIS Device-Independent Writing Modes

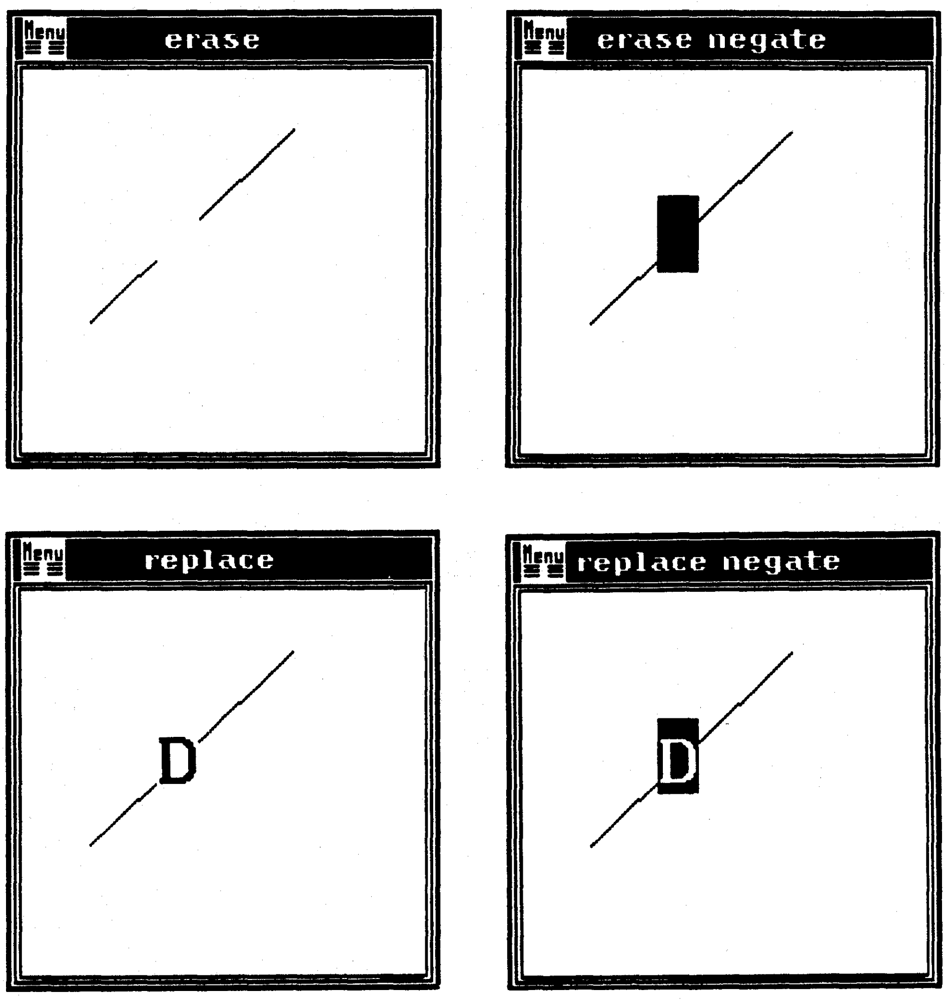
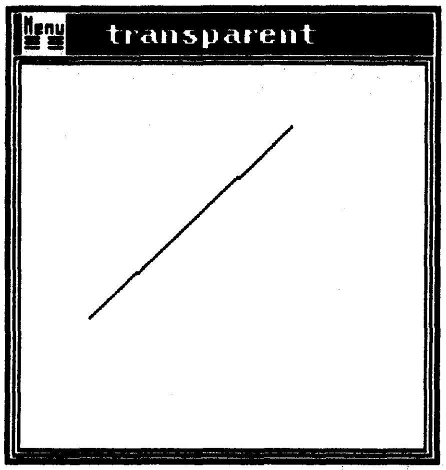


Figure 9–2 Cont'd. on next page

Figure 9-2 (Cont.) UIS Device-Independent Writing Modes



ZK-4545-85

9.4.1.4 Program Development II

Programming Objective

To illustrate the behavior of device-dependent writing modes.

Programming Tasks

1 Create an eight-entry virtual color map containing intensity values.

2 Draw three overlapping circles—one in overlay mode and two in bit set mode.

3 Redraw the same circles—one in overlay mode, one in bit clear mode, and one in bit set mode.

4 Redraw two of the circles in the remaining device-dependent writing modes. One circle is always drawn in overlay mode. Both are drawn with the same writing index.

```
PROGRAM PLANE_MODES
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
INCLUDE 'SYS$LIBRARY:UISENTRY'
REAL*4 I_VECTOR(8)                                      1
DATA I_VECTOR/0.0,0.125,0.25,0.375,0.50,0.625,0.75,1.0/   2
DATA VCM_SIZE/8/                                          3
DATA INDEX2/2/                                      4
DATA INDEX4/4/                                    5

VCM_ID=UIS$CREATE_COLOR_MAP(VCM_SIZE)
VD_ID=UIS$CREATE_DISPLAY(0.0,0.0,40.0,40.0,15.0,15.0,VCM_ID)
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION')
CALL UIS$SET_INTENSITIES(VD_ID,0,8,I_VECTOR)

CALL UIS$SET_FONT(VD_ID,0,1,'UIS$FILL_PATTERNS')
CALL UIS$SET_FILL_PATTERN(VD_ID,1,1,PATT$C_FOREGROUND)

CALL UIS$SET_FONT(VD_ID,0,2,'UIS$FILL_PATTERNS')
CALL UIS$SET_WRITING_INDEX(VD_ID,2,2,INDEX2)      6
CALL UIS$SET_WRITING_MODE(VD_ID,2,2,UIS$C_MODE_BIS)
CALL UIS$SET_FILL_PATTERN(VD_ID,2,2,PATT$C_FOREGROUND)
CALL UIS$SET_WRITING_INDEX(VD_ID,2,4,INDEX4)                  7

CALL UIS$CIRCLE(VD_ID,1,15.0,20.0,10.0)     8
CALL UIS$CIRCLE(VD_ID,2,25.0,20.0,10.0)     9
CALL UIS$CIRCLE(VD_ID,4,20.0,30.0,10.0)     10

PAUSE

CALL UIS$SET_WRITING_MODE(VD_ID,4,4,UIS$C_MODE_BIC)  11
CALL UIS$CIRCLE(VD_ID,4,20.0,30.0,10.0)

PAUSE

CALL UIS$ERASE(VD_ID)
CALL UIS$DELETE_WINDOW(WD_ID)

PAUSE

WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION')
CALL UIS$SET_WRITING_MODE(VD_ID,2,2,UIS$C_MODE_BICN)
CALL UIS$CIRCLE(VD_ID,1,15.0,25.0,10.0)                12
CALL UIS$CIRCLE(VD_ID,2,25.0,25.0,10.0)               13

PAUSE

CALL UIS$ERASE(VD_ID)
CALL UIS$DELETE_WINDOW(WD_ID)

PAUSE

WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION')
CALL UIS$SET_WRITING_MODE(VD_ID,2,2,UIS$C_MODE_BISN)
CALL UIS$CIRCLE(VD_ID,1,15.0,25.0,10.0)                 14
CALL UIS$CIRCLE(VD_ID,2,25.0,25.0,10.0)               15

PAUSE

CALL UIS$ERASE(VD_ID)
CALL UIS$DELETE_WINDOW(WD_ID)

PAUSE
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION')
CALL UIS$SET_WRITING_MODE(VD_ID,2,2,UIS$C_MODE_COPY)
CALL UIS$CIRCLE(VD_ID,1,15.0,20.0,10.0)               16
CALL UIS$CIRCLE(VD_ID,2,25.0,20.0,10.0)             17

PAUSE

CALL UIS$ERASE(VD_ID)
CALL UIS$DELETE_WINDOW(WD_ID)

PAUSE

WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION')
CALL UIS$SET_WRITING_MODE(VD_ID,2,2,UIS$C_MODE_COPYN)
CALL UIS$CIRCLE(VD_ID,1,15.0,20.0,10.0)                 18
CALL UIS$CIRCLE(VD_ID,2,25.0,20.0,10.0)               19

PAUSE
END
```

An array I_VECTOR is declared to hold the intensity values 1. Each location in the array element is initialized with an intensity value 2. The color map size variable is initialized to the number of color map entries 3. Color index variables *index2* and *index4* are initialized 4 5.

Three circles are drawn 8 9 10 with three different indices in the virtual color map—index 1 (the default), index 2, and index 4 6 7. The circles are filled with the current foreground color. The following table lists the circles, their writing modes and indices, and corresponding intensity values.

Circle	Writing Mode	Writing Index	Intensity Value
1	Overlay	1	0.0
2	Bit Set	2	0.125
3	Bit Set	4	0.375

The three circles are redrawn with circle 3 drawn in bit clear mode 11.

In subsequent drawings, only overlapping circles 1 and 2 are redrawn. Circle 1 is always drawn in overlay mode 8 12 14 16 18, and circle 2 is drawn in the remaining writing modes 9 13 15 17 19.

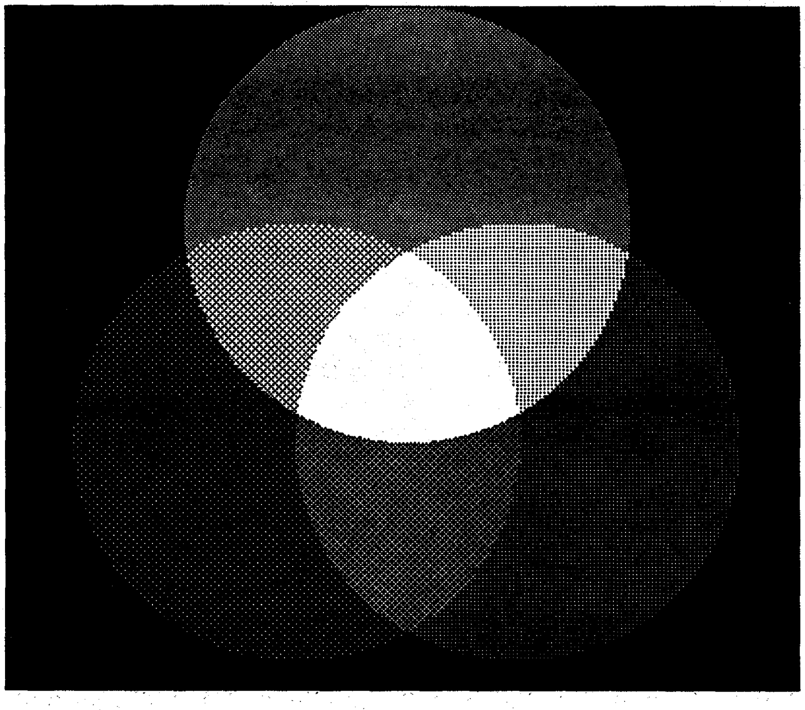
9.4.1.5 Using Device-Dependent Writing Modes

The program PLANE_MODES produces Figures 9–3 through 9–8. In each figure, the circle on the left (circle 1) is drawn in overlay mode and writing index 1. The circle on the right (circle 2) is drawn in a different writing mode with writing index 2. The top circle (circle 3), in Figures 9–3 and 9–4 only, is drawn with writing index 4. The following table lists the writing indices, their binary value, and binary bitwise complements.

Object	Writing Index	Binary Value	Bitwise Complement
Background	0	000_2	111_2
Circle 1	1	001_2	110_2
Circle 2	2	010_2	101_2
Circle 3	4	100_2	011_2

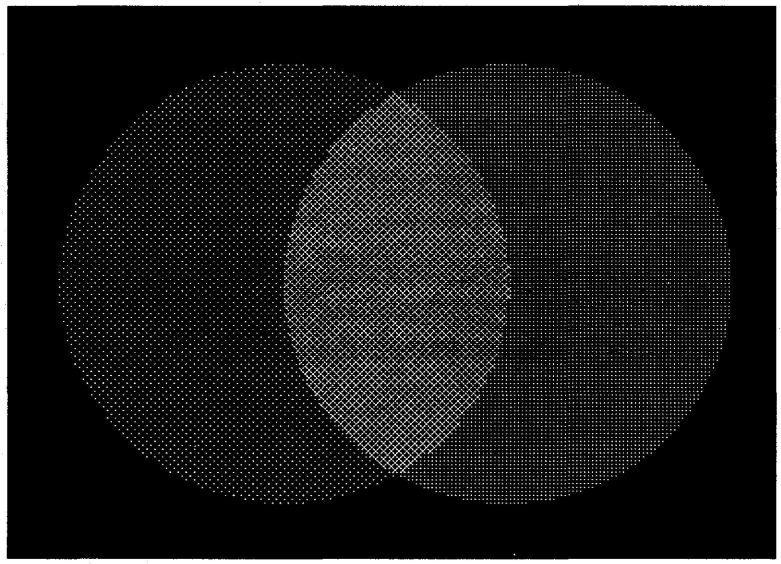
In Figure 9–3, whenever the circles 1, 2, and 3 intersect, their writing indices 001_2, 010_2, and 100_2 are logically .OR.ed with the pixel values of the existing graphic objects and the background. The bit set writing mode has the effect of combining the value of the bit plane settings of each object. Therefore, the intersections of the circles are lighter than the rest of the circles.

Figure 9-3 Bit Set Mode



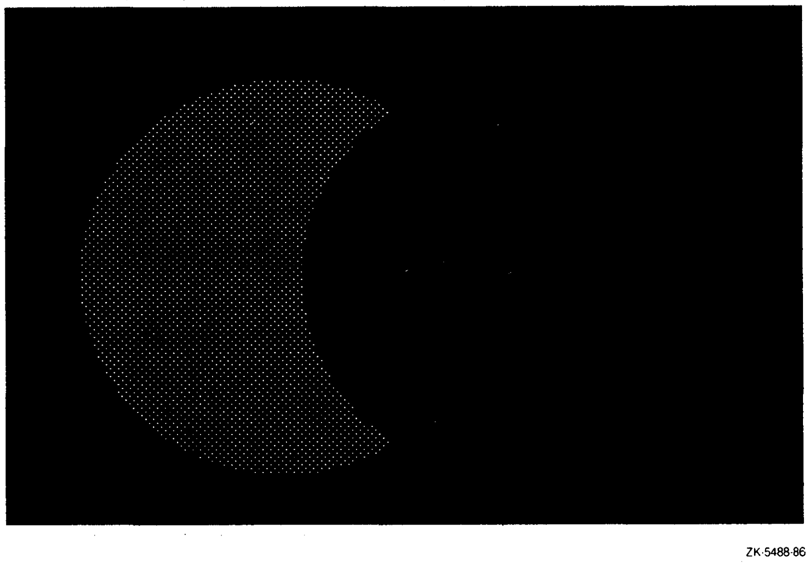
In Figure 9-4, circle 3 is drawn in bit clear mode with a writing index of 4 or 100_2. Circle 2 is drawn in bit set mode in writing index 2 or 011_2. The binary bitwise complement of the writing index of circle 3 is 011_2. It is logically .AND.ed with the pixel values of the existing graphic objects—circle 1, circle 2, and the background. In bit clear mode, the appropriate bit plane settings are now changed such that circle 3 appears to blend into the background of circles 1 and 2.

Figure 9–4 Bit Clear Mode



In Figure 9–5, the writing index of circle 2 or 010_2 is logically .AND.ed with the pixel values of the existing circle 001_2 and the background 000_2 to produce the pixel value 000_2. The appropriate bit plane settings are now changed such that all of circle 2, including the area of intersection with circle 1, matches the background.

Figure 9-5 Bit Clear Negate Mode



In Figure 9-6, the binary bitwise complement of the writing index of the circle 2 is 101_2. It is logically .OR.ed with the pixel values of the existing graphic object and background, which are 001_2 and 000_2. In bit set negate mode, the appropriate bit plane settings are now changed such that all of circle 2 is drawn in writing index 5.

Figure 9-6 Bit Set Negate Mode

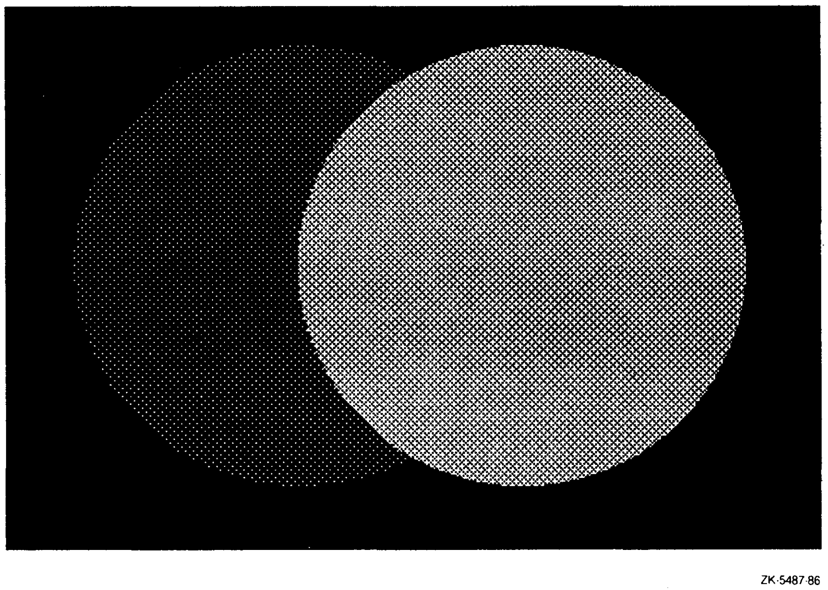
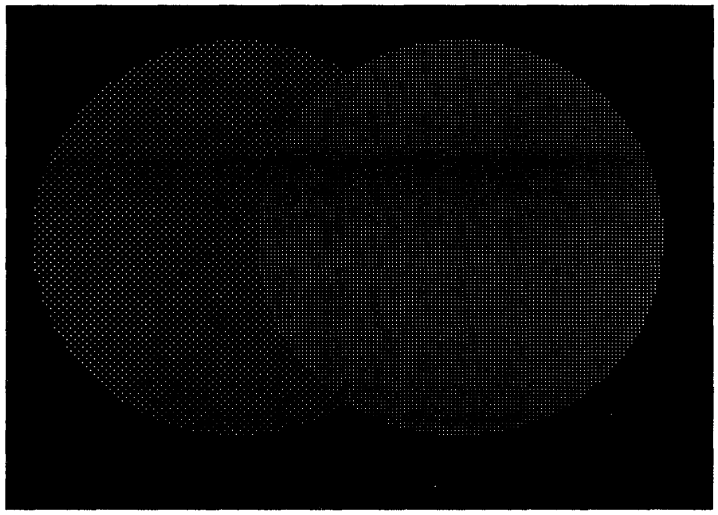


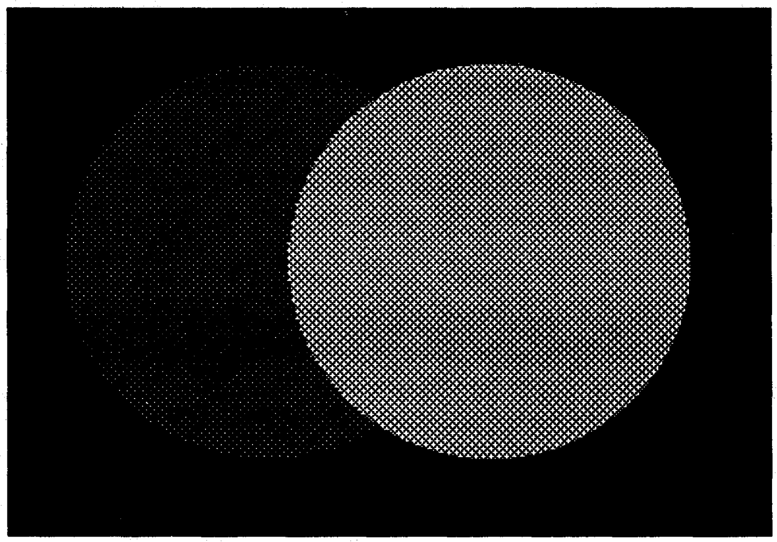
Figure 9–7 Copy Mode



In Figure 9–7, the writing index of circle 2 is used as the index in the virtual color map to draw the circle, regardless of existing graphic objects or background.

In Figure 9–8, the binary bitwise complement of the writing index of circle 2 101_2 is used as the index into the virtual color map to draw the circle regardless of existing graphic objects or background.

Figure 9–8 Copy Negate Mode



10 Text Attributes

10.1 Overview

UIS draws characters in the virtual display according to font specifications. The appearance or shape of characters remains unaltered until you change a text attribute. Likewise, UIS draws characters and character strings at user-specified locations within the coordinate space. This orientation within the coordinate space does not change until you execute an attribute modification routine.

Character and character string shape orientation spacially define how UIS draws these objects on the display screen. You can use text attribute modification routines to alter the appearance of characters and character strings or to redefine the spatial relationship of a character to other characters. This chapter discusses the following topics:

- Structure of text
- Text attributes
- Default text attribute settings

10.2 Structure of Text

The underlying structure of a single character is a character cell. Every character drawn on the display screen is contained in a character cell. Figure 10-1 illustrates a character cell and its reference points.

10.2.1 Monospaced and Proportionally Spaced Fonts

For text drawing purposes, fonts are either monospaced or proportionally spaced. Monospaced fonts use a standard character cell size for each character in the font. The character cells of proportionally spaced fonts vary in width for each font character, although the height of each cell is the same for each font character. Figure 10-2 shows the two types of fonts.

The character cell is a bitmap whose settings are mapped to the display screen as a character.

Figure 10–1 Character Cell

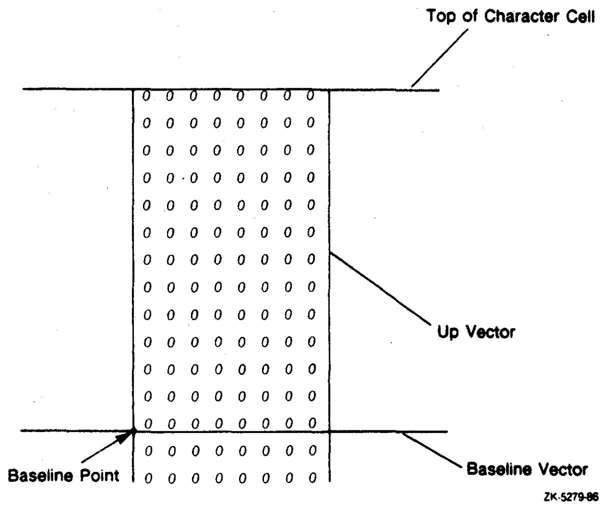
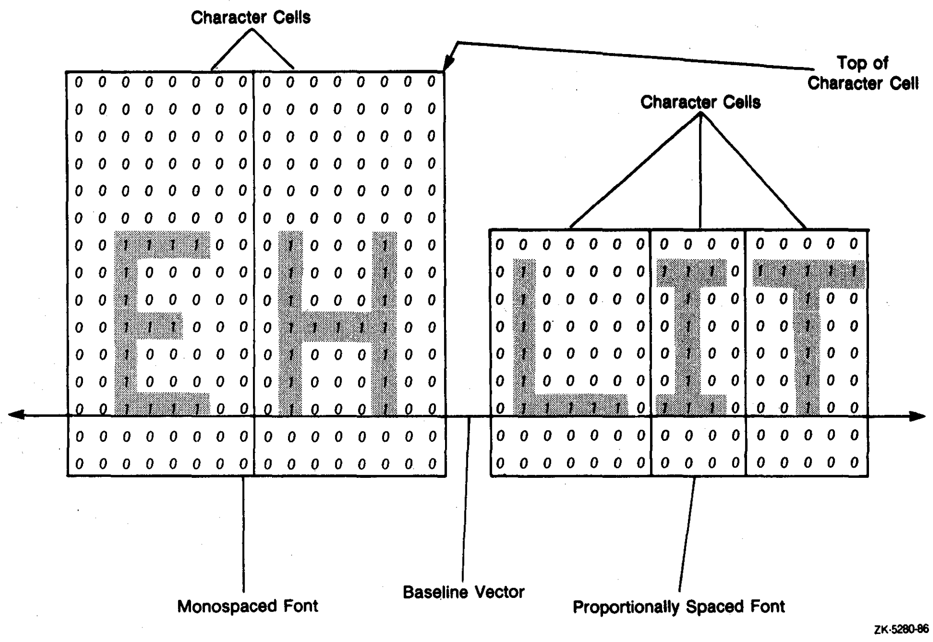


Figure 10–2 Monospaced and Proportionally Spaced Characters

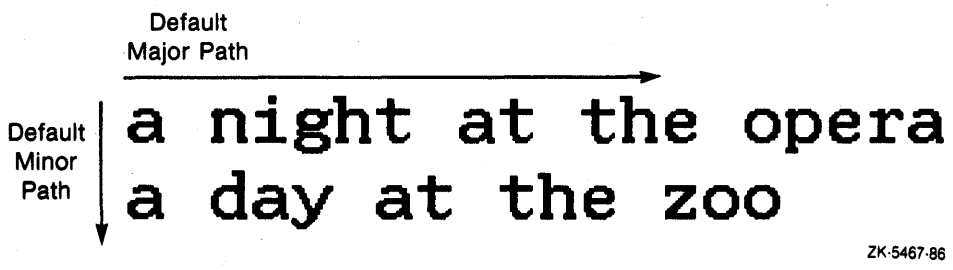


10.2.2 Lines of Text

Lines of text (for example, within a paragraph) share a spatial relationship with other lines of text. Ordinarily, you read lines of English text from left to right. Your eyes trace an imaginary path across the page from the left margin to the right margin. When you reach the end of a line, you read the next line below the current one.

By default, UIS draws a line of text in this left-to-right direction called the *default major path*. To begin a new line of text, UIS performs a secondary downward movement called the *default minor path*. This path is the normal relationship between lines of English text and the direction in which they are drawn. Figure 10-3 illustrates the two default paths that UIS uses to draw text.

Figure 10-3 Text Path



10.2.3 Character Strings

Characters within character strings share a spatial relationship with one another.

Text Slope

UIS draws all character string characters at the same angle to the major path. The *actual path* of text drawing is a line that contains the baseline points of all character cells in a character string. The angle between the actual path and the major path, measured counterclockwise, is called the angle of *text slope*. UIS can draw text at any angle from 0 to 360 degrees. Figure 10-4 shows how to manipulate text slope.

Text Margins

UIS draws character strings along the actual path of the text drawing within certain explicit or implicit boundaries called *margins*. The implied text margin for all text output is the minor text path when the angle of text slope is 0 degrees. The programming interface lets you set explicit text margins that are always parallel to the implied margins.

Character Spacing

Use *x* and *y* spacing factors to increase space uniformly between characters and lines throughout the character string. The size of the characters remains constant, while space between them increases or decreases.

Figure 10-5 shows how text path affects character spacing.

Figure 10–4 Text Slope

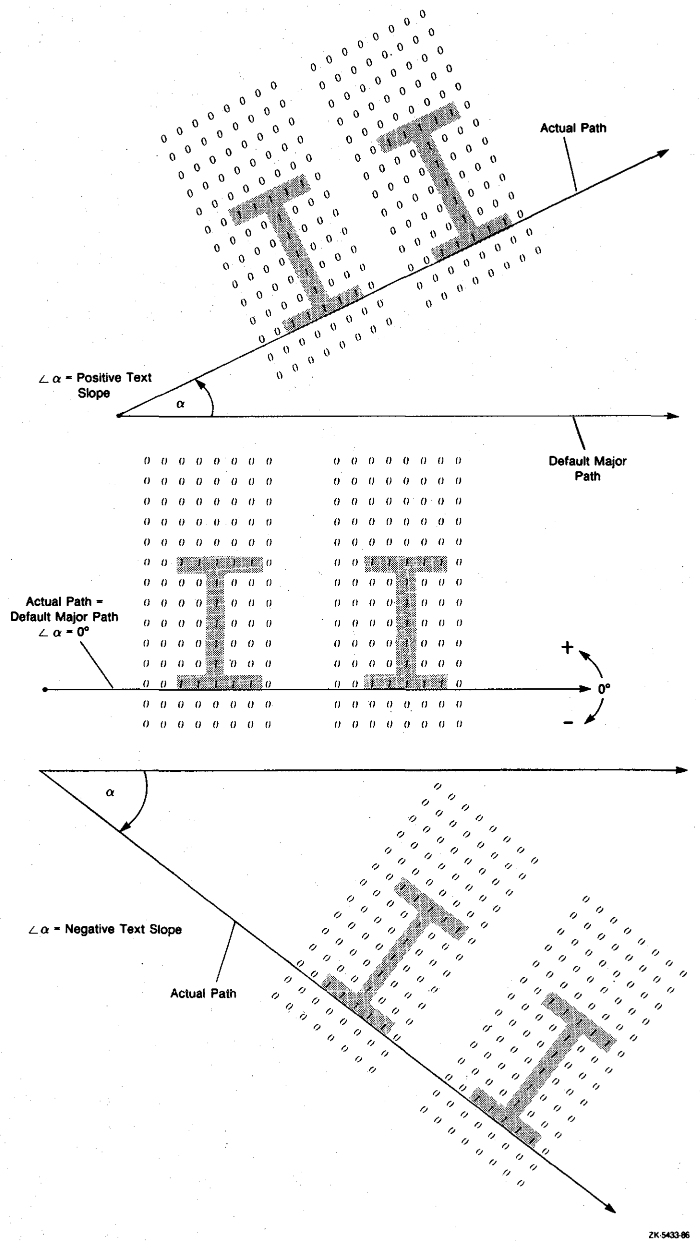

Actual Path
∠ α = Positive Text Slope
α
Default Major Path
Actual Path = Default Major Path ∠ α = 0°
+
0°
−
α
∠ α = Negative Text Slope
Actual Path
ZK-5433-86

Figure 10–5 Character Spacing

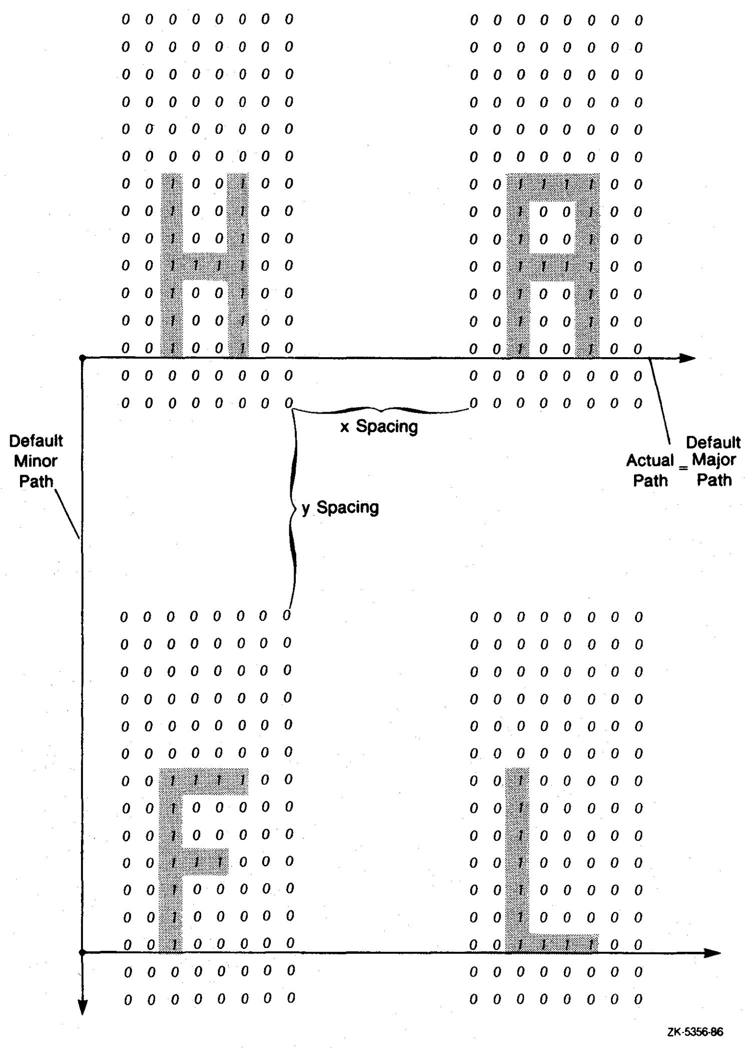
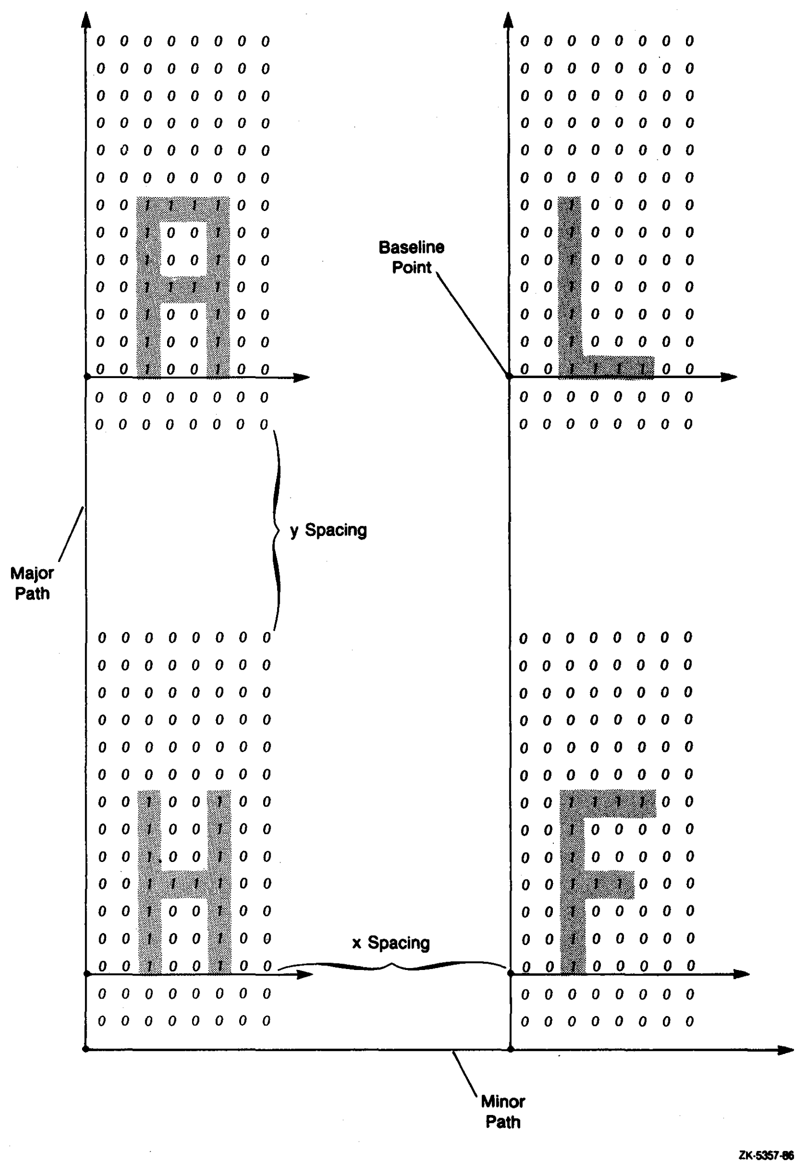


Figure 10–5 Cont'd. on next page

Figure 10-5 (Cont.) Character Spacing



Text Formatting

Use *justification* to arrange character strings on a line as follows:

- Flush against the left margin
- Flush against the right margin
- Centered between margins
- Both right and left justified (*fully justified*)

10.2.4 Character Cell

Character cell components share a spatial relationship with one another. You can change orientation and shape of a single character cell in the virtual display through character rotation, slanting, and scaling. When you modify these attributes, you alter the character cell with respect to its baseline vector. For example, if you modify the height of a scaled character, its height-relationship changes. The resulting letter might appear "squat" or vertically elongated.

Rotating Characters

You can rotate a single character around its baseline point. The angle of character rotation is the angle between the baseline vector and the actual path of text drawing, measured counterclockwise. Figure 10–6 shows simple character cell rotation.

Figure 10–7 shows simultaneous character rotation and text slope manipulation.

Figure 10-6 Simple Character Rotation

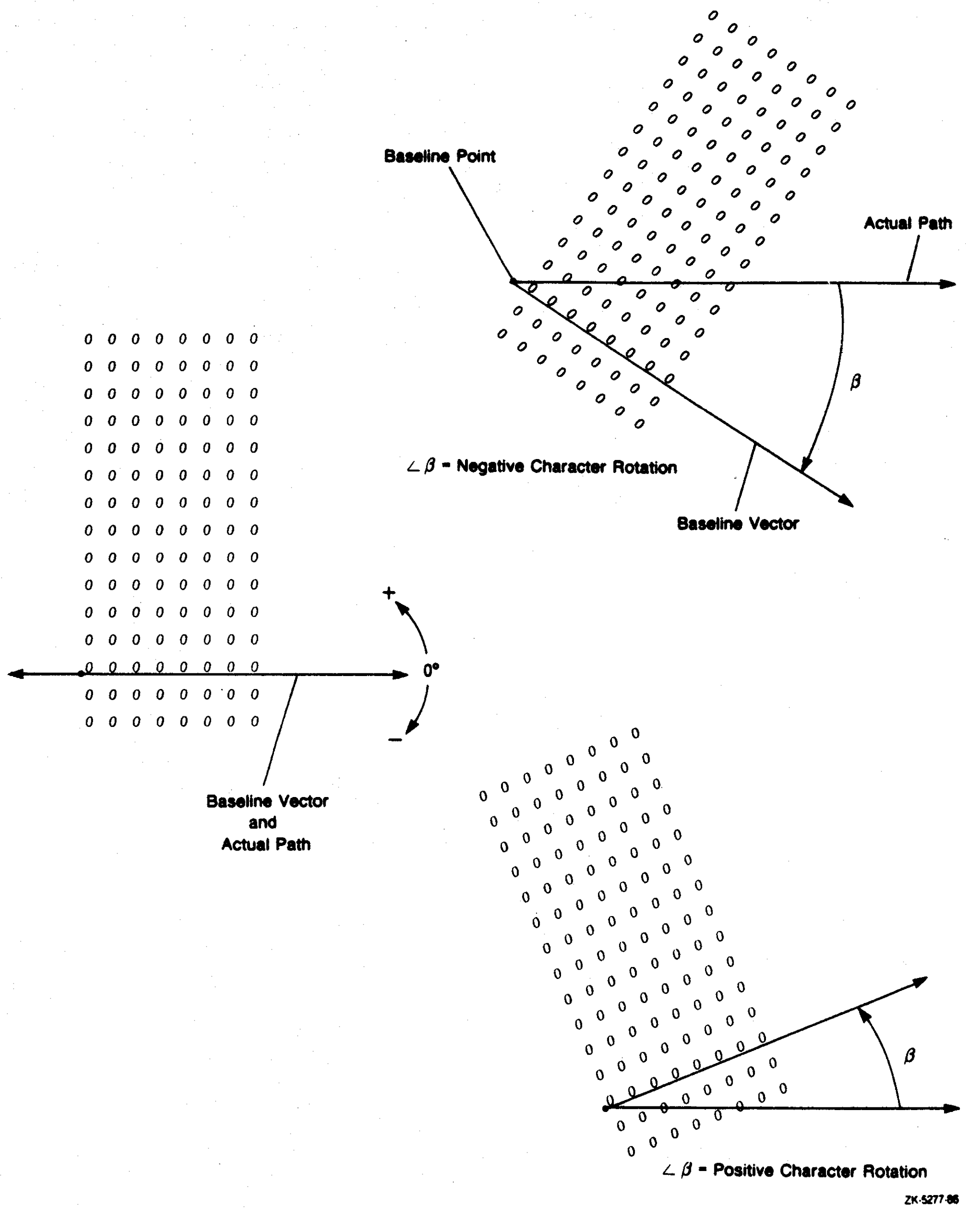

Baseline Point
Actual Path
β
∠ β = Negative Character Rotation
Baseline Vector
+
0°
−
Baseline Vector and Actual Path
β
∠ β = Positive Character Rotation
ZK-5277-86

Figure 10-7 Character Rotation with Slope Manipulation

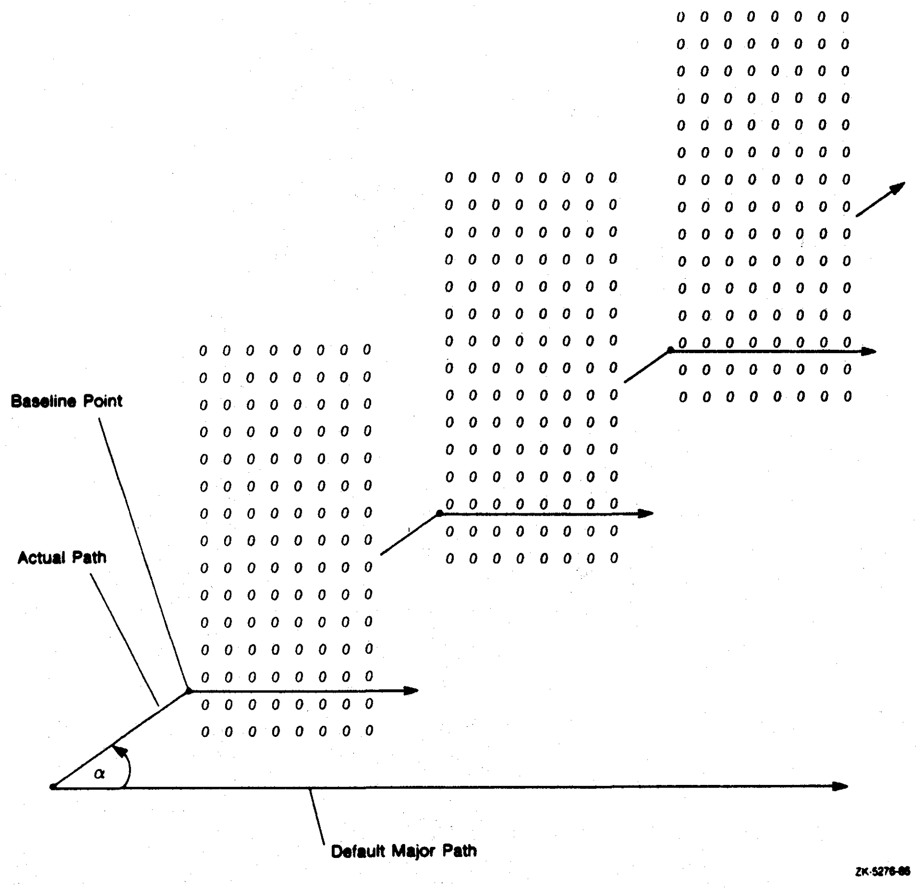
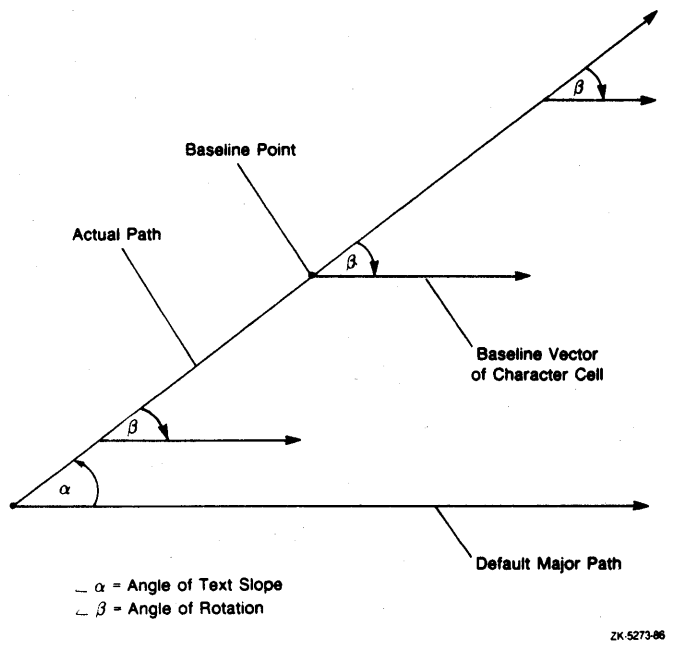


Figure 10-7 Cont'd. on next page

Figure 10-7 (Cont.) Character Rotation with Slope Manipulation



When you set the character rotation attribute to 0 and text slope is 0 degrees, the angle of character rotation behaves in the following manner.

Slope (degrees)	Major Path	Rotation (degrees)
0	Left to right (default)	0
0	Bottom to top	-90
0	Right to left	-180
0	Top to bottom	-270

Figure 10-8 illustrates the appearance of the angle of rotation after text path modification when default character rotation is in effect.

Figure 10-8 Text Path Manipulation Without Character Rotation

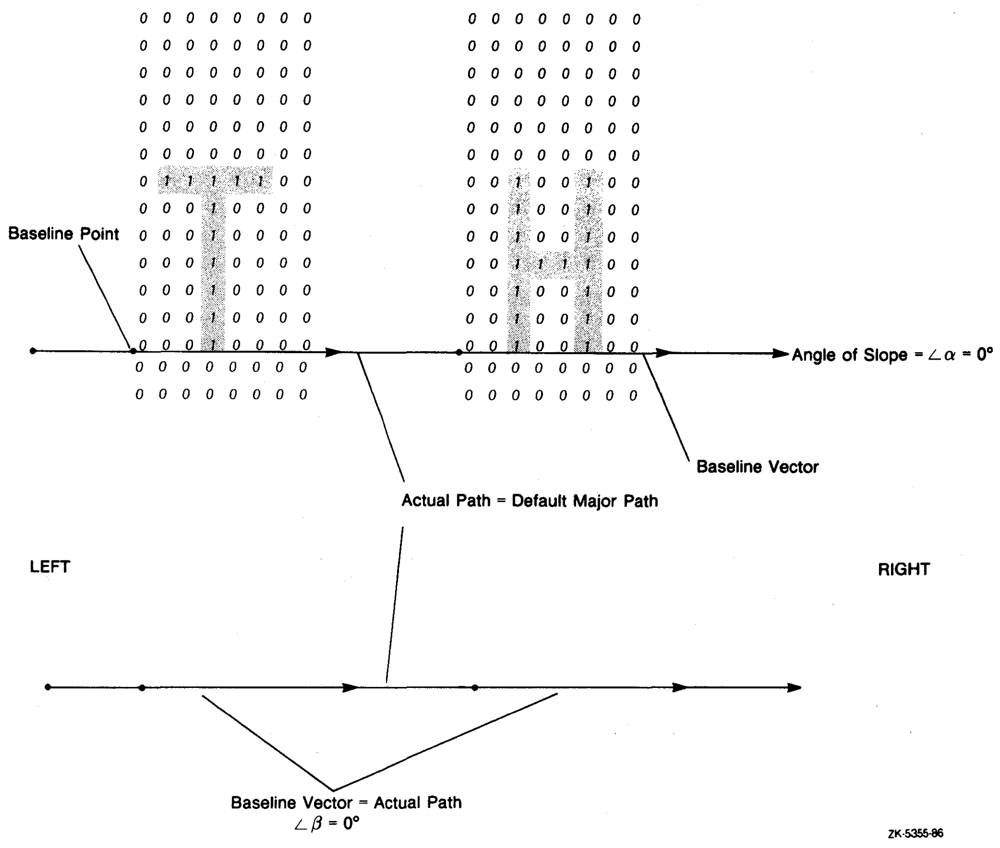


Figure 10-8 Cont'd. on next page

Figure 10-8 (Cont.) Text Path Manipulation Without Character Rotation

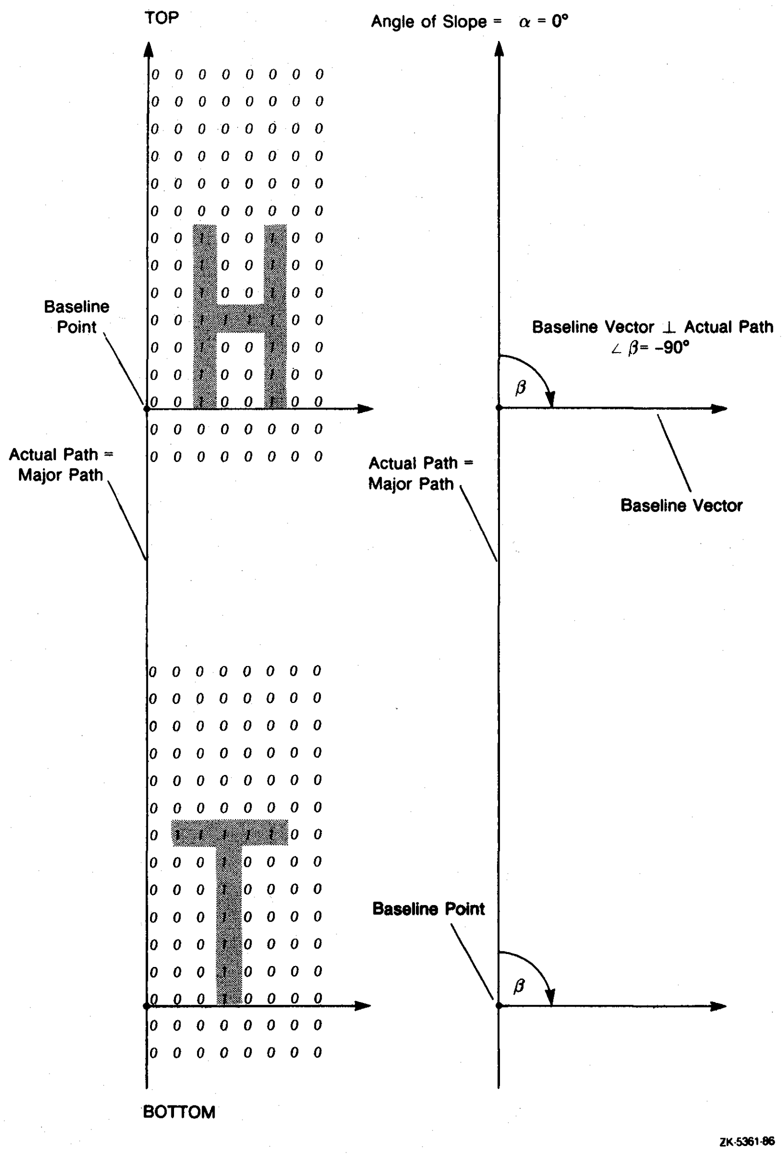


Figure 10-8 Cont'd. on next page

Figure 10-8 (Cont.) Text Path Manipulation Without Character Rotation

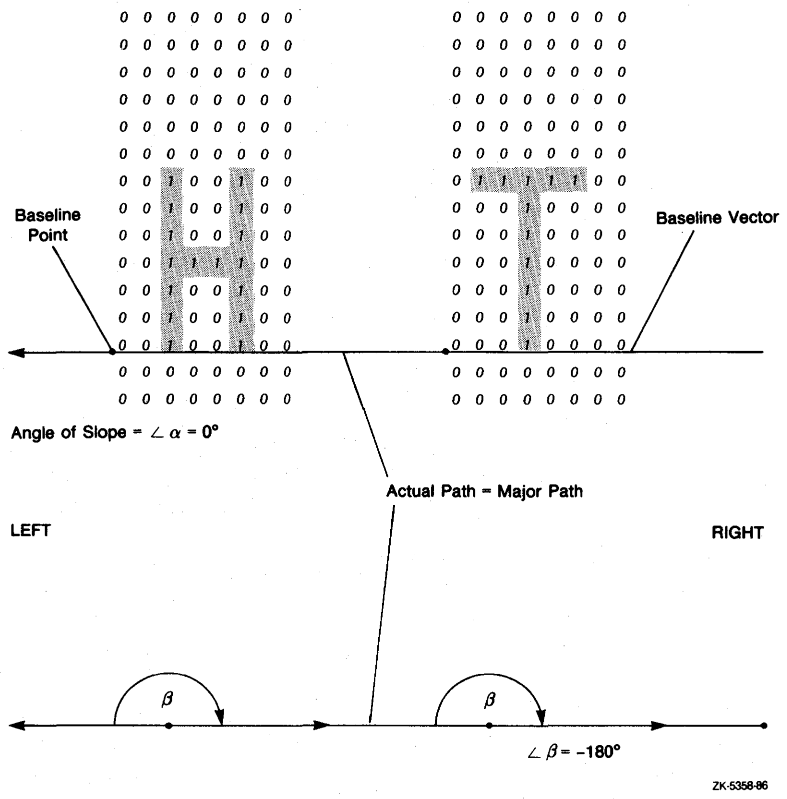
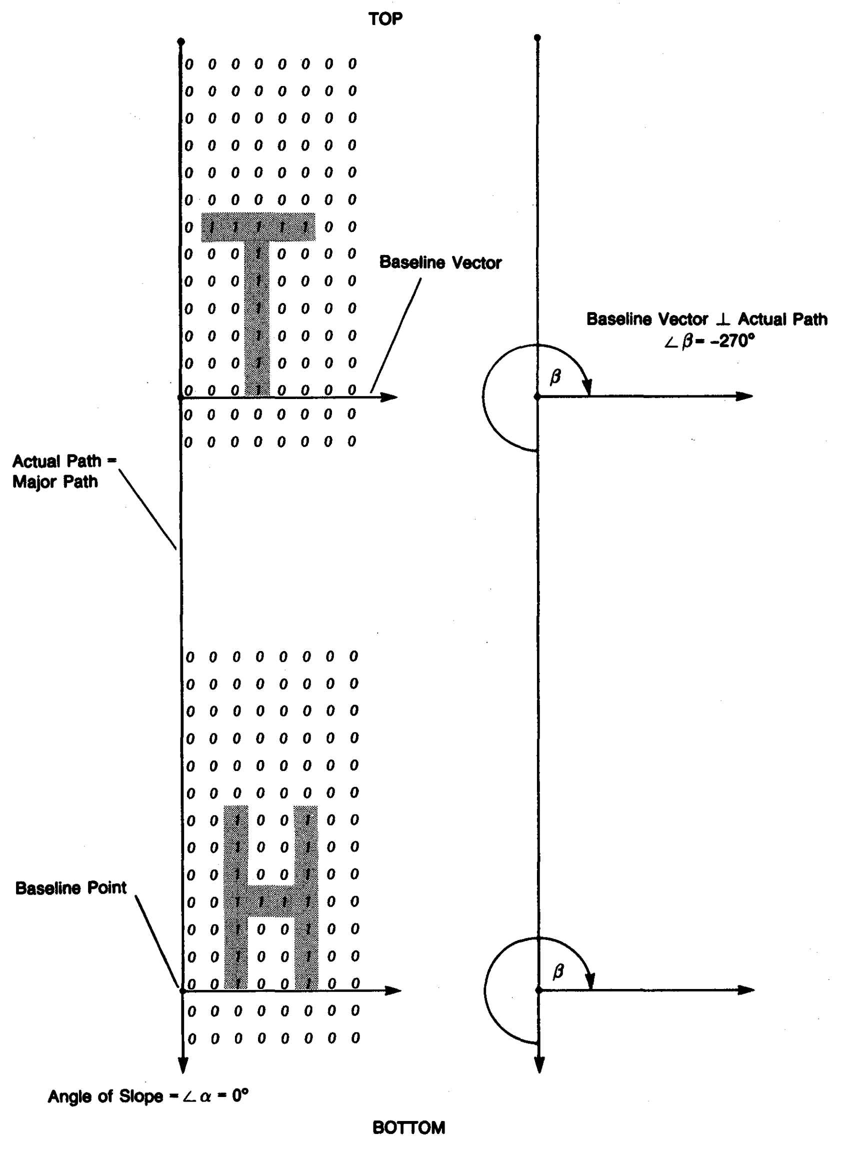


Figure 10-8 Cont'd. on next page

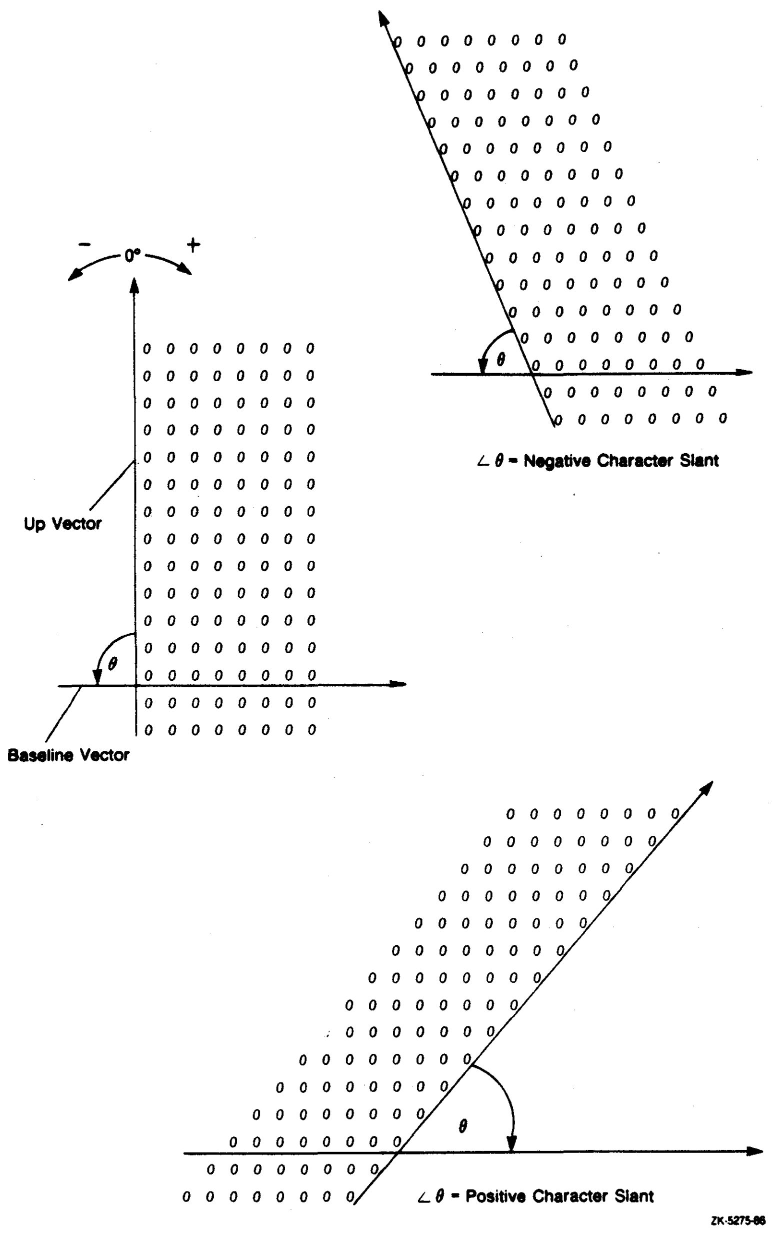
Figure 10-8 (Cont.) Text Path Manipulation Without Character Rotation



Slanting Characters

Character slant is a measure of the angle between the up vector and baseline vector of the character cell. Character slant is 0.0 when this angle is 90 degrees. As slant increases, the up vector rotates clockwise toward the baseline vector, until the two vectors coincide at a slant of 90 degrees. Figure 10-9 shows a slanted character cell where the actual path and the default major path form an angle of 0 degrees.

Figure 10-9 Character Slanting



−
0°
+
Up Vector
θ
Baseline Vector
θ
∠ θ = Negative Character Slant
θ
∠ θ = Positive Character Slant
ZK-5275-86

Figure 10–10 shows character slanting, character rotation, and text slope operations performed simultaneously on two character cells.

Figure 10–10 Character Slanting and Rotation with Slope Manipulation

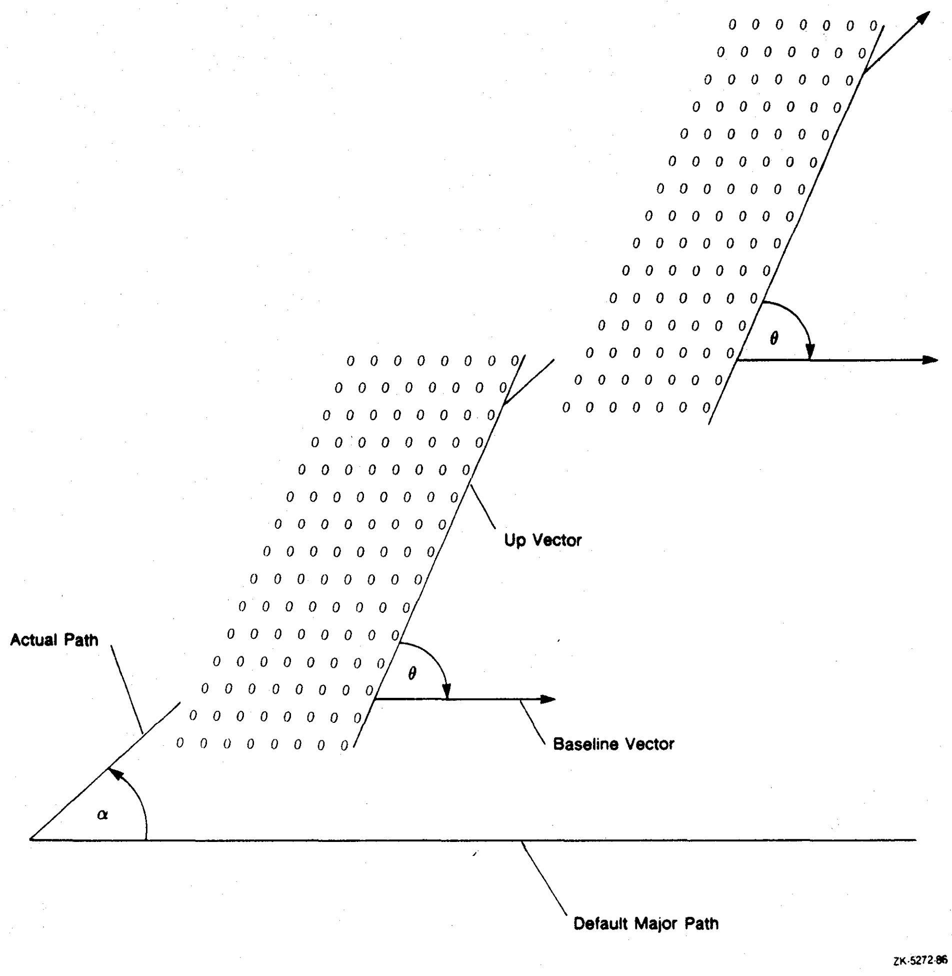
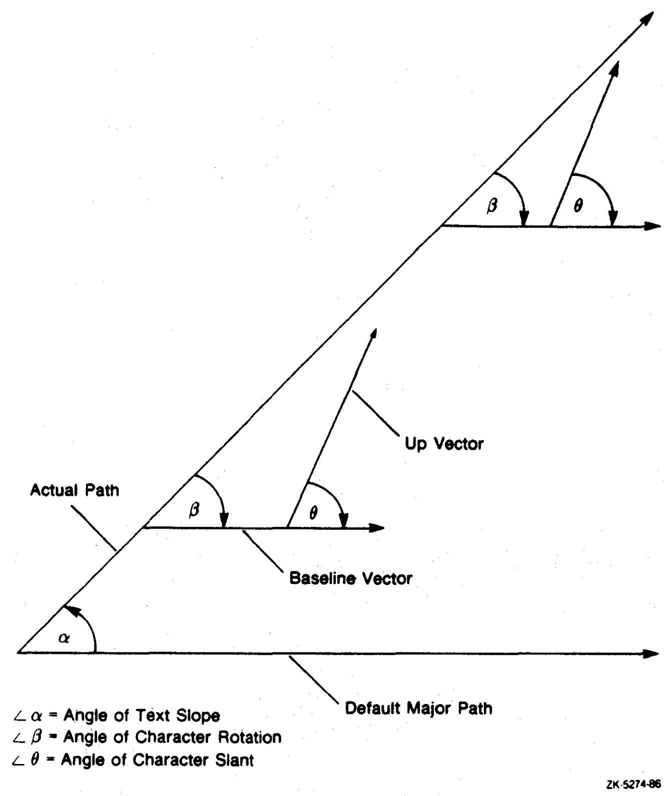


Figure 10–10 Cont'd. on next page

Figure 10-10 (Cont.) Character Slanting and Rotation with Slope Manipulation

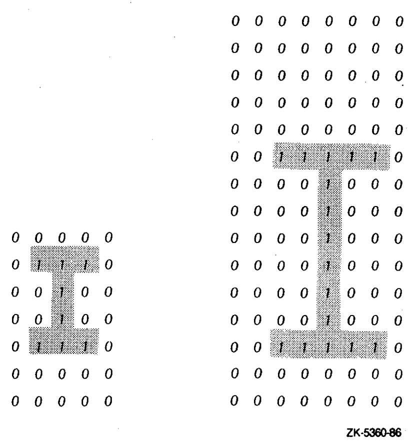

Up Vector
Actual Path
β
θ
Baseline Vector
α
Default Major Path
∠ α = Angle of Text Slope
∠ β = Angle of Character Rotation
∠ θ = Angle of Character Slant
ZK-5274-86

Scaling Characters

Character scaling involves increasing or decreasing the size of the character cell. Scaling factors specify the world coordinate space where the scaled character is drawn. The character cell is expanded or contracted to fit the specified space.

Figure 10-11 illustrates character scaling.

Figure 10–11 Character Scaling



10.3 Using Text Attributes

Several attributes are associated with text output. You can do more than simply choose from a library of fonts. For example, you can perform the following operations:

- Use scaling and slanting to modify the appearance of any font
- Use formatting modes and paths to change the way the system draws text in the virtual display

The following table lists routines that provide other types of text manipulation.

Routine	Function
UIS$NEW_TEXT_LINE	Moves the current text position along the minor text path
UIS$SET_ALIGNED_POSITION	Sets the current text position at the upper-left corner of the character cell
UIS$SET_POSITION	Sets the current text position at the baseline point of the character cell

These routines contain an **atb** argument, which indicates that appropriate text attribute settings can modify their behavior.

10.3.1 Modifying Text Attributes

When you modify text attributes, you do not change the default attribute settings within attribute block 0 itself. Think of attribute block 0 as a template of default settings; you modify a copy of this attribute block for use within your program. Attribute modification routines contain two arguments:

- **iatb**—Input attribute block number
- **oatb**—Output attribute block number

Table 10–1 lists all text attributes and their default settings.

Table 10–1 Default Settings of Text Attributes in Attribute Block 0

Text Attribute	Default Setting	Modification Routine
Character rotation	0.0	UIS$SET_CHAR_ROTATION
Character size	Specified by the font	UIS$SET_CHAR_SIZE
Character slant	0.0	UIS$SET_CHAR_SLANT
Character spacing	0.0,0.0	UIS$SET_CHAR_SPACING
Text formatting	Normal	UIS$SET_TEXT_FORMATTING
Text margins	0.0,0.0	UIS$SET_TEXT_MARGINS
Text path	Left to right (default major path) top to bottom (default minor path)	UIS$SET_TEXT_PATH
Text slope	0.0	UIS$SET_TEXT_SLOPE
Font	Multinational ASCII, 14-point, fixed pitch	UIS$SET_FONT

Modify attributes as follows:

1. Choose an appropriate attribute routine.
2. Specify *0* as the **iatb** argument to obtain a copy of attribute block 0.
3. Specify a number from *1* to *255* as the **oatb** argument. You can then refer to the attribute block in subsequent UIS graphics and text routines or in any other attribute modification routine.

The following routines reference modified attribute blocks in the **atb** argument.

- Graphics routines
- Text routines
- UIS$MEASURE_TEXT
- UIS$NEW_TEXT_LINE

These routines are discussed later in this chapter.

10.4 Programming Options

You can modify text attributes in your application to change font type, margin settings, and character spacing.

Fonts

Use UIS$SET_FONT to change the font type of a line of text. You must specify the desired font file name in the **font_id** argument. Font files reside in the directory SYS$FONT. The directory contains one file of fill patterns (UIS$FILL_PATTERNS) and 26 font files. You can choose between two types of fonts:

- Multinational character fonts — Contain international alphanumeric characters, including characters with diacritical marks.
- Technical fonts — Include scientific and mathematical symbols.

Font File Names

A standard 31-character file name identifies each font file as follows:

```
DTERMINM06OK00PG0001UZZZZ02A000
```

The following table defines the first 16 bytes of this sample file name, which represents unique font specifications.

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by Digital
2-7	Type family ID	TERMIN	Terminal
8	Spacing	M_{36}	13 pitch (monospaced)
9-11	Type size	$06O_{36}$	24 points (240 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	P	Bold
16	Proportion	G	Regular

Refer to Appendix Appendix C for more information about UIS fonts.

NOTE: **You can define logical names to represent font file names.**

Font File Types

The following table lists sample font file names and their device-dependent font file types.

System	Font File Name
	Mutinational Character Set Fonts
Monochrome	DTERMINM06OK00PG0001UZZZZ02A000.VWS$FONT
Intensity or color	DTERMINM06OK00PG0001UZZZZ02A000.VWS$VAFONT

System	Font File Name
	Technical Character Set Fonts
Monochrome	DVWSVT0G03CK00GG0001QZZZZ02A000.VWS$FONT
Intensity or color	DVWSVT0G03CK00GG0001QZZZZ02A000.VWS$VAFONT

NOTE: **Whenever you reference a font file name as in UIS$SET_FONT, do not specify the directory SYS$FONT or the file type.**

Setting the Text Margins

Use UIS$SET_TEXT_MARGINS to set the left and right margins.

Setting the Text Formatting Mode

Use UIS$SET_TEXT_FORMATTING to set the four text formatting modes—left justification, right justification, center justification, and full justification.

NOTE: **UIS$SET_TEXT_FORMATTING does not automatically wrap long lines of text.**

Setting the Character Spacing

Use UIS$SET_CHAR_SPACING to change the *kerning* (spacing between characters) or the *leading* (spacing between lines).

New Text Lines

Use UIS$NEW_TEXT_LINE to move to a new line. Use UIS$SET_CHAR_SPACING in conjunction with UIS$NEW_TEXT_LINE to manipulate the space between the old and the new line.

Character Rotation

Use UIS$SET_CHAR_ROTATION to rotate characters about a pivotal point (called the baseline point) from 0 to 360 degrees.

Aligning Text Along the Baseline and Top of Chararcter Cell

Use UIS$SET_POSITION to align text along the baseline vector; use UIS$SET_ALIGNED_POSITION to align text along the upper-left corner of the character cell.

Specifying Character Slant

Use UIS$SET_CHAR_SLANT to specify the angle relative to the text baseline vector by which text is to be slanted.

Specifying Character Scaling

Use UIS$SET_CHAR_SIZE to specify the width and height of characters in a font.

Specifying Slope of the Text Baseline

Use UIS$SET_TEXT_SLOPE to specify the angle of the actual path of text drawing relative to the major path.

Specifying the Text Path

Use UIS$SET_TEXT_PATH to specify the direction of text drawing. You can draw text in four directions:

- Left to right
- Right to left
- Bottom to top
- Top to bottom

You must use the direction in the context of a major text drawing path and a minor text drawing path. The major path of text drawing is the relationship between letters; the minor path is the relationship between lines.

10.4.1 Program Development I

Programming Objectives

To draw the multinational character set fonts available in the directory SYS$FONT and to show how to move to a new text line.

Programming Tasks

1. Create a virtual display.
2. Create a display window and viewport.
3. Modify the font attribute in attribute block 0.
4. Use UIS$NEW_TEXT_LINE and the appropriate attribute setting to move to the beginning of a new line.
5. Draw a line of text.
6. Repeat steps 3 through 5.

Note that program TEXT_1 uses logical names to represent font file names. Some actual font names occupy two lines.

```
PROGRAM TEXT_1
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
VD_ID=UIS$CREATE_DISPLAY(1.0,1.0,30.0,30.0,20.0,10.0)
WD_ID1=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','FONTS')

CALL UIS$SET_FONT(VD_ID,0,1,'MY_FONT_1')  1
CALL UIS$TEXT(VD_ID,1,'The quality of mercy is not strained',
2       1.0,30.0)  2
CALL UIS$SET_FONT(VD_ID,0,2,'MY_FONT_2')
CALL UIS$NEW_TEXT_LINE(VD_ID,2)  3
CALL UIS$TEXT(VD_ID,2,'Long visits bring short compliments')

CALL UIS$SET_FONT(VD_ID,0,3,'MY_FONT_3')
CALL UIS$NEW_TEXT_LINE(VD_ID,3)
CALL UIS$TEXT(VD_ID,3,'Wise men make proverbs and fools')
CALL UIS$NEW_TEXT_LINE(VD_ID,3)
CALL UIS$TEXT(VD_ID,3,'repeat them')
CALL UIS$SET_FONT(VD_ID,0,4,'MY_FONT_4')
CALL UIS$NEW_TEXT_LINE(VD_ID,4)
CALL UIS$TEXT(VD_ID,4,'Je pense donc je suis')
CALL UIS$SET_FONT(VD_ID,0,5,'MY_FONT_5')
CALL UIS$NEW_TEXT_LINE(VD_ID,5)
CALL UIS$TEXT(VD_ID,5,'Do well and have well')
CALL UIS$SET_FONT(VD_ID,0,6,'MY_FONT_6')
CALL UIS$NEW_TEXT_LINE(VD_ID,6)
CALL UIS$TEXT(VD_ID,6,'You cannot make a crab walk straight')

CALL UIS$SET_FONT(VD_ID,0,7,'MY_FONT_7')
CALL UIS$NEW_TEXT_LINE(VD_ID,7)
CALL UIS$TEXT(VD_ID,7,'Great minds think alike')

CALL UIS$SET_FONT(VD_ID,0,8,'MY_FONT_8')
CALL UIS$NEW_TEXT_LINE(VD_ID,8)
CALL UIS$TEXT(VD_ID,8,'One today is worth two tomorrows')
CALL UIS$SET_FONT(VD_ID,0,9,'MY_FONT_9')
CALL UIS$NEW_TEXT_LINE(VD_ID,9)
CALL UIS$TEXT(VD_ID,9,'With Latin, a horse, and money, you may')
CALL UIS$NEW_TEXT_LINE(VD_ID,9)
CALL UIS$TEXT(VD_ID,9,'travel the world')

CALL UIS$SET_FONT(VD_ID,0,10,'MY_FONT_10')
CALL UIS$NEW_TEXT_LINE(VD_ID,10)
CALL UIS$TEXT(VD_ID,10,'Whispered words are heard afar')
CALL UIS$SET_FONT(VD_ID,0,11,'MY_FONT_11')
CALL UIS$NEW_TEXT_LINE(VD_ID,11)
CALL UIS$TEXT(VD_ID,11,'Et tu, Brute?')
CALL UIS$NEW_TEXT_LINE(VD_ID,11)
CALL UIS$TEXT(VD_ID,11,'Per ardua astra')

CALL UIS$SET_FONT(VD_ID,0,12,'MY_FONT_12')
CALL UIS$NEW_TEXT_LINE(VD_ID,12)
CALL UIS$TEXT(VD_ID,12,'Velut arbor aevo')

CALL UIS$SET_FONT(VD_ID,0,13,'MY_FONT_13')
CALL UIS$NEW_TEXT_LINE(VD_ID,13)
CALL UIS$TEXT(VD_ID,13,'One mule scrubs another')
CALL UIS$SET_FONT(VD_ID,0,14,'MY_FONT_14')
CALL UIS$NEW_TEXT_LINE(VD_ID,14)
CALL UIS$TEXT(VD_ID,14,'Life is just a bowl of cherries')

PAUSE

END
```

The font attribute in attribute block 0 is modified in 14 calls to UIS$SET_FONT ❶. An attribute block with a modified font attribute for each font in SYS$FONT now exists. Each attribute block is identified by its creation-time output attribute block number.

The **atb** argument of UIS$TEXT ❷ uses the appropriate attribute block number to generate text in the desired font.

A call to UIS$NEW_TEXT_LINE ❸ causes each new line of text to begin on a new line at the left margin.

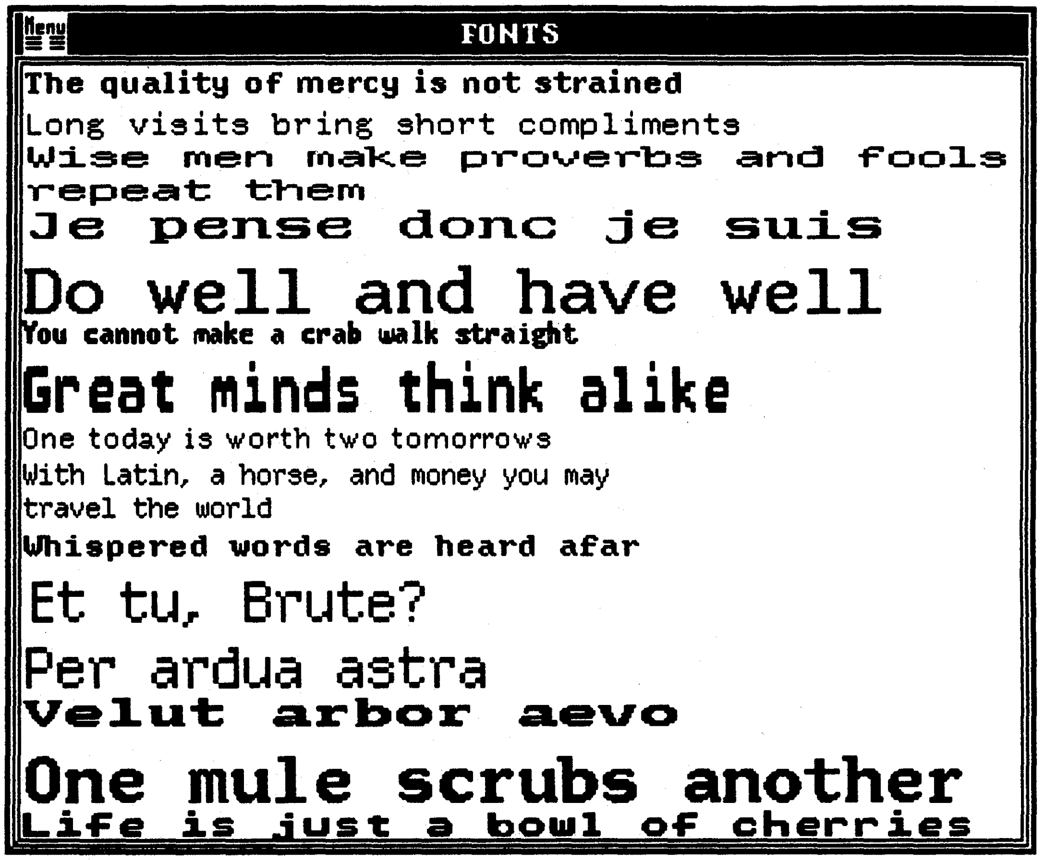
NOTE: **Before you run the demonstration programs, you must assign a logical name to the font used in the demonstration program. To do this, invoke the indirect command file SYS$EXAMPLES:DEFFONT.COM.**

10.4.2 Calling UIS$SET_FONT and UIS$NEW_TEXT_LINE

Note the positional order of the attribute routines. Attribute routines modify the attribute block used by the routine creating the graphic object and, therefore, must precede that routine. The attribute routine and the output routine must reference the same attribute block. Figure 10-12 contains examples of each UIS font.

Refer to Appendix Appendix C for a listing of UIS fonts.

Figure 10-12 UIS Fonts



The quality of mercy is not strained
Long visits bring short compliments
Wise men make proverbs and fools repeat them
Je pense donc je suis
Do well and have well
You cannot make a crab walk straight
Great minds think alike
One today is worth two tomorrows
With Latin, a horse, and money you may travel the world
Whispered words are heard afar
Et tu, Brute?
Per ardua astra
Velut arbor aevo
One mule scrubs another
Life is just a bowl of cherries

ZK-4546-85

10.4.3 Program Development II

Programming Objective

To increase character and line spacing in two lines of text.

Programming Tasks

1 Create a virtual display.

2 Create a display window and viewport with a title.

3 Use the default character spacing factor to draw a line of text.

4 Use UIS$SET_CHAR_SPACING to modify the character and line spacing factors.

5 Use the modified spacing attribute to draw a line of text.

6 Use UIS$NEW_TEXT_LINE with the modified spacing attribute to move to the beginning of a new line.

7 Repeat steps 3 through 5.

```
PROGRAM SPACE_1
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'

VD_ID=UIS$CREATE_DISPLAY(0.0,0.0,40.0,40.0,14.0,6.0)
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','KERNING AND LEADIN

CALL UIS$SET_FONT(VD_ID,0,15,'MY_FONT_1')  ❶

CALL UIS$TEXT(VD_ID,15,'The best mirror is an old friend',0.0,40.0)

CALL UIS$NEW_TEXT_LINE(VD_ID,15)  ❸
CALL UIS$SET_CHAR_SPACING(VD_ID,15,16,3.0,3.0)  ❹
CALL UIS$TEXT(VD_ID,16,'The best mirror is an old friend')  ❺

CALL UIS$NEW_TEXT_LINE(VD_ID,16)  ❻
CALL UIS$TEXT(VD_ID,15,'In the coldest flint there is hot fire')

CALL UIS$NEW_TEXT_LINE(VD_ID,15)
CALL UIS$TEXT(VD_ID,16,'In the coldest  flint there is hot fire')

PAUSE

END
```

A call to UIS$SET_FONT ❶ sets the font attribute. The attribute block containing the newly modified font attribute is assigned the number 15. The logical name MY_FONT_1 denotes a font used throughout the program.

The first line of text is drawn in the appropriate font ❷ at the virtual display location specified in UIS$TEXT.

When the next line of text is written, UIS$NEW_TEXT_LINE references attribute block 15 ❸. UIS$NEW_TEXT_LINE uses the new font characteristics to determine proper line spacing. If you use attribute block number 0, UIS$NEW_TEXT_LINE uses the characteristics of the default font. In that case, the descenders of letters in the previous line and the ascenders of the letters of the new line might crash into each other or obscure portions of letters in either line. Therefore, you should call UIS$NEW_TEXT_LINE using the appropriate attribute block number.

Attribute block 15 is further modified in a call to UIS$SET_CHAR_SPACING ❹. Now that attribute block 15 contains the previously modified font attribute and the newly modified character spacing attribute, it is assigned the number 16.

NOTE: Attribute block 15 still exists and can be referenced.

The character and line spacing attributes are set to a factor of 3. Characters are spaced by a factor of three times their width. Lines of text are spaced by a factor of three times the height of the character.

Text is drawn and spaced, character by character, according to values specified in the font attribute and the character spacing attribute in attribute block 16 5. The character spacing component of the character spacing attribute, or *x factor,* determines spacing between characters for left-to-right and right-to-left text paths.

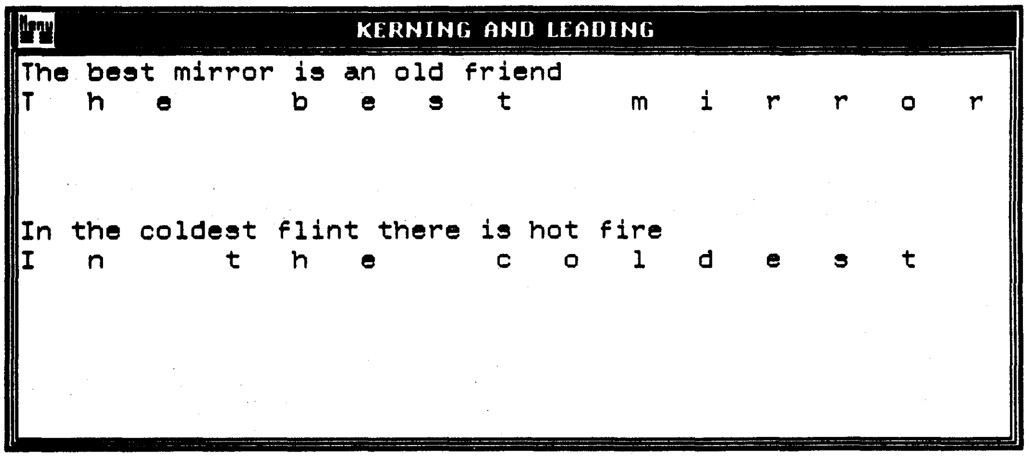
A call to UIS$NEW_TEXT_LINE 6 creates a new text line using attribute block 16. UIS$NEW_TEXT_LINE uses the line spacing component of the character spacing attribute, or *y factor,* to determine spacing between lines. The *y* factor is used for top-to-bottom and bottom-to-top text paths.

10.4.4 Calling UIS$SET_CHAR_SPACING

Call UIS$SET_CHAR_SPACING as shown here to set character spacing in one line of the previous example.

UIS$SET_CHAR_SPACING specifies a spacing factor of 3. If you run this program with the changes described above, your workstation screen will display the graphic objects shown in Figure 10–13.

Figure 10–13 Character and Line Spacing



```
The best mirror is an old friend
T  h  e     b  e  s  t     m  i  r  r  o  r



In the coldest flint there is hot fire
I  n     t  h  e     c  o  l  d  e  s  t
```

ZK-4547-85

The line now extends beyond the right margin of the display viewport.

10.4.5 Program Development III

Programming Objectives

To create alignment along the top of the character cell and along the baseline vector.

Programming Tasks

1 Create a virtual display.

2 Create a display window and viewport with title.

3 Draw a horizontal line the width of the viewport.

4 Use UIS$SET_ALIGNED_POSITION to set the current position for text output at the leftmost point on the line.

5 Choose a font and modify the font attribute block in attribute block 0.

6 Use the new font to draw a line of text.

7 Repeat step 4 using UIS$SET_POSITION.

8 Repeat steps 5 and 6.

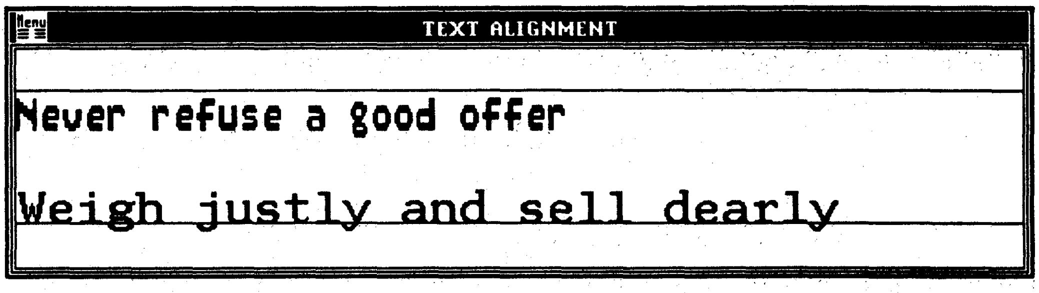
```
PROGRAM SET_POS
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'

VD_ID=UIS$CREATE_DISPLAY(0.0,0.0,40.0,40.0,18.0,5.0)
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','TEXT ALIGNMENT')

CALL UIS$PLOT(VD_ID,0,0.0,35.0,40.0,35.0)  ❶

CALL UIS$SET_FONT(VD_ID,0,1,'MY_FONT_7')
CALL UIS$SET_ALIGNED_POSITION(VD_ID,1,0.0,35.0)  ❷

CALL UIS$TEXT(VD_ID,1,'Never refuse a good offer')  ❸

CALL UIS$PLOT(VD_ID,0,0.0,20.0,40.0,20.0)  ❹

CALL UIS$SET_POSITION(VD_ID,0.0,20.0)  ❺
CALL UIS$SET_FONT(VD_ID,0,2,'MY_FONT_5')
CALL UIS$TEXT(VD_ID,2,'Weigh justly and sell dearly') ❻

PAUSE

END
```

Two horizontal and parallel lines are drawn with UIS$PLOT ❶ ❹.

Unless the current position is specified in UIS$TEXT, both calls to UIS$SET_ALIGNED_POSITION and UIS$SET_POSITION ❷ ❺ use the starting points of the respective lines to establish the current position for new text output.

Text creation ❸ ❻ begins by default at the current position established in UIS$SET_ALIGNED_POSITION and UIS$SET_POSITION.

Figure 10–14 Baseline and Top of Character Cell



Never refuse a good offer

Weigh justly and sell dearly

ZK-4548-85

10.4.6 Calling UIS$SET_POSITION and UIS$SET_ALIGNED_POSITION

In Figure 10–14, the first sentence illustrates the alignment of text along the top of the character cell. The second sentence illustrates alignment on the baseline vector.

10.4.7 Program Development IV

Programming Objective

To draw characters at three different angles relative to the baseline vector.

Programming Tasks

1 Create a virtual display.

2 Create a display window and a viewport with a title.

3 Choose a font and modify the font attribute in attribute block 0.

4 Draw a character string at the default angle 0 degrees.

5 Use UIS$SET_CHAR_SLANT to modify the character slant attribute.

6 Use the modified attribute and draw the character string again.

7 Repeat step 5 and specify negative degrees.

The file name MY_FONT_12 is a logical name for a font in SYS$FONT.

```
PROGRAM SLANT
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
VD_ID=UIS$CREATE_DISPLAY(0.0,0.0,20.0,5.0,18.0,4.5)
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','CHARACTER SLANTING

CALL UIS$SET_FONT(VD_ID,0,1,'MY_FONT_12')  ❶

CALL UIS$TEXT(VD_ID,1,'Unslanted characters do not lean',0.1,5.0) ❷
PAUSE

CALL UIS$SET_CHAR_SLANT(VD_ID,1,2,25.0)      ❸
CALL UIS$TEXT(VD_ID,2,'Slanted characters lean forward',0.5,3.0)

PAUSE
```

```
CALL UIS$SET_CHAR_SLANT(VD_ID,1,3,-25.0)     ❹
CALL UIS$TEXT(VD_ID,3,'Slanted characters lean backward',0.5,1.0)

PAUSE
END
```

A font is selected using UIS$SET_FONT ❶. A text string is drawn with the default attribute setting in attribute block 0 ❷.

Next, the character slant attribute is modified ❸ to specify a 25 degree shift to the right of a line perpendicular to the text baseline.

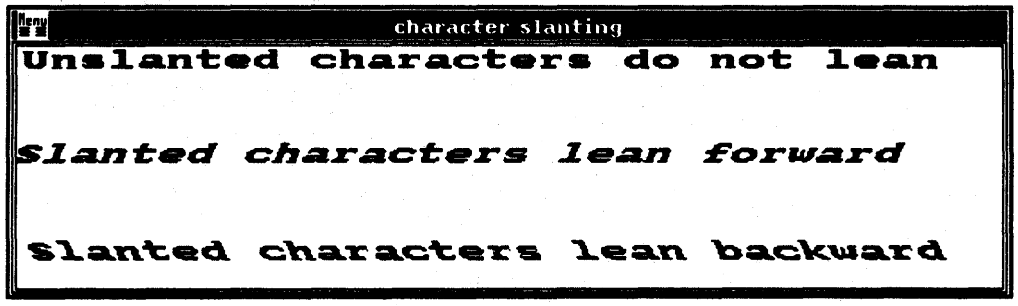
The character slant attribute is further modified ❹ to specify a 25 degree shift to the left of a line perpendicular to the text baseline.

10.4.8 Calling UIS$SET_CHAR_SLANT

First, the character string is drawn at the default slant—0 degrees. Next, the character string is drawn twice, first slanting each character 25 degrees to the right of a line perpendicular to the text baseline and then slanting each character 25 degrees to the left of that line.

Figure 10–15 shows character slanting.

Figure 10–15 Character Slanting



character slanting

Unslanted characters do not lean

Slanted characters lean forward

Slanted characters lean backward

10.4.9 Program Development V

Programming Objective

To draw a character string whose actual path increases at 20-degree increments from 0 to 340 degrees.

Programming Tasks

1 Create a virtual display.

2 Create a display window and viewport.

3 Create a DO loop that increases from 0 to 360 degrees by 20-degree increments, as follows:

- Place the slope attribute modification routine UIS$SET_TEXT_SLOPE within the DO loop.
- Place the text drawing routine UIS$TEXT within the DO loop.

The font file name MY_FONT_13 is a logical name for a font in SYS$FONT.

```
PROGRAM SLOPE
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
VD_ID=UIS$CREATE_DISPLAY(0.0,0.0,50.0,50.0,10.0,10.0)
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','text slope')

CALL UIS$SET_FONT(VD_ID,0,1,'MY_FONT_13')    1
DO I=0,340,20                            2
CALL UIS$SET_TEXT_SLOPE(VD_ID,1,2,FLOAT(I))       3
CALL UIS$TEXT(VD_ID,2,'   Slope!',25.0,25.0)     4
ENDDO                                        5

PAUSE

END
```

A font is selected and the default font attribute setting is modified with UIS$SET_FONT 1.

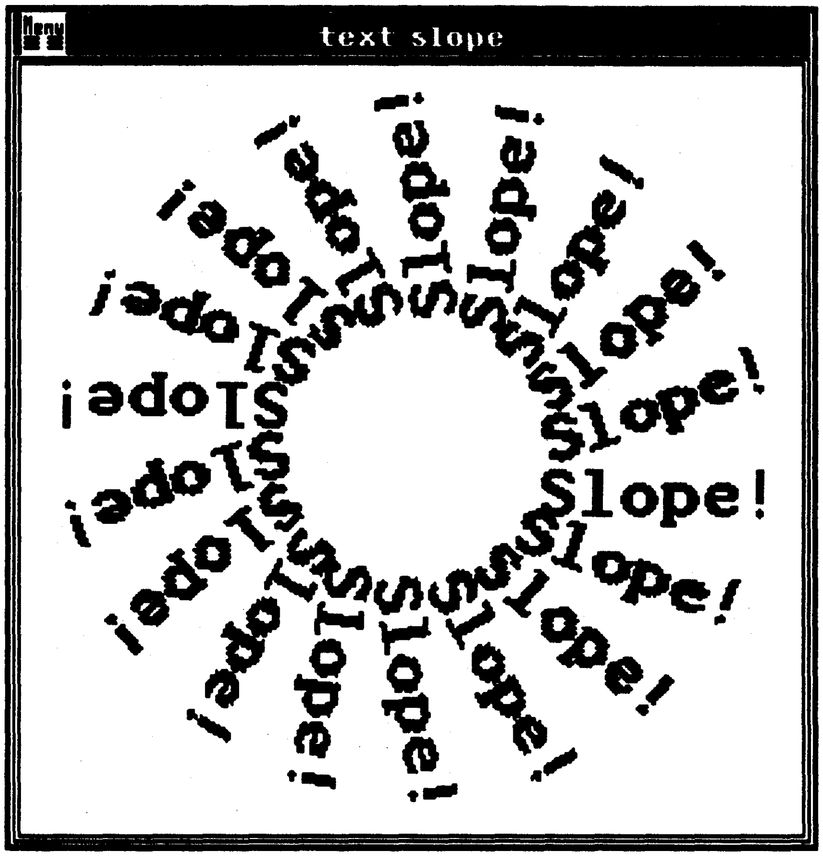
A DO loop is established 2 5. The counter *I* is initialized to 0 and increases by increments of 20. The **angle** argument in UIS$SET_TEXT_SLOPE uses the value of *I* as the new text baseline attribute setting 3. The VAX FORTRAN function FLOAT changes the integer counter *I* to a real number 3.

From UIS$TEXT, text strings are drawn from a central point (25.0,25.0) at 20-degree intervals 4.

10.4.10 Calling UIS$SET_TEXT_SLOPE

Text strings are drawn at 20-degree intervals from 0 degrees to 360 degrees. The angle of each new text baseline increases by a multiple of 20. Text is drawn in a counterclockwise direction from the default horizontal baseline.

Figure 10-16 Manipulating the Text Baseline



ZK-5422-86

10.4.11 Program Development VI

Programming Objective

To rotate each character to offset text slope.

Programming Tasks

1 Create a virtual display.

2 Create a display window and viewport.

3 Create a DO loop.

4 Modify the attributes within the DO loop.

```
PROGRAM SLOPE_ROTATE
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
```

```
VD_ID=UIS$CREATE_DISPLAY(0.0,0.0,50.0,51.0,10.0,10.0)
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION',
2       'TEXT SLOPE AND CHARACTER ROTATION')

CALL UIS$SET_FONT(VD_ID,0,1,'MY_FONT_13')

DO I=0,340,20
CALL UIS$SET_TEXT_SLOPE(VD_ID,1,2,FLOAT(I))   ❶
CALL UIS$SET_CHAR_ROTATION(VD_ID,2,2,FLOAT(-I))  ❷
CALL UIS$TEXT(VD_ID,2,'   Rotate!',24.0,28.5)
ENDDO

PAUSE

END
```

This program is identical to the previous program SLOPE, except that this program modifies the character rotation attribute as well as the text slope attribute.

Within the DO loop, both attribute modification calls use the value of the counter *I* to increase text slope angles and character rotation for different purposes ❶ ❷.

For every 20-degree increase text slope angle, the character rotation angle of each character must be decremented by 20 degrees. Consequently, each character baseline vector remains parallel to the default major path.

10.4.12 Calling UIS$SET_CHAR_ROTATION

The program SLOPE_ROTATE draws a series of character strings from a center point from 0 to 360 degrees at 20-degree intervals. Because the character rotation angle exactly offsets the text slope angle, characters maintain a readable orientation.

If you add a single call to modify the character slanting attribute, your viewport displays character rotation and slanting as the text slopes from 0 to 360 degrees at 20-degree intervals. Figure 10–18 illustrates this character rotation with slanting.

10.4.13 Program Development VII

Programming Objective

To manipulate the width and height of characters through scaling.

Programming Tasks

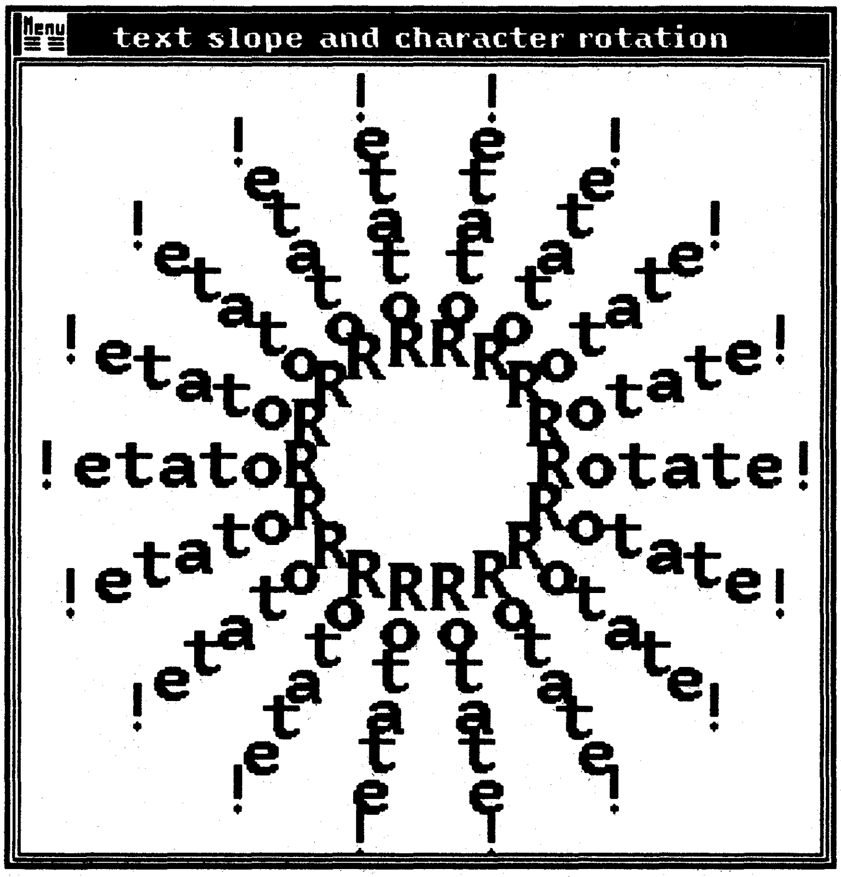
1 Create a virtual display.

2 Create a display window and viewport with title.

3 Draw a character string.

4 Increase the character size for width and height by 1.

Figure 10–17 Character Rotation Without Slanting

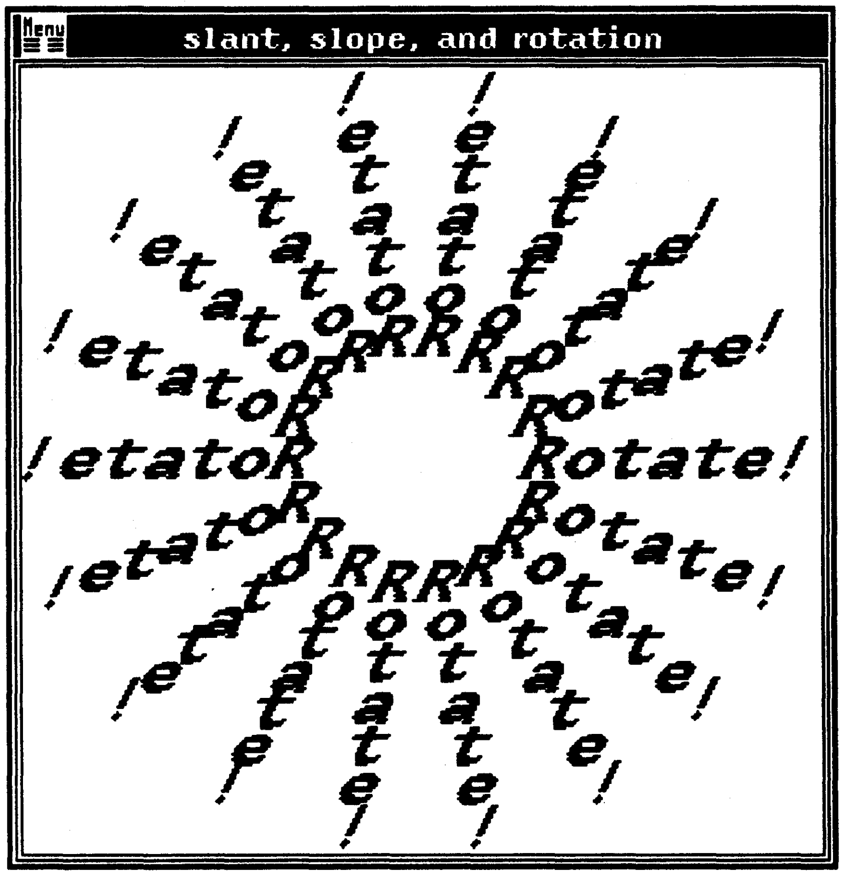


5 Repeat steps 3 and 4.

Font names used in this program are logical names.

```
PROGRAM CHARSIZE
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
REAL*4 WIDTH,HEIGHT
VD_ID=UIS$CREATE_DISPLAY(0.0,0.0,70.0,90.0,12.0,16.0)
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','CHARACTER SCALING'

CALL UIS$SET_FONT(VD_ID,0,1,'MY_FONT_1')  1
CALL UIS$TEXT(VD_ID,1,'Great scott!',0.0,90.0)       2
CALL UIS$SET_CHAR_SIZE(VD_ID,1,2,,2.0,2.0)     3
CALL UIS$TEXT(VD_ID,2,'Great scott!',0.0,80.0)     4
CALL UIS$SET_CHAR_SIZE(VD_ID,1,2,,3.0,3.0)
CALL UIS$TEXT(VD_ID,2,'Great scott!',0.0,70.0)
CALL UIS$SET_CHAR_SIZE(VD_ID,1,2,,4.0,4.0)
CALL UIS$TEXT(VD_ID,2,'Great scott!',0.0,60.0)
CALL UIS$SET_CHAR_SIZE(VD_ID,1,2,,5.0,5.0)
CALL UIS$TEXT(VD_ID,2,'Great scott!',0.0,50.0)
```

Figure 10–18 Character Rotation with Slanting



```
CALL UIS$SET_CHAR_SIZE(VD_ID,1,2,,6.0,6.0)
CALL UIS$TEXT(VD_ID,2,'Great scott!',0.0,40.0)
CALL UIS$SET_CHAR_SIZE(VD_ID,1,2,,7.0,7.0)
CALL UIS$TEXT(VD_ID,2,'Great scott!',0.0,30.0)
CALL UIS$SET_CHAR_SIZE(VD_ID,1,2,,8.0,8.0)
CALL UIS$TEXT(VD_ID,2,'Great scott!',0.0,20.0)
CALL UIS$SET_CHAR_SIZE(VD_ID,1,2,,9.0,9.0)
CALL UIS$TEXT(VD_ID,2,'Great scott!',0.0,10.0)
PAUSE
END
```

A font is selected [1].

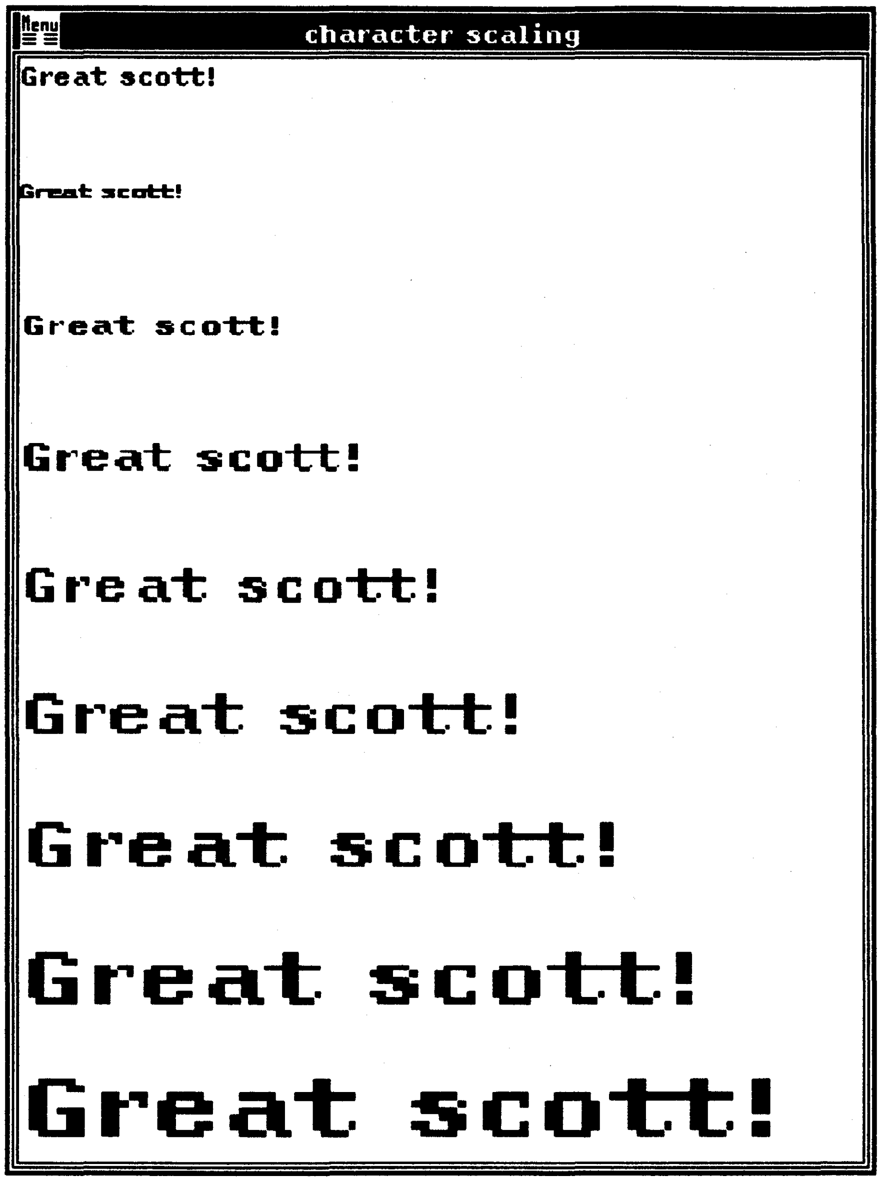
The unscaled character string Great scott! is drawn in the virtual display [2].

The character string is redrawn as scaled text. The scale factors for the width and height are incremented [3] each time the character string is drawn [4].

10.4.14 Calling UIS$SET_CHAR_SIZE

Figure 10-19 shows how the character string increases in height and width as the scale factors increment.

Figure 10-19 Manipulating Character Size



Great scott!

Great scott!

Great scott!

Great scott!

Great scott!

Great scott!

Great scott!

Great scott!

Great scott!

11 Graphics and Windowing Attributes

11.1 Overview

This chapter discusses the following topics:

- Creating dashed lines
- Creating lines of varying widths
- Using fill patterns
- Using clipping rectangles

11.2 Using Graphics Attributes

Graphics attributes affect arc type, line width, line style, and fill pattern use.

11.2.1 Modifying Graphics and Windowing Attributes

When you modify graphics and windowing attributes, you do not change the default attribute settings within attribute block 0 itself. Think of attribute block 0 as a template of default settings; you modify a copy of this attribute block for use within your program. Attribute modification routines contain two arguments:

- **iatb**—The input attribute block number
- **oatb**—The output attribute block number

Table 11-1 lists the default settings of graphics and windowing attributes.

Table 11-1 Default Settings of Graphics and Windowing Attributes

Attribute	Default Setting	Modification Routine
Arc type	Open	UIS$SET_ARC_TYPE
Fill pattern	Off	UIS$SET_FILL_PATTERN
Line style	Solid	UIS$SET_LINE_STYLE
Line width	1.0 (unscaled)	UIS$SET_LINE_WIDTH
Clipping rectangle	Off	UIS$SET_CLIP

Use the following procedure to modify attributes:

1 Choose an appropriate attribute routine.

2 Specify *0* as the **iatb** argument to obtain a copy of attribute block 0.

3 Specify a number from *1* to *255* as the **oatb** argument. You can then reference the attribute block in subsequent UIS graphics and text routines or in any other attribute modification routine.

Graphics and text routines reference modified attribute blocks in the **atb** argument and in these routines:

- UIS$MEASURE_TEXT
- UIS$NEW_TEXT_LINE
- UIS$SET_ALIGNED_POSITION

11.2.2 Programming Options

Depending on the graphic object you create—a line, a polygon, an ellipse, or a circle—you can choose from several attributes.

Fill Patterns

Fill patterns add shading to geometric figures on the workstation screen; you use them most often to accentuate portions of a pie graph. Fill patterns range in coloration:

- Light fill patterns—Represent light activity or minimum density.
- Heavy fill patterns—Represent heavy activity or maximum density.

To create your own fill pattern, select a character from any UIS font to serve as a fill pattern glyph.

All fill patterns are stored together in a font file in the directory SYS$FONT. For your convenience, this file name has been converted to the logical name UIS$FILL_PATTERNS.

Select a fill pattern as follows:

1 Using UIS$SET_FONT, specify *0* to select a copy of attribute block 0 to modify or specify the number of a previously modified attribute block as the input attribute block.

2 Assign an output attribute block number to the newly modified attribute block in UIS$SET_FONT. This number allows you to track attributes and to modify some other element in this attribute block later.

3 Specify the name of the fill pattern file in UIS$SET_FONT. Use the predefined logical name for the fill pattern file, UIS$FILL_PATTERNS.

   To use a character from a font other than the default fill pattern file as fill pattern glyph, specify the appropriate font name.

4 Use UIS$SET_FILL_PATTERN to specify the actual fill pattern with a UIS symbol in the argument **index**. A UIS symbol in the form PATT$C_*xxxx* exists for each fill pattern and serves as an index of each fill pattern in the file. The symbolic constant represents a hexadecimal offset that indicates the fill pattern position in the font file.

If you create a fill pattern from a UIS font other than the default fill pattern file, specify the ASCII code of the desired character in the index of UIS$SET_FILL_PATTERN.

NOTE: To disable fill patterns without modifying the fill pattern attribute, do not specify the index argument in UIS$SET_FILL_PATTERN.

Refer to 6.6 for more information about UIS constants.

Setting the Arc Type

If you want to draw a pie chart, you can draw chords or use UIS$SET_ARC_TYPE to request that no chord be drawn and specify one of the constants shown in the following table.

Arc Type	Description
UIS$C_ARC_OPEN	Does not draw any chords
UIS$C_ARC_PIE	Draws a line from both end points of the arc to the center position
UIS$C_ARC_CHORD	Draws a line connecting the end points of the arc

Remember that fill patterns are not drawn in the arc when the arc type attribute is specified as OPEN.

Line Width

Use UIS$SET_LINE_WIDTH to increase the apparent thickness of lines displayed on the screen. Note that this routine affects the thickness of lines created with the following routines only:

- UIS$LINE
- UIS$LINE_ARRAY
- UIS$PLOT
- UIS$PLOT_ARRAY
- UIS$ELLIPSE

Line Style

Occasionally, you need something other than a solid line. Use UIS$SET_LINE_STYLE to create dots, hyphens, and dashes.

11.2.2.1 Program Development I

Programming Objectives

To draw the different arc types and to demonstrate their use with fill patterns.

Programming Tasks

1. Create a virtual display.
2. Create a display window and viewport with title.
3. Use the chord arc type in attribute block 0 to modify the arc type attribute.

4 Use UIS$CIRCLE to draw an arc with the modified attribute block.

5 Repeat steps 3 and 4.

6 Erase the virtual display and delete the display window.

7 Create a display window and viewport with an identifying title.

8 Modify the arc type attribute. Select the pie arc type.

9 Select a fill pattern as follows:

- Modify the font attribute in attribute block 0.
- Modify the fill pattern attribute block 0.

10 Use the modified arc type, font, and fill pattern attribute blocks to draw an arc.

```
PROGRAM ARC
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'

VD_ID=UIS$CREATE_DISPLAY(0.0,0.0,40.0,40.0,15.0,15.0)
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','CHORD AND PIE')

CALL UIS$SET_ARC_TYPE(VD_ID,0,6,UIS$C_ARC_CHORD)  1
CALL UIS$CIRCLE(VD_ID,6,5.0,20.0,15.0,0.0,150.0)

CALL UIS$SET_ARC_TYPE(VD_ID,0,1,UIS$C_ARC_PIE)  2
CALL UIS$CIRCLE(VD_ID,1,23.0,20.0,15.0,0.0,150.0)

PAUSE

CALL UIS$DELETE_WINDOW(WD_ID)  3
CALL UIS$ERASE(VD_ID)  4

PAUSE

WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','FILLED PIE')  5

CALL UIS$SET_ARC_TYPE(VD_ID,0,1,UIS$C_ARC_PIE)
CALL UIS$SET_FONT(VD_ID,1,2,'UIS$FILL_PATTERNS')
CALL UIS$SET_FILL_PATTERN(VD_ID,2,3,PATT$C_HORIZ2_6)  6
CALL UIS$CIRCLE(VD_ID,3,18.0,20.0,15.0,0.0,150.0)

PAUSE

END
```

The program ARC creates two arcs and specifies two ways of closing those arcs 1 2.

To change the window caption, delete the display window and its associated viewport 3. Because the second part of the program draws a new graphic object, erase existing graphic objects 4.

A new display window is created and its viewport has a new title. 5.

The new graphic object is another arc with a pie arc type that contains a fill pattern 6.

11.2.2.2 Calling UIS$SET_ARC_TYPE and Using Fill Patterns

Figure 11–1 shows two ways to close an arc.

The second part of the program ARC executes and the fill pattern is drawn in the pie as shown in Figure 11–2.

Figure 11–1 Closing an Arc

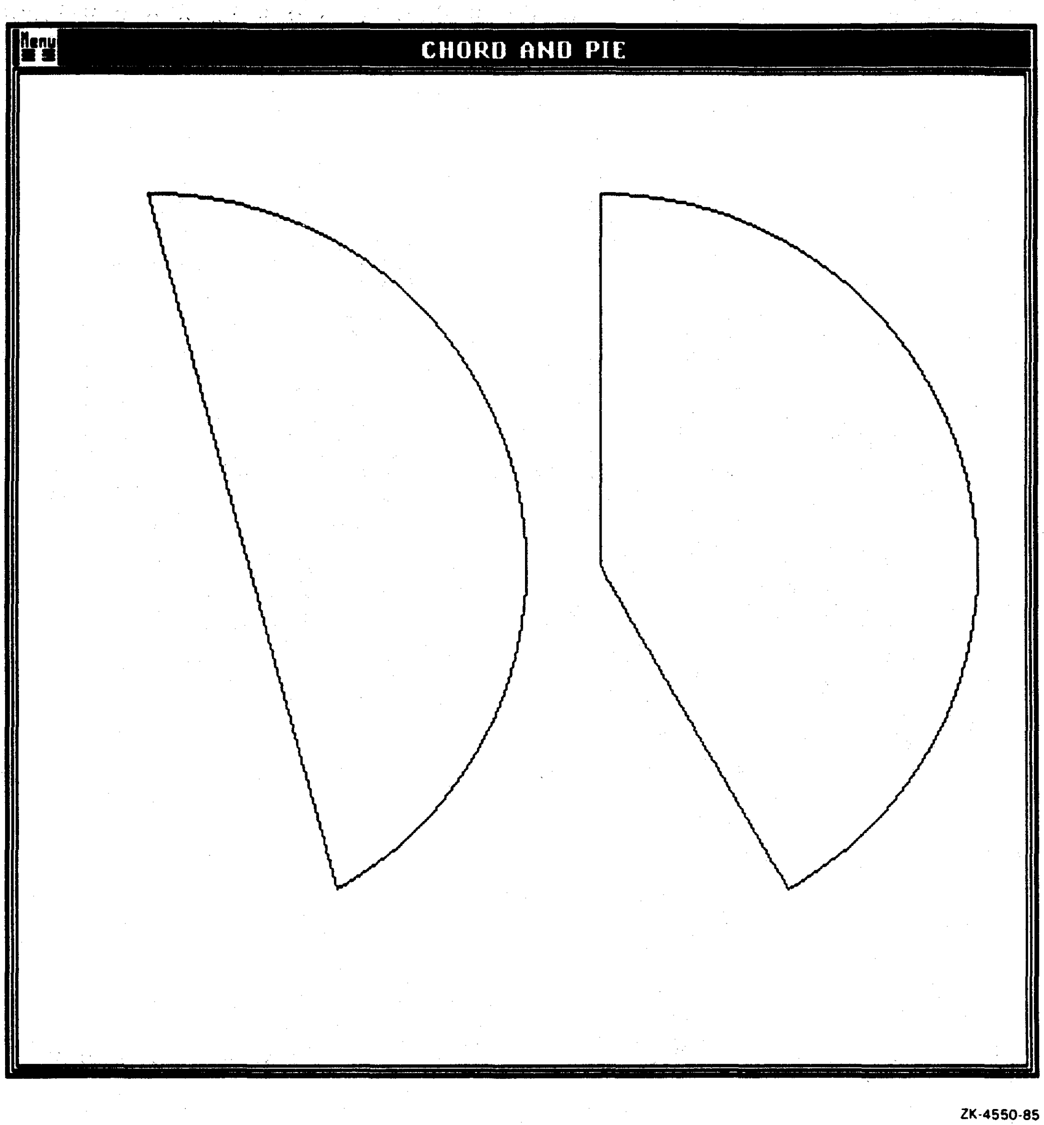
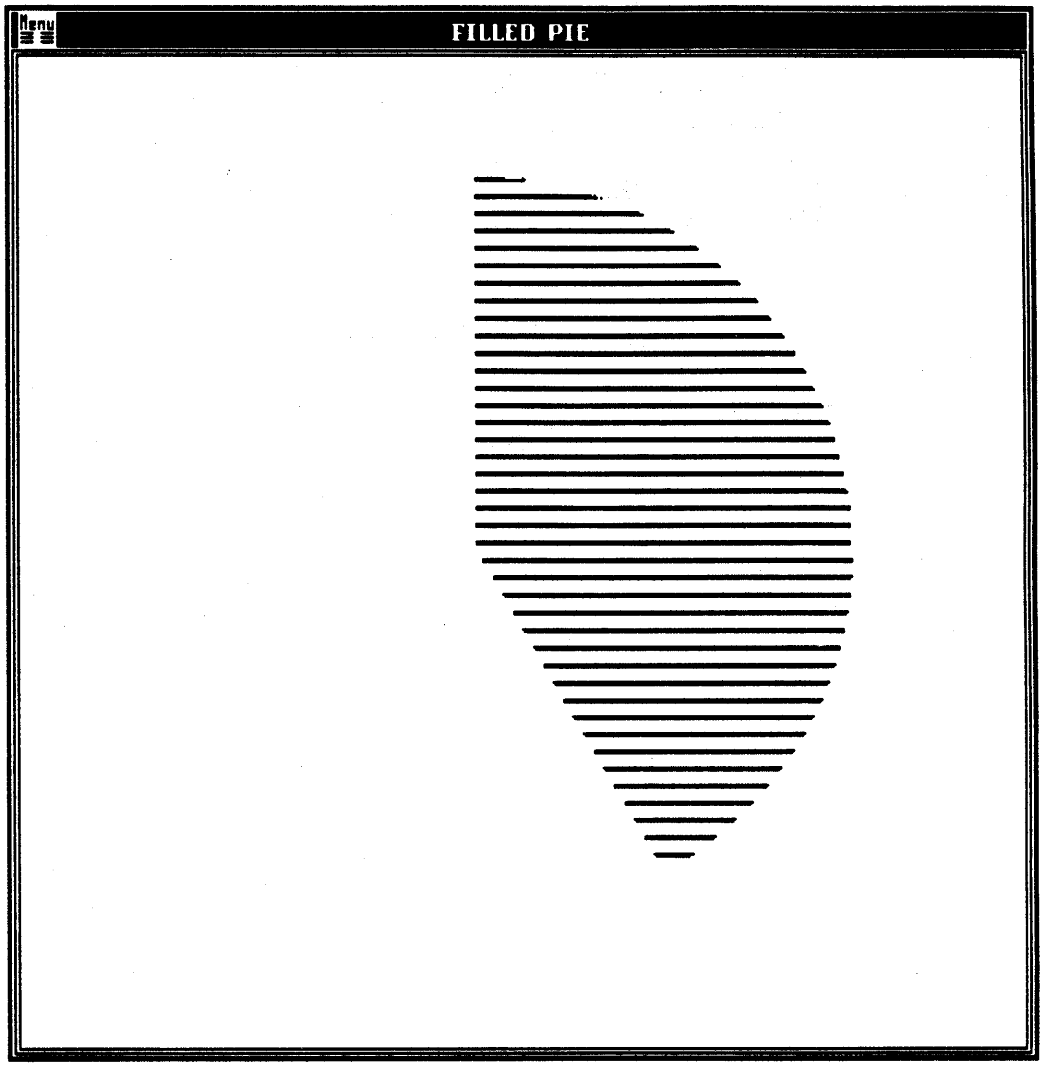


Figure 11–2 Filling a Closed Arc



ZK-4551-85

11.2.2.3 Program Development II

Programming Objective

To draw thickened lines.

Programming Tasks

1 Create a virtual display.

2 Create a display window and viewport with a title.

3 Draw two horizontal lines the width of the viewport—one near the bottom of the viewport and one near the top of the viewport.

4 Draw a vertical line connecting the horizontal lines.

5 Modify the line width attribute in attribute block 0 by a factor of 2.

6 Repeat steps 4 and 5.

```
PROGRAM LINE_WIDTH
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'

VD_ID=UIS$CREATE_DISPLAY(1.0,1.0,60.0,30.0,15.0,15.0)
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','LINE WIDTH')

CALL UIS$PLOT(VD_ID,0,1.0,25.0,60.0,25.0)   1
CALL UIS$PLOT(VD_ID,0,1.0,5.0,60.0,5.0)   2

CALL UIS$PLOT(VD_ID,0,5.0,5.0,5.0,25.0)   3

CALL UIS$SET_LINE_WIDTH(VD_ID,0,1,2.0)   4
CALL UIS$PLOT(VD_ID,1,10.0,5.0,10.0,25.0)   5

CALL UIS$SET_LINE_WIDTH(VD_ID,0,1,4.0)
CALL UIS$PLOT(VD_ID,1,15.0,5.0,15.0,25.0)

CALL UIS$SET_LINE_WIDTH(VD_ID,0,1,6.0)
CALL UIS$PLOT(VD_ID,1,20.0,5.0,20.0,25.0)

CALL UIS$SET_LINE_WIDTH(VD_ID,0,1,8.0)
CALL UIS$PLOT(VD_ID,1,25.0,5.0,25.0,25.0)

CALL UIS$SET_LINE_WIDTH(VD_ID,0,1,10.0)
CALL UIS$PLOT(VD_ID,1,30.0,5.0,30.0,25.0)

CALL UIS$SET_LINE_WIDTH(VD_ID,0,1,12.0)
CALL UIS$PLOT(VD_ID,1,35.0,5.0,35.0,25.0)

CALL UIS$SET_LINE_WIDTH(VD_ID,0,1,14.0)
CALL UIS$PLOT(VD_ID,1,40.0,5.0,40.0,25.0)

CALL UIS$SET_LINE_WIDTH(VD_ID,0,1,16.0)
CALL UIS$PLOT(VD_ID,1,45.0,5.0,45.0,25.0)

CALL UIS$SET_LINE_WIDTH(VD_ID,0,1,18.0)
CALL UIS$PLOT(VD_ID,1,50.0,5.0,50.0,25.0)

CALL UIS$SET_LINE_WIDTH(VD_ID,0,1,20.0)
CALL UIS$PLOT(VD_ID,1,55.0,5.0,55.0,25.0)

PAUSE

END
```

Two parallel lines are drawn with normal thickness the width of the display window with UIS$PLOT 1 2.

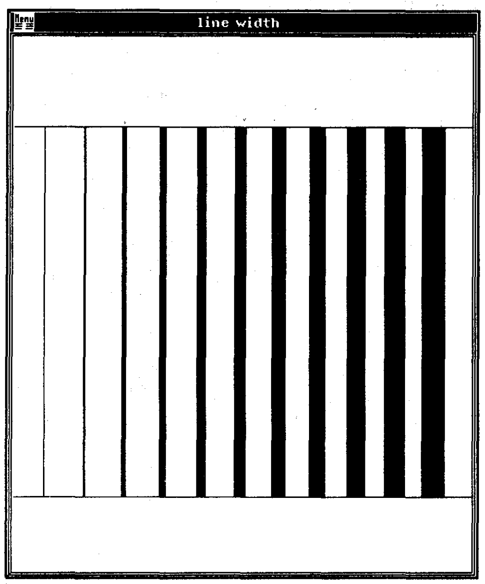
A vertical line of normal thickness is drawn 3.

Subsequent calls modify the line width attribute 4 and draw the resulting line 5 from the line in the lower half of the display window to the line in the upper half of the display screen.

11.2.2.4 Calling UIS$SET_LINE_WIDTH

Figure 11-3 shows lines drawn from point to point with increasing thickness.

Figure 11-3 Line Width



ZK 4626-85

NOTE: **Use UIS$PLOT or UIS$PLOT_ARRAY to draw extremely thick lines. Use UIS$SET_FILL_PATTERN to draw filled rectangles.**

11.2.2.5 Program Development III

Programming Objective

To draw various patterns of thickened dots and dashes.

Programming Tasks

1. Create a virtual display.
2. Create a display window and viewport with a title.
3. Modify the line width attribute to a thickness of 5 pixels.
4. Draw a solid thick line.
5. Modify the line style attribute.
6. Draw the dashed line.
7. Repeat steps 5 and 6.

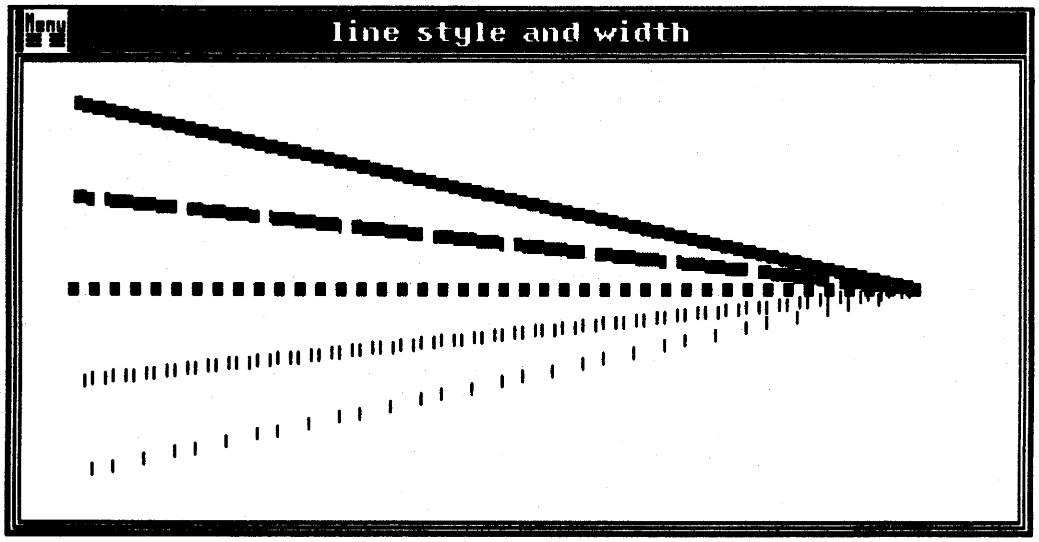
```
PROGRAM LINE_STYLE
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
VD_ID=UIS$CREATE_DISPLAY(0.0,0.0,20.0,20.0,15.0,6.0)
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','LINE STYLE AND WID
CALL UIS$SET_LINE_WIDTH(VD_ID,0,1,5.0)  1
CALL UIS$PLOT(VD_ID,1,1.0,18.0,18.0,10.0)

CALL UIS$SET_LINE_STYLE(VD_ID,1,1,'FFFFFFF0'X)   2
CALL UIS$PLOT(VD_ID,1,1.0,14.0,18.0,10.0)
CALL UIS$SET_LINE_STYLE(VD_ID,1,2,'F0F0F0F0'X)  3
CALL UIS$PLOT(VD_ID,2,1.0,10.0,18.0,10.0)

CALL UIS$SET_LINE_STYLE(VD_ID,2,3,'90909090'X)  4
CALL UIS$PLOT(VD_ID,3,1.0,6.0,18.0,10.0)

CALL UIS$SET_LINE_STYLE(VD_ID,3,4,'10010010'X)  5
CALL UIS$PLOT(VD_ID,4,1.0,2.0,18.0,10.0)

PAUSE

END
```

Create different line styles by selecting different hexadecimal values in the calls to UIS$SET_LINE_STYLE 1 2 3 4. The hexadecimal values set bits in the line style bit vector, which, in turn, generates a pattern.

11.2.2.6 Calling UIS$SET_LINE_WIDTH and UIS$SET_LINE_STYLE

When the program LINE_STYLE executes, five lines are drawn, each with the same width but different style. The pattern of dots and dashes is determined by the value supplied to the line style longword bit vector as shown in Figure 11–4.

Figure 11–4 Modifying Line Width and Style



ZK-4552-85

11.2.2.7 Program Development IV Programming Objective

To construct a vertical bar graph.

Programming Tasks

1. Load arrays from DATA statements.
2. Create a virtual display.
3. Create a display window and viewport with a title.
4. Draw the x and y axes.
5. Draw the legend.
6. Draw the information along the *x* axis.
7. Draw the information along the *y* axis.
8. Modify the font and fill pattern attributes.
9. Use the appropriate fill patterns with the arrays to draw vertical bars to their proper heights.

```
        PROGRAM GRAPH
        IMPLICIT INTEGER(A-Z)
        CHARACTER*4 STRING
        REAL ARRAY1(8),ARRAY2(8),X,X2,HEIGHT,Y  1
        DATA ARRAY1 /5.0,10.0,12.0,13.0,15.0,20.0,25.0,30.0/
        DATA ARRAY2 /0.0, 1.0, 2.0, 1.0, 4.0, 9.0,15.0,21.0/
        INCLUDE 'SYS$LIBRARY:UISENTRY'
        INCLUDE 'SYS$LIBRARY:UISUSRDEF'

        VD_ID=UIS$CREATE_DISPLAY(-5.0,-5.0,50.0,50.0,20.0,20.0)
        WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','GRAPH')

        CALL UIS$SET_LINE_WIDTH(VD_ID,0,16,5.0)  2
        CALL UIS$PLOT(VD_ID,16,0,0,0,35.0)  3
        CALL UIS$PLOT(VD_ID,16,0,0,45.0,0)  4

        CALL UIS$TEXT(VD_ID,0,'U.S. ADULT POPULATION VS. CAR OWNERSHIP',
        2       10.0,-3.0)  5
c       Information along the y axis

        DO 20 I = 1,7
        Y = 5.0 * FLOAT (I)  6
        N = 25 * I  7
        ENCODE (3,10,STRING) N    8
10      FORMAT (I3)
20      CALL UIS$TEXT(VD_ID,0,STRING,-3.0,Y)  9

        CALL UIS$TEXT(VD_ID,0,'(in millions)',-3.0,37.0)
c       Information along the x axis

        DO 40 I = 1,8
        Y = 5.0 * FLOAT (I)
        N = 1900 + (10 * I)
        ENCODE (4,30,string) N
30      FORMAT (I4)
40      CALL UIS$TEXT(VD_ID,0,string,Y,-1.0)

        CALL UIS$SET_FONT(VD_ID,0,1,'UIS$FILL_PATTERNS')  10
        CALL UIS$SET_FILL_PATTERN(VD_ID,1,1,PATT$C_HORIZ4_4)   11
        CALL UIS$SET_FILL_PATTERN(VD_ID,1,2,PATT$C_GREY12_16)  12
C PLOT POPULATION RECTANGLE

        DO 100 I = 1,8

        X = 5.0 * FLOAT(I)
        X2 = X + 2.0
        HEIGHT = ARRAY1(I)  13
        CALL UIS$PLOT (VD_ID,1, X,0.0, X,HEIGHT, X2,HEIGHT, X2,0.0)
C PLOT CAR RECTANGLE
```

```
          X = X + 1.0
          X2 = X + 2.0
          HEIGHT = ARRAY2(I)  ⓮
          CALL UIS$PLOT (VD_ID,2, X,0.0, X,HEIGHT, X2,HEIGHT, X2,0.0)
100       CONTINUE

          PAUSE
          END
```

Two arrays, ARRAY1 and ARRAY2, are declared ❶ to store the height of each vertical bar in the graph.

The *x* and *y* axes are drawn ❸ ❹. However, a previous call to UIS$SET_LINE_WIDTH ❷ modified the attribute block that controls line appearance. Line width (*x* and *y* axes) should be five times wider than normal.

A call to UIS$TEXT ❺ creates the graph legend.

The *y* world coordinate values are computed ❻ as multiples of 5, where *I* represents the number of passes through the DO loop. The adult population numbers are written at these intervals.

The numbers along the y axis are computed and stored in the variable *N* ❼, then returned to the variable *string* as character string constants ❽ ❾.

Before you create the rectangles to represent the eight vertical bars in the graph, you must specify the fill pattern—either an existing or new one. Because this program does not modify the font attribute, UIS$SET_FONT uses a copy of attribute block 0 to set the font attribute ❿. In this case, specify the font ID UIS$FILL_PATTERNS to indicate you want the file of fill patterns.

Now use UIS$SET_FILL_PATTER to set the fill pattern attribute. The program must use two different fill patterns to contrast adult population vertical bars with automobile vertical bars ⓫ ⓬.

The values previously assigned to each element of ARRAY1 and ARRAY2 control the height of the vertical bars ⓭ ⓮.

11.2.2.8 Calling UIS$SET_FONT and UIS$SET_FILL_PATTERN

If you run the program GRAPH now, it produces the vertical bar graph as shown in Figure 11-5.

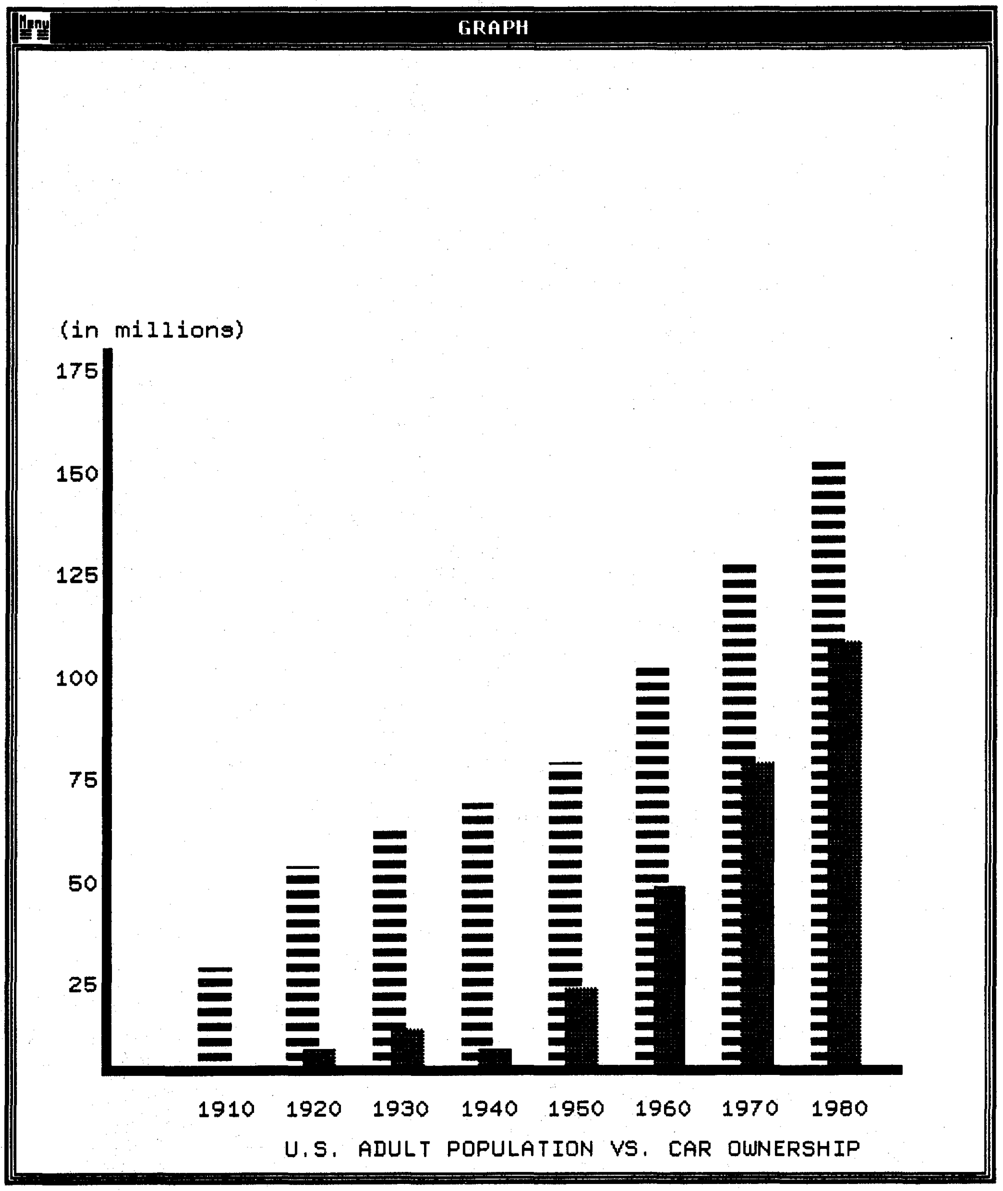
Whenever you create a fill pattern, you must include UIS$SET_FONT and UIS$SET_FILL_PATTERN. The positional order of the calls is important. Calls to UIS routines that modify an attribute block must precede the call that creates the graphic object.

To produce the desired change in the resulting graphic object, the accompanying call to UIS$PLOT must reference the same output attribute block number.

11.2.3 Using the Windowing Attribute

The clipping rectangle attribute modifies the size of the viewable portion of the virtual display. It does not resize the display window or display viewport.

Figure 11-5 Vertical Bar Graph



ZK-4553-85

11.2.3.1 Programming Options

Only the clipping attribute controls what is visible through the display window and viewport.

Clipping Rectangle

To restrict drawing in the virtual display to a specified rectangle, you can use UIS$SET_CLIP to create clipping rectangles that view a portion of your original display window. These rectangles are not display windows, but you can use them to partition your virtual display into discrete areas. They create an environment within your virtual display that can be visited whenever you reference the appropriate attribute block with a modified clipping rectangle attribute. Note that the clipping rectangle merely restricts drawing to an area; it does not change mapping between the virtual display and the display window.

11.2.3.2 Program Development

Programming Objective

To construct three clipping rectangles.

Programming Tasks

1 Create a virtual display.

2 Create a display window and viewport with a title.

3 Choose a font and modify the font attribute.

4 Specify a clipping rectangle and modify the clipping attribute.

5 Use the modified font attribute with clipping disabled to draw a line of text.

6 Use the modified font attribute with clipping enable to draw a line of text.

7 Repeat steps 3 through 6 two more times.

Logical names have been defined for font file names.

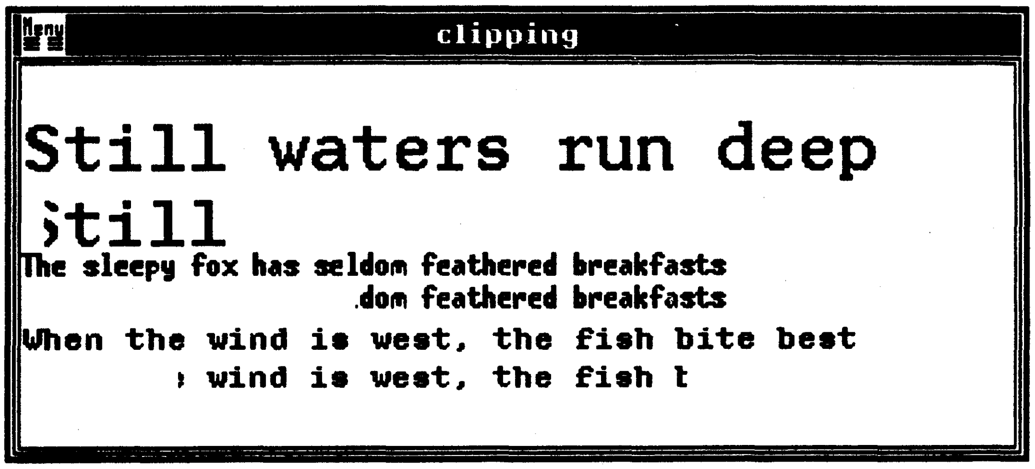
```
PROGRAM CLIP
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
VD_ID=UIS$CREATE_DISPLAY(0.0,0.0,45.0,45.0,15.0,5.0)
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','CLIPPING')

CALL UIS$SET_FONT(VD_ID,0,1,'MY_FONT_5')   1
CALL UIS$SET_CLIP(VD_ID,1,5,1.0,1.0,10.0,40.0)   2
CALL UIS$TEXT(VD_ID,1,'Still waters run deep',0.0,40.0)
CALL UIS$NEW_TEXT_LINE(VD_ID,1)
CALL UIS$TEXT(VD_ID,5,'Still waters run deep')
CALL UIS$SET_FONT(VD_ID,0,2,'MY_FONT_6')  3
CALL UIS$NEW_TEXT_LINE(VD_ID,2)
CALL UIS$SET_CLIP(VD_ID,2,6,15.0,15.0,35.0,40.0)   4
CALL UIS$TEXT(VD_ID,2,'The sleepy fox has seldom feathered breakfas
CALL UIS$NEW_TEXT_LINE(VD_ID,2)
CALL UIS$TEXT(VD_ID,6,'The sleepy fox has seldom feathered breakfas
CALL UIS$SET_FONT(VD_ID,0,3,'MY_FONT_10')   5
CALL UIS$NEW_TEXT_LINE(VD_ID,3)
CALL UIS$SET_CLIP(VD_ID,3,7,7.0,5.0,30.0,40.0)   6
CALL UIS$TEXT(VD_ID,3,'When the wind is west, the fish bite best')
CALL UIS$NEW_TEXT_LINE(VD_ID,3)
CALL UIS$TEXT(VD_ID,7,'When the wind is west, the fish bite best')

PAUSE

END
```

Three fonts ❶ ❸ ❺ illustrate clipping rectangles. The call to UIS$SET_CLIP modifies the attribute block that controls clipping rectangle size. Each call to UIS$SET_CLIP ❷ ❹ ❻ specifies a different clipping rectangle size. Although only one display viewport has been specified in this program, UIS$SET_CLIP creates many compartments within the display window.

11.2.3.3 Calling UIS$SET_CLIP

Your workstation screen displays the graphic objects shown in Figure 11-6.

Figure 11-6 Clipping rectangles



```
Still waters run deep
Still
The sleepy fox has seldom feathered breakfasts
                  .dom feathered breakfasts
When the wind is west, the fish bite best
       ; wind is west, the fish t
```

As you can see, UIS$SET_CLIP has altered the display window of the last three lines. Only portions of each lines are now visible.

12 Inquiry Routines

12.1 Overview

Inquiry routines return program-specific information to the application; in this way, they behave like functions. However, unlike functions that return a single value through a return variable, certain UIS inquiry routines return data in two or more parameters in the argument list. This data can range from current attribute settings to current state of the pointer buttons. Your application program can use this data to establish context during program execution, to check for true or false conditions, or to verify that a requested operation has been performed.

12.2 Using Inquiry Routines

Many common graphics application programs rely on program-specific data such as pointer device position or font size. Inquiry routines return such data to the program. You can use this data as input to the application. Inquiry routines are more properly termed *functions* when you use them with high-level programming languages.

12.2.1 Using Inquiry Routines

Generally, UIS routines in the form UIS$GET_xxxx return information to the application program. Some of these routines behave like functions and return a single value to the program; others return more than one value in the argument list. The routines obtain data about text and font size, windows, keyboard attributes, pointer position, and attribute settings. You can use this data as input to subsequent routines.

12.2.1.1 Programming Options

Your application program can request the following types of application-specific information:

- Color information
- Display list information
- Graphics and text attributes
- Keyboard and pointer characteristics
- Windowing information

Table 12-1 groups inquiry routines by function.

Table 12-1 Inquiry Routines

Inquiry	Information Returned
Color[1]	
UIS$GET_BACKGROUND_INDEX	Background color index
UIS$GET_COLOR	Single RGB color value in a color map entry
UIS$GET_COLORS	RGB color values
UIS$GET_HW_COLOR_INFO	Hardware color map characteristics
UIS$GET_INTENSITIES	Intensity values in virtual color map
UIS$GET_INTENSITY	Single intensity value in a virtual color map entry
UIS$GET_VCM_ID	Virtual color map identifier
UIS$GET_WRITING_INDEX	Writing color index
UIS$GET_WRITING_MODE	Writing mode
UIS$GET_WS_COLOR	Workstation standard color
UIS$GET_WS_INTENSITY	Workstation standard color intensity
Color Conversion[2]	
UIS$HLS_TO_RGB	Converts HLS values to RGB color values
UIS$HSV_TO_RGB	Converts HSV values to RGB color values
UIS$RGB_TO_HLS	Converts RGB values to HLS color values
UIS$RGB_TO_HSV	Converts RGB values to HSV color values
Display List	
UIS$FIND_PRIMITIVE	Identifier of the next primitive in the specified rectangle
UIS$FIND_SEGMENT	Segment identifier of the next segment that contains objects in a specified rectangle
UIS$GET_CURRENT_OBJECT	Identifier of last object drawn in virtual display
UIS$GET_NEXT_OBJECT	Identifier of next object
UIS$GET_OBJECT_ATTRIBUTES	Object type
UIS$GET_PARENT_SEGMENT	Parent segment identifier
UIS$GET_PREVIOUS_OBJECT	Identifier of the previous object
UIS$GET_ROOT_SEGMENT	Root segment identifier
Graphics	
UIS$GET_ARC_TYPE	Arc type used to close arc
UIS$GET_FILL_PATTERN	Fill pattern index and status
UIS$GET_LINE_STYLE	Line style vector
UIS$GET_LINE_WIDTH	Line width in pixels or as a world coordinate x-coordinate width
Keyboard and Pointer	
UIS$GET_ABS_POINTER_POS	Absolute position of the pointer

[1]See Chapter 16 for more information about color and intensity inquiry routines.

[2]See Chapter Chapter 16 for more information about color conversion routines.

Table 12-1 (Cont.) Inquiry Routines

Inquiry	Information Returned
Keyboard and Pointer	
UIS$GET_BUTTONS	State of the pointer device buttons
UIS$GET_KB_ATTRIBUTES	Keyboard characteristics
UIS$GET_POINTER_POSITION	Position of pointer in world coordinates
UIS$GET_TB_INFO	Characteristics of the tablet
UIS$GET_TB_POSITION	Position on tablet in centimeters
UIS$TEST_KB	Successful or unsuccessful connection between virtual and physical keyboard
Text	
UIS$GET_ALIGNED_POSITION	World coordinates along the x-height of the current position of the next character
UIS$GET_CHAR_ROT	Angle of character rotation in degrees
UIS$GET_CHAR_SIZE	If character scaling is enabled and the scaling factors used
UIS$GET_CHAR_SLANT	Angle of character slant in degrees
UIS$GET_CHAR_SPACING	Character and line spacing factor
UIS$GET_FONT	Font name
UIS$GET_FONT_ATTRIBUTES	All font character characteristics
UIS$GET_FONT_SIZE	Font size in centimeters
UIS$GET_LEFT_MARGIN	World coordinate of left margin
UIS$GET_POSITION	World coordinates of text baseline
UIS$GET_TEXT_FORMATTING	Formatting mode
UIS$GET_TEXT_MARGINS	Text margin settings for a line of text
UIS$GET_TEXT_PATH	Direction of text drawing
UIS$GET_TEXT_SLOPE	Angle of the text baseline in degrees
UIS$MEASURE_TEXT	Proportions of text in world coordinates
Windowing	
UIS$GET_CLIP	Clipping rectangle
UIS$GET_DISPLAY_SIZE	Display screen dimensions in centimeters
UIS$GET_VIEWPORT_ICON	Whether or not the icon is occluded
UIS$GET_VIEWPORT_POSITION	Absolute position of display viewport on display screen
UIS$GET_VIEWPORT_SIZE	Dimensions of the display viewport in centimeters
UIS$GET_VISIBILITY	Whether or not viewport is occluded
UIS$GET_WINDOW_ATTRIBUTES	Window and viewport attributes
UIS$GET_WINDOW_SIZE	Dimensions of the display window in world coordinates

12.2.1.2 Program Development I Programming Objective

To return font and viewport information to center text.

Programming Tasks

1. Create a virtual display.
2. Create a display window and viewport with a title.
3. Obtain the font size for a particular character string, viewport size, and display screen size.
4. Choose a font and modify the font attribute block.
5. Use the modified font attribute and information from the inquiry routines to draw a line of centered text in the viewport.
6. Print the inquiry information in the terminal emulation window.
7. Repeat steps 3 through 6.

The font file names used in this program are logical names.

```
          PROGRAM CENTER
          IMPLICIT INTEGER(a-z)
          INCLUDE 'SYS$LIBRARY:UISENTRY'
          INCLUDE 'SYS$LIBRARY:UISUSRDEF'
          REAL F_WIDTH,F_HEIGHT,D_WIDTH,D_HEIGHT
          REAL V_WIDTH,V_HEIGHT
          VD_ID1=UIS$CREATE_DISPLAY(1.0,1.0,15.0,2.0,15.0,2.0)
          WD_ID1=UIS$CREATE_WINDOW(VD_ID1,'SYS$WORKSTATION','CENTERED TEXT')

          CALL UIS$GET_FONT_SIZE('MY_FONT_7','Time has wings',
          2       F_WIDTH,F_HEIGHT)  ❶
          CALL UIS$GET_DISPLAY_SIZE('SYS$WORKSTATION',D_WIDTH,D_HEIGHT)  ❷
          CALL UIS$GET_VIEWPORT_SIZE(WD_ID1,V_WIDTH,V_HEIGHT)  ❸

          CALL UIS$SET_FONT(VD_ID1,0,7,'MY_FONT_7')  ❹
          CALL UIS$TEXT(VD_ID1,7,'Time has wings',
          2       (V_WIDTH-F_WIDTH)/2,
          2       V_HEIGHT)    ❺

          PAUSE
          PRINT 50
 50       FORMAT(T10,'FIRST LINE',T39,'WIDTH',T51,'HEIGHT')

          PRINT 75
 75       FORMAT(T2,'---------------------------------------------------',
          2       '------')

          PRINT 100, F_WIDTH, F_HEIGHT
 100      FORMAT(T2,'The dimensions of the font are:',
          2       T39,f5.2,T46,'cm.',T51,f5.2,T58,'cm.')
          PRINT 150,D_WIDTH,D_HEIGHT
 150      FORMAT(T2,'The dimensions of the display are:',
          2       T39,f6.2,T46,'cm.',T51,f6.2,T58,'cm.')

          PRINT 200,V_WIDTH,V_HEIGHT
 200      FORMAT(T2,'The dimensions of the viewport are:',
          2       T39,f6.2,T46,'cm.',T51,f6.2,T58,'cm.')
          CALL UIS$SET_FONT(VD_ID1,7,8,'MY_FONT_5')  ❻
          CALL UIS$MEASURE_TEXT(VD_ID1,8,'four seasons',
          2       F_WIDTH,F_HEIGHT)
          CALL UIS$NEW_TEXT_LINE(VD_ID1,8)
          CALL UIS$TEXT(VD_ID1,8,'four seasons',
          2       (V_WIDTH-F_WIDTH)/2,(V_HEIGHT-F_HEIGHT))  ❼
          TYPE *,' '
          PRINT 550
550       FORMAT(T10,'SECOND LINE',T39,'WIDTH',T51,'HEIGHT')

          PRINT 575
575       FORMAT(T2,'---------------------------------------------------------',
2               '------')

          PRINT 610, F_WIDTH, F_HEIGHT
610       FORMAT(T2,'THE DIMENSIONS OF THE FONT ARE:',
2               T39,f5.2,T46,'cm.',T51,f5.2,T58,'cm.')
          PRINT 700,D_WIDTH,D_HEIGHT
700       FORMAT(T2,'The dimensions of the display are:',
2               T39,f6.2,T46,'cm.',T51,f6.2,T58,'cm.')

          PRINT 800,V_WIDTH,V_HEIGHT
800       FORMAT(T2,'The dimensions of the viewport are:',
2               T39,f6.2,T46,'cm.',T51,f6.2,T58,'cm.')

          PAUSE
          END
```

The three inquiry functions UIS$GET_FONT_SIZE, UIS$GET_DISPLAY_SIZE, and UIS$GET_VIEWPORT_SIZE are called ❶ ❷ ❸. Each function returns data to uniquely specified variables within its argument list.

A logical name is defined ❹ ❻ to represent the 31-character font file name. The first call to UIS$TEXT ❺ places a text string in the window. The starting position for creating text is calculated from the expression in the argument list. VAX FORTRAN allows arithmetic expressions as arguments.❼ If your application is written in a programming language other than VAX FORTRAN, refer to the appropriate language reference manual.

To center the text in this window, the length of the text is subtracted from the total width of the viewport and the result divided by two. The distance of the text from the lower border of the window (the *y* coordinate) equals the value of the variable *v_height,* the height of the display viewport.

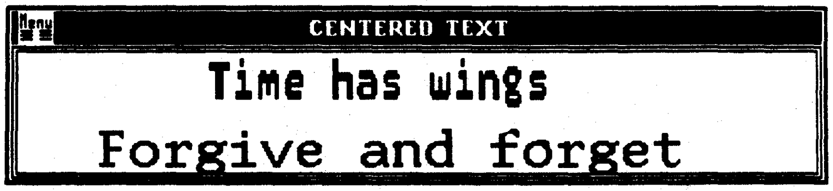
NOTE: **Before you run the demonstration programs, you must invoke the indirect command file SYS$EXAMPLES:DEFFONT.COM.**

12.2.1.3 Invoking UIS$GET_FONT_SIZE, UIS$GET_DISPLAY_SIZE, and UIS$GET_VIEWPORT_SIZE

If you run this program now, your workstation screen will display graphic objects as shown in Figure 12-1.

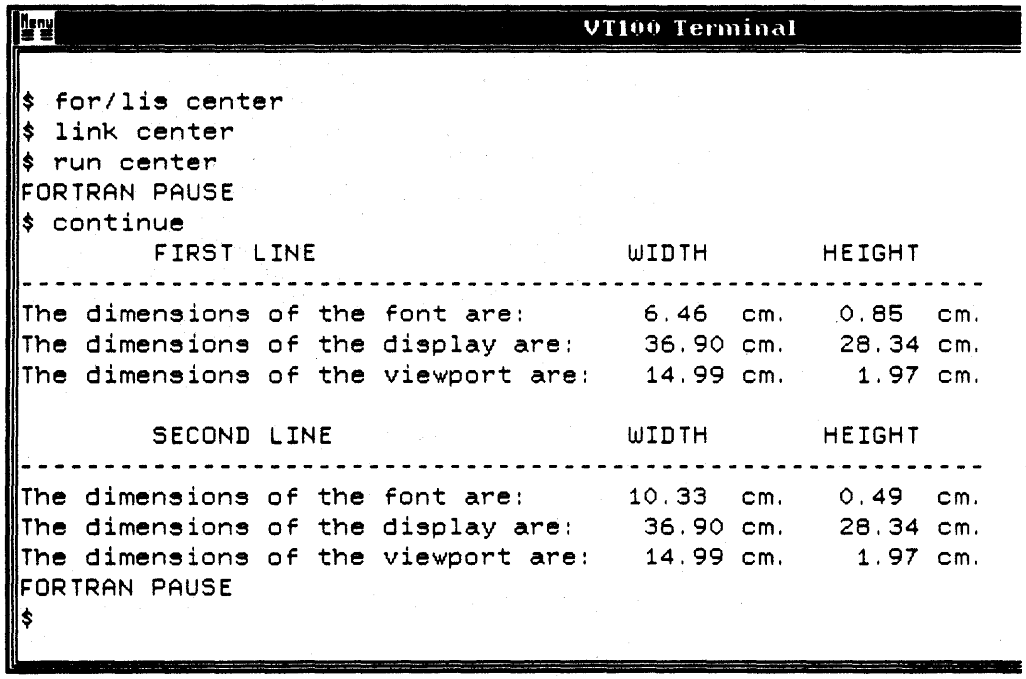
Note that output from the FORTRAN PRINT or TYPE statement is not displayed in the window. The TYPE and PRINT statements are equivalent to the logical names FOR$TYPE and FOR$PRINT, which translate to the logical name SYS$OUTPUT. Only UIS$TEXT can write text to a virtual display.

Figure 12–1 Centering Text



Time has wings

Forgive and forget



```
$ for/lis center
$ link center
$ run center
FORTRAN PAUSE
$ continue
         FIRST LINE                          WIDTH        HEIGHT
-------------------------------------------------------------------
The dimensions of the font are:        6.46  cm.     0.85  cm.
The dimensions of the display are:    36.90 cm.     28.34 cm.
The dimensions of the viewport are:   14.99 cm.      1.97 cm.

         SECOND LINE                         WIDTH        HEIGHT
-------------------------------------------------------------------
The dimensions of the font are:       10.33  cm.     0.49  cm.
The dimensions of the display are:    36.90 cm.     28.34 cm.
The dimensions of the viewport are:   14.99 cm.      1.97 cm.
FORTRAN PAUSE
$
```

ZK-4555-85

12.2.1.4 Program Development II

Programming Objective

To construct a pie graph that illustrates the operating budget of a small New England town.

Programming Tasks

1 Create a virtual display.

2 Create a display window and viewport with a title.

3 Choose a font and modify the font attribute.

4 Use the modified font attribute to print the title of the graph.

5 Obtain font information.

6 Modify the arc type attribute.

7 Choose a fill pattern and modify the font attribute and the fill pattern attribute.

8 Use the modified fill pattern attribute to draw an arc.

9 Draw part of the legend below the pie graph.

10 Obtain and print arc type and fill pattern information.

11 Repeat steps 6 through 9.

```
         PROGRAM PIE_GRAPH
         IMPLICIT INTEGER(A-Z)
         INCLUDE 'SYS$LIBRARY:UISENTRY'
         INCLUDE 'SYS$LIBRARY:UISUSRDEF'
         CHARACTER*32 BUFFERDESC
         LOGICAL*4 FILL_ENABLED
         VD_ID=UIS$CREATE_DISPLAY(-3.0,-3.0,25.0,25.0,15.0,15.0)
         WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','PIE GRAPH')

         CALL UIS$SET_FONT(VD_ID,0,9,'MY_FONT_10')
         CALL UIS$TEXT(VD_ID,9,'OPERATING BUDGET',6.0,24.0)
         CALL UIS$TEXT(VD_ID,9,'TOWN OF GREENWICH, MASS.',4.0,22.0)
         CALL UIS$GET_FONT(VD_ID,9,BUFFERDESC,LENGTH)     ❶

         PRINT 10,BUFFERDESC
10       FORMAT(T2,'THE FONT NAME IS',T20,A31)    ❷

         PRINT 11,LENGTH
11       FORMAT(T2,'THE LENGTH OF THE FONT NAME IS ',T33,I3,T37,'CHARACTERS'

         CALL UIS$SET_ARC_TYPE(VD_ID,0,1,UIS$C_ARC_PIE)    ❸
         CALL UIS$SET_FONT(VD_ID,1,1,'UIS$FILL_PATTERNS')
         CALL UIS$SET_FILL_PATTERN(VD_ID,1,1,PATT$C_BRICK_DOWNDIAG)
         CALL UIS$CIRCLE(VD_ID,1,10.0,10.0,8.0,0.0,50.0)
         call uis$plot(vd_id,1,0.0,0.0,2.0,0.0,2.0,-1.0,
         2       0.0,-1.0,0.0,0.0)
         call uis$text(vd_id,0,'Fire',3.0,0.0)
         ARC_TYPE=UIS$GET_ARC_TYPE(VD_ID,1)        ❹
         FILL_ENABLED=UIS$GET_FILL_PATTERN(VD_ID,1,INDEX)  ❺

         PRINT 15,ARC_TYPE
15       FORMAT(T2,'THE ARC TYPE IS',T25,I1)          ❻

         PRINT 20,FILL_ENABLED
20       FORMAT(T2,'IS THE FILL PATTERN ENABLED?',T32,L1)

         CALL UIS$SET_FONT(VD_ID,1,2,'UIS$FILL_PATTERNS')
         CALL UIS$SET_FILL_PATTERN(VD_ID,2,2,PATT$C_DOWNDIAG4_4)
         CALL UIS$CIRCLE(VD_ID,2,10.0,10.0,8.0,50.0,95.0)
         CALL UIS$PLOT(VD_ID,2,10.0,0.0,12.0,0.0,12.0,-1.0,
         2       10.0,-1.0,10.0,0.0)
         CALL UIS$TEXT(VD_ID,0,'Sanitation',14.0,0.0)

         CALL UIS$SET_FONT(VD_ID,2,3,'UIS$FILL_PATTERNS')
         CALL UIS$SET_FILL_PATTERN(VD_ID,3,3,PATT$C_HORIZ2_6)
         CALL UIS$CIRCLE(VD_ID,3,10.0,10.0,8.0,95.0,165.0)
         CALL UIS$PLOT(VD_ID,3,0.0,-2.0,2.0,-2.0,2.0,-3.0,
         2       0.0,-3.0,0.0,-2.0)
         CALL UIS$TEXT(VD_ID,0,'Police',3.0,-2.0)

         CALL UIS$SET_FONT(VD_ID,3,4,'UIS$FILL_PATTERNS')
         CALL UIS$SET_FILL_PATTERN(VD_ID,4,4,PATT$C_GREY4_16D)
         CALL UIS$CIRCLE(VD_ID,4,10.0,10.0,8.0,165.0,360.0)
         CALL UIS$PLOT(VD_ID,4,10.0,-2.0,12.0,-2.0,12.0,-3.0,
         2       10.0,-3.0,10.0,-2.0)
         CALL UIS$TEXT(VD_ID,0,'Schools',14.0,-2.0)

         PAUSE
         END
```

The program PIE_GRAPH returns information about the graph heading. A call to UIS$GET_FONT ❶ identifies the font and its length ❷. The font MY_FONT_10 is a logical name for a 31-character font file name.

Attribute block 1 contains the modified arc type attribute $\boxed{3}$. When a new section of the arc is drawn, it will have a pie arc type that enables fill pattern.

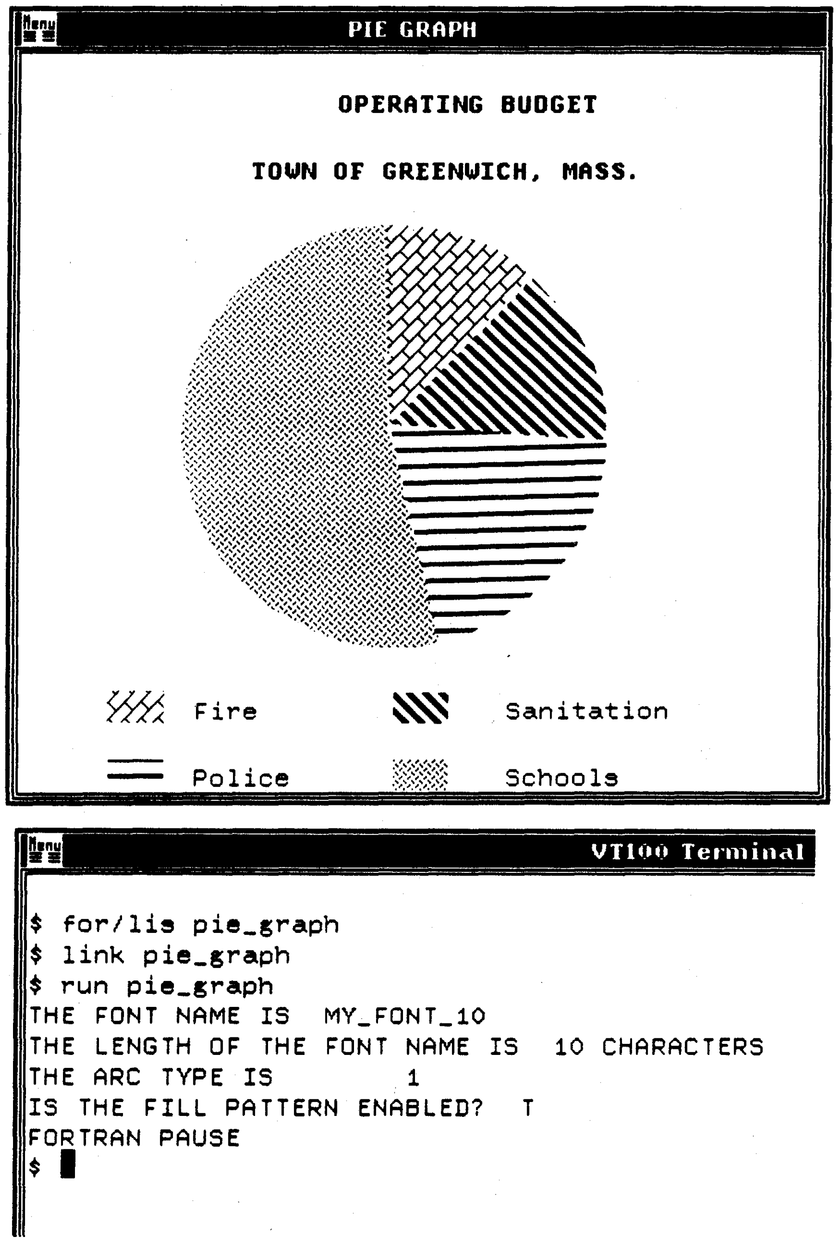
Arc type information is returned in the variable *arc_type* $\boxed{4}$.

A call to UIS$GET_FILL_PATTERN $\boxed{5}$ tests whether fill patterns are enabled. Fill pattern information is returned in the variable *fill_enabled* $\boxed{6}$ as a Boolean value.

12.2.1.5 Invoking UISGET_ARC_TYPE, UISGET_FILL_PATTERN, and UIS$GET_FONT

The program PIE_GRAPH draws a pie graph with four fill patterns. It requests and displays certain program-specific information as shown in Figure 12–2.

Figure 12–2 Pie Graph



```
$ for/lis pie_graph
$ link pie_graph
$ run pie_graph
THE FONT NAME IS  MY_FONT_10
THE LENGTH OF THE FONT NAME IS  10 CHARACTERS
THE ARC TYPE IS          1
IS THE FILL PATTERN ENABLED?  T
FORTRAN PAUSE
$ █
```

ZK-4556-85

13 Display Lists and Segmentation

13.1 Overview

As your displays become more complex, you should understand display list concepts. This chapter discusses the following topics:

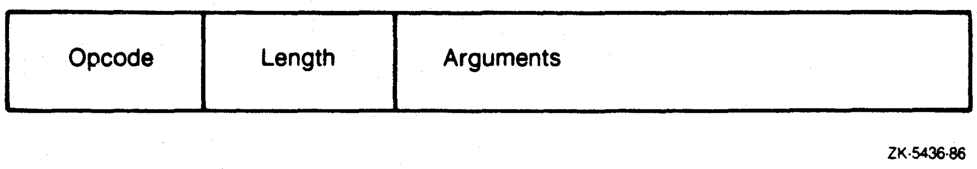
- Creating and searching segments
- Editing and walking the display list
- Disabling display lists
- Creating UIS metafiles
- Attaching private data to graphic objects

Consider the creation of complex objects as a challenge to simplify and to modularize your coding through the use of *segmentation*.

13.2 Display Lists

UIS constructs a *display list* of encoded commands for graphics. A display list is a device-independent encoding of the exact contents of the virtual display. The display list remains resident in memory for use by UIS routines. Figure 13–1 shows the format of an entry in the display list.

Figure 13–1 Binary Encoded Instruction



UIS signals an error if it encounters an invalid opcode.

Whenever you call UIS routines to create graphic objects or modify attribute blocks, you add an entry to a display list. Each virtual display has only one display list.

UIS maintains display lists for the following purposes:

- Automatic management of panning, zooming, resizing, and duplication of display windows
- High resolution printing of physical and virtual displays
- Structuring and manipulation of graphic objects in the virtual display

- Storage of the contents of the virtual display in a buffer for later reexecution

13.3 Segments

A *segment* consists of calls to UIS graphics and text routines (and any nested segments). You create a segment explicitly with a call to UIS$BEGIN_SEGMENT; you terminate a segment with a call to UIS$END_SEGMENT. A complex display list is a hierarchy of nested segments.

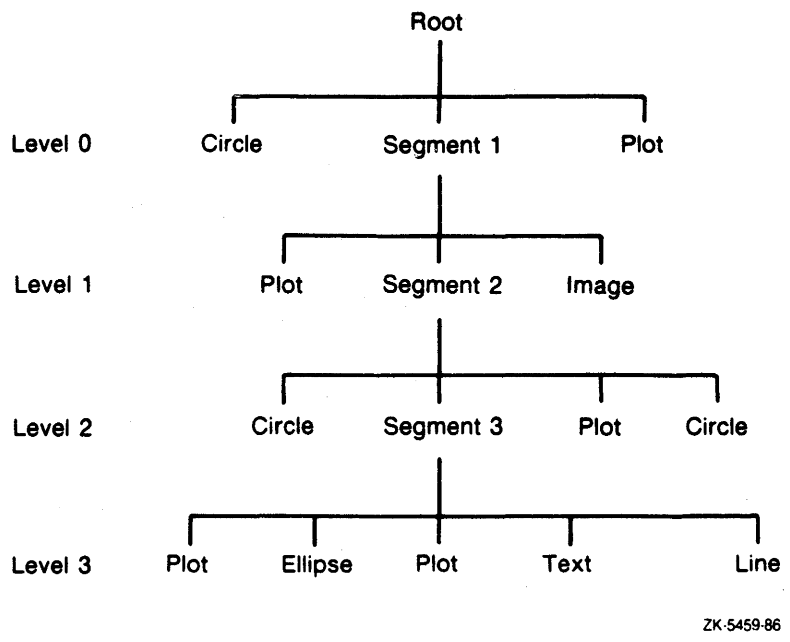
A top-level *root* segment contains any segment or output (graphic and text) routine that is not in an explicitly created segment.

Segmentation of graphics routines facilitates transformations—scaling, rotation, and translation. Segmentation also modularizes attributes. You can construct complex graphic objects in sections, where each logical grouping of display list entries is in a segment. You can transform or display such segments individually and independently of the rest of the object. Changes to attributes in a segment do not affect the attribute settings of a higher-level segment.

For example, a house, a barn, and landscape are constructed as three logical groupings, or *subpictures*, of a complex display. Each subpicture is a segment of appropriate UIS routines. You can manipulate each subpicture independently of one other.

Figure 13-2 shows a tree diagram of a display list containing nested segments. Read the diagram from left to right and downward until there are no more segments. Read each level to the right and move upward to the next level where you left off.

Figure 13–2 Nested Segments



13.3.1 Identifiers and Object Types

There are many types of UIS identifiers—for example, virtual display identifier, virtual keyboard identifier, transformation identifier, and so on. Identifiers allow an application to reference and manipulate internal objects. To manage the display list, follow these steps:

1. Traverse the display list downward object by object.
2. Search a segment.
3. Traverse upward through the segment path.

Segments

UIS$BEGIN_SEGMENT returns a unique identifier to each segment. If you do not use UIS$BEGIN_SEGMENT to declare any segments explicitly, you can use the unique identifier of the root segment to manipulate the display list.

Objects

Every object in the virtual display has an object identifier. However, not all routines return identifiers explicitly. Object and segment identifiers are useful in walking and editing the display list. Use them as reference points within complex display lists.

Sometimes the identifier is not part of the calling sequence; in this case, you must use another UIS routine to return the identifier. For example, none of the graphics and text routines return identifiers explicitly. You can use the routines listed in the following table to return the identifiers.

Graphic Object	Identifier	Routine
Segment	seg_id	UIS$BEGIN_SEGMENT[1]
Root segment	root_id	UIS$GET_ROOT_SEGMENT
Parent segment	parent_id	UIS$GET_PARENT_SEGMENT
Graphic objects	prev_id	UIS$GET_PREVIOUS_OBJECT
	current_id	UIS$GET_CURRENT_OBJECT
	next_id	UIS$GET_NEXT_OBJECT

[1]UIS$BEGIN_SEGMENT returns the segment identifier in a return variable, *seg_id*.

Object Types

Although you can use segment and object identifiers to manipulate the display list, you must further identify those objects within a segment. You should know the display list entry *object type*. UIS categorizes graphic objects by object type. The following table lists six object types and their symbols.

Symbol	Graphic Object
UIS$C_OBJECT_SEGMENT	New segment
UIS$C_OBJECT_PLOT	Point, line, or polygon
UIS$C_OBJECT_TEXT	Text
UIS$C_OBJECT_ELLIPSE	Ellipse or circle
UIS$C_OBJECT_IMAGE	Raster image
UIS$C_OBJECT_LINE	Unconnected lines

UIS$GET_OBJECT_ATTRIBUTES returns object type information.

13.3.2 Programming Options

From the options available below, the following programs are constructed:

- Program to disable display lists
- Program to walk the display list

Creating Segments

You can use UIS$BEGIN_SEGMENT and UIS$END_SEGMENT to create an unlimited number of segments explicitly. For each newly created segment, UIS returns a unique identifier that appropriate UIS routines use to locate and edit segments. You can also nest segments within segments.

NOTE: **If you call UIS$BEGIN_SEGMENT before you call any graphics and text routines, the segment is deleted and the returned identifier is no longer valid. To create an empty segment, call UIS$BEGIN_SEGMENT, then UIS$PRIVATE. This sequence places private data in the segment. Now UIS$END_SEGMENT does not consider the segment empty.**

Enabling and Disabling Display Lists

When you disable a display list, nothing can be added to the list. You can enable and disable a display list explicitly any number of times with UIS$ENABLE_DISPLAY_LIST and UIS$DISABLE_DISPLAY_LIST. However, to see the results of disabling a display list, you must execute the display list. Use UIS$EXECUTE or any of the routines listed in the following table to execute the display list.

Routine	Function
UIS$CREATE_WINDOW	Creates a display window and viewport
UIS$DELETE_OBJECT[1,5]	Deletes an object in the virtual display
UIS$EXECUTE[2,5]	Executes the display list
UIS$MOVE_AREA[3,5]	Moves a portion of the virtual display another part of the virtual display
UIS$MOVE_WINDOW[4,5]	Redefines the display window coordinate space.

[1]UIS$DELETE_OBJECT executes the display list only when the object to be deleted occluded another object.

[2]UIS$EXECUTE executes the entire display list if **buflen** and **bufaddr** are not specified.

[3]UIS$MOVE_AREA executes the display list only if the specified source and destination rectangles lie within a display window.

[4]UIS$MOVE_WINDOW executes the display list only if the window size is changed.

[5]This routine checks display list flags.

The position of UIS$DISABLE_DISPLAY_LIST and UIS$ENABLE_DISPLAY_LIST in your program is important. If the display list is disabled **after** the display list is executed, the viewport displays all the graphic objects drawn in the virtual display. If the display list is disabled **before** one of the above routines is called, the viewport displays none of the graphic objects created between calls to UIS$DISABLE_DISPLAY_LIST and UIS$ENABLE_DISPLAY_LIST. No binary instructions are added to the display list.

Walking the Display List

You can traverse, or *walk* the entire display list from top to bottom and from object to object with UIS$GET_ROOT_SEGMENT and UIS$GET_NEXT_OBJECT.

Searching a Segment

If the display list contains segments, you can search the contents of any segment in the display list with UIS$GET_NEXT_OBJECT.

Traversing the Segment Path

Because the root segment is the ultimate parent segment, every nested segment has a parent segment. The root segment acts as the parent for all level-one segments (see Figure 13–2). A segment identifier notes the beginning of each segment in a display list. The segment identifiers within a display list constitute its *segment path*. You can traverse the segment path from the innermost segment outward with UIS$GET_PARENT_SEGMENT.

13.3.3 Program Development I

Programming Objective

To disable a display list.

Programming Tasks

1 Create a virtual display.
2 Create a display window and viewport.
3 Disable the display list.
4 Draw some graphic objects in the virtual display.
5 Reenable the display list.
6 Draw some graphic objects in the virtual display.
7 Create a second display window and viewport.

```
        PROGRAM LIST
        IMPLICIT INTEGER(A-Z)
        INCLUDE 'SYS$LIBRARY:UISENTRY'
        INCLUDE 'SYS$LIBRARY:UISUSRDEF'
        VD_ID=UIS$CREATE_DISPLAY(-1.0,-1.0,50.0,50.0,10.0,10.0)
        WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','MORE')  1
c Disable the display list

        CALL UIS$DISABLE_DISPLAY_LIST(VD_ID)   2
c Draw the graphic objects

        CALL UIS$CIRCLE(VD_ID,0,15.0,15.0,5.0)
        CALL UIS$CIRCLE(VD_ID,0,5.0,5.0,5.0)
        CALL UIS$PLOT(VD_ID,0,27.0,17.0,35.0,17.0,35.0,24.0,27.0,24.0,
        2      27.0,17.0)
        CALL UIS$CIRCLE(VD_ID,0,35.0,35.0,8.0)
        CALL UIS$PLOT(VD_ID,0,5.0,30.0,15.0,30.0,10.0,40.0,5.0,30.0)

        PAUSE
c Reenable the display list

        CALL UIS$ENABLE_DISPLAY_LIST(VD_ID)    3
c Draw circle and triangle

        CALL UIS$CIRCLE(VD_ID,0,33.0,35.0,8.0)  4
        CALL UIS$PLOT(VD_ID,0,7.0,31.0,17.0,31.0,12.0,41.0,7.0,31.0)  5

        WD_ID1=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','LESS')   6

        PAUSE

        END
```

Initially, a display window and viewport labeled MORE are created 1. The world coordinate range of the window defaults to that of the virtual display.

The display list is disabled 2.

Five graphic objects are drawn in the virtual display—three circles, a triangle, and a square. Although all five objects appear in the viewport MORE, no entries are added to the display list.

After the PAUSE statement, the display list is reenabled 3 and a triangle and another circle are drawn 4 5.

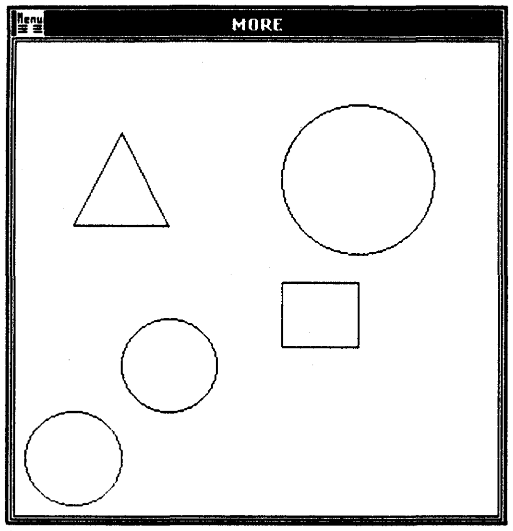
Because the first call to UIS$CREATE_WINDOW was executed **before** the display list was disabled, objects drawn in the virtual display and within the display window are displayed in the viewport but are **not** added to the display list.

Finally, the second display window and viewport labeled LESS are created ❻. The display list is executed, and all objects **except** those included within the disable-enable request appear in the viewport LESS.

13.3.3.1 Calling UIS$DISABLE_DISPLAY_LIST and UIS$ENABLE_DISPLAY_LIST

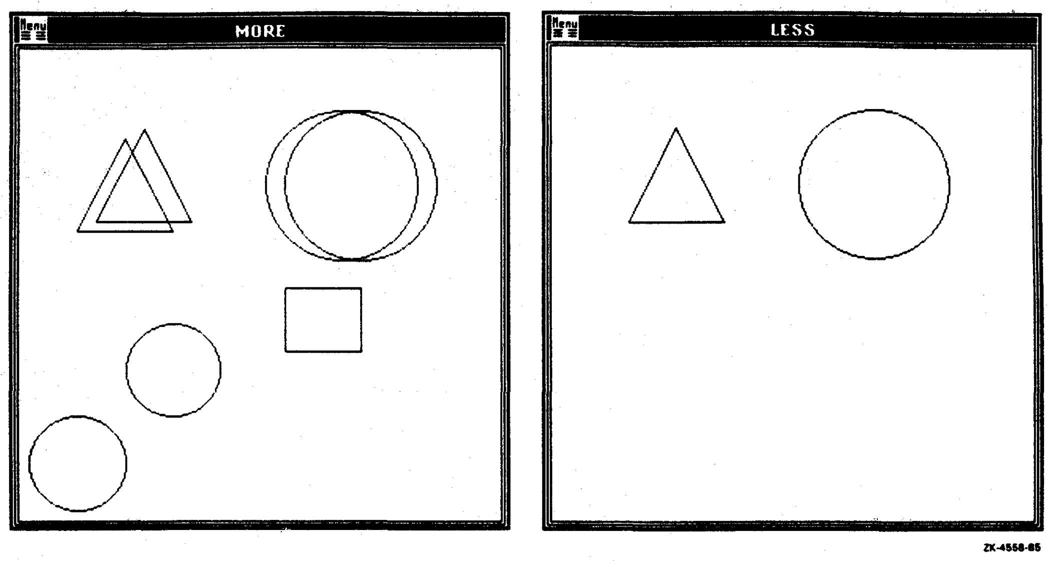
When the program executes, the viewport MORE is displayed first as shown in Figure 13-3.

Figure 13-3 Disabling a Display List



Type CONTINUE at the dollar sign prompt ($). Figure 13-4 shows viewports MORE and LESS. Note that the second call to UIS$CREATE_ WINDOW executes the display list.

Figure 13–4 After Display List Execution



13.3.3.2 Program Development II

Programming Objectives

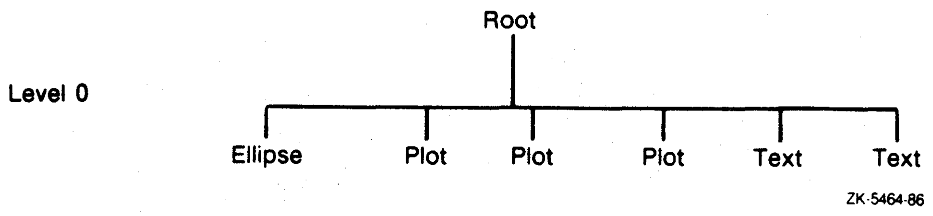
To traverse the entire display list and examine each object type.

Programming Tasks

1. Create a virtual display.
2. Draw graphic objects in the virtual display.
3. Print output headings in the emulation window.
4. Obtain the identifier of the root segment.
5. Walk downward through the display list.
6. Examine each object type and place its identifier in one of five arrays.

Figure 13–5 shows a tree diagram of the program WALK.

Figure 13-5 Tree Diagram—Program WALK



The program WALK draws objects in a virtual display, then identifies each object by walking the entire display list and examining the various object type values. The program also shows how to collect and store object identifiers according to object type. If you run program WALK, compile the subroutine DETERMINE as a separate module and link it with WALK.

```
        PROGRAM WALK
        IMPLICIT INTEGER(A-Z)
        INCLUDE 'SYS$LIBRARY:UISENTRY'
        INCLUDE 'SYS$LIBRARY:UISUSRDEF'
        COMMON NEXT_ID1,TYPE1    ❶
        VD_ID1=UIS$CREATE_DISPLAY(0.0,0.0,40.0,40.0,20.0,20.0)  ❷

C Draw objects in virtual display

        CALL UIS$CIRCLE(VD_ID1,0,15.0,15.0,6.0)
        CALL UIS$PLOT(VD_ID1,0,1.0,1.0,20.0,1.0,20.0,8.0,1.0,1.0)
        CALL UIS$PLOT(VD_ID1,0,20.0,20.0,40.0,20.0,30.0,35.0,20.0,
        2      20.0)
        CALL UIS$PLOT(VD_ID1,0,3.0,25.0,13.0,25.0,13.0,35.0,
        2      3.0,35.0,3.0,25.0)
        CALL UIS$TEXT(VD_ID1,0,'The footsteps of fortune are slippery',
        2      0.0,38.0)
        CALL UIS$NEW_TEXT_LINE(VD_ID1,0)
        CALL UIS$TEXT(VD_ID1,0,'Mirth without measure is madness')
        PRINT 10
 10     FORMAT(T2,'DISPLAY LIST ELEMENTS')
        PRINT 20
 20     FORMAT(T1,'-------------------------------')
        PRINT 30
 30     FORMAT(T2,'IDENTIFIER',T17,'OBJECT TYPE')

        ROOT_ID1=UIS$GET_ROOT_SEGMENT(VD_ID1)     ❸
        NEXT_ID1 = ROOT_ID1

c Walk the display list

        DO WHILE (NEXT_ID1 .NE. 0)                ❹
        TYPE1=UIS$GET_OBJECT_ATTRIBUTES(NEXT_ID1) ❺
        CALL DETERMINE                            ❻
        NEXT_ID1=UIS$GET_NEXT_OBJECT(NEXT_ID1)    ❼
        ENDDO                                     ❽
        WD_ID1=UIS$CREATE_WINDOW(VD_ID1,'SYS$WORKSTATION')   ❾

        PAUSE
        END

        SUBROUTINE DETERMINE    ❿
        IMPLICIT INTEGER(A-Z)
        INCLUDE 'SYS$LIBRARY:UISENTRY'
        INCLUDE 'SYS$LIBRARY:UISUSRDEF'
        INTEGER*4 SEG_ARRAY(6),PLOT_ARRAY(6),TEXT_ARRAY(6),ELLIP_ARRAY(6) ⓫
        INTEGER*4 LINE(6),IMAGE(6)                                  ⓬
        DATA H,I,J,K,L,M/1,1,1,1,1,1/        ⓭
        COMMON NEXT_ID1,TYPE1            ⓮
```

```
         IF (TYPE1 .EQ. UIS$C_OBJECT_SEGMENT) THEN          15
         SEG_ARRAY(H)= NEXT_ID1
         PRINT 40,SEG_ARRAY(H),TYPE1
40       FORMAT(T2,I6,T19,I1,T24,'SEGMENT')
         H = H + 1
         ENDIF

         IF (TYPE1 .EQ. UIS$C_OBJECT_PLOT) THEN             16
         PLOT_ARRAY(I) = NEXT_ID1
         PRINT 50,PLOT_ARRAY(I),TYPE1
50       FORMAT(T2,I6,T19,I1,T24,'PLOT')
         I = I + 1
         ENDIF
         IF (TYPE1 .EQ. UIS$C_OBJECT_TEXT) THEN             17
         TEXT_ARRAY(J) = NEXT_ID1
         PRINT 55,TEXT_ARRAY(J),TYPE1
55       FORMAT(T2,I6,T19,I1,T24,'TEXT')
         J = J + 1
         ENDIF

         IF (TYPE1 .EQ. UIS$C_OBJECT_ELLIPSE) THEN          18
         ELLIP_ARRAY(K) = NEXT_ID1
         PRINT 60,ELLIP_ARRAY(K),TYPE1
60       FORMAT(T2,I6,T19,I1,T24,'ELLIPSE')
         K = K + 1
         ENDIF
         IF (TYPE1 .EQ. UIS$C_OBJECT_LINE) THEN     19
         LINE(L) = NEXT_ID1
         PRINT 70,LINE(L),TYPE1
70       FORMAT(T2,I6,T19,I1,T24,'NEW TEXT LINE')
         L = L + 1
         ENDIF

         IF (TYPE1 .EQ. UIS$C_OBJECT_IMAGE) THEN      20
         IMAGE(M) = NEXT_ID1
         PRINT 80,IMAGE(M),TYPE1
80       FORMAT(T2,I6,T19,I1,T24,'IMAGE')
         M = M + 1
         ENDIF

         RETURN
         END
```

The variables *next_id1* and *type1* are used in both the main program and the subroutine DETERMINE. The COMMON statement ensures access to data stored in both locations by both the main program and the subroutine 1 14.

A virtual display is created 2. As objects are drawn in the virtual display, display list entries in the form of encoded binary data identifying the particular objects are added to the display list. Only one display list is created for each virtual display.

Because the entire display list is to be traversed, the root segment will be the starting point and its identifier must be returned 3.

A DOWHILE loop 4 8 implements traversing the display list through successive calls to UIS$GET_NEXT_OBJECT 7.

An object type for each display list entry is returned 5.

Within the DOWHILE loop, the subroutine DETERMINE is called 6 10 to sort each object identifier according to its object type 15 16 17 18 19 20. For more information about object type symbols such as UIS$C_OBJECT_PLOT, see UIS$GET_OBJECT_ATTRIBUTES.

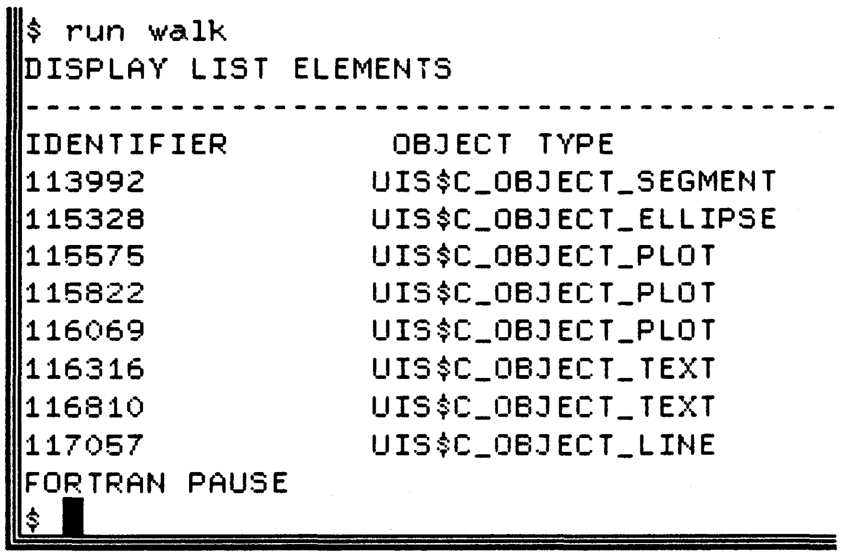
For each object type represented in the display list, five arrays are declared 11 12. Each object identifier is stored in one of these arrays. All counter variables are initialized to the value *1* 13.

A call to UIS$CREATE_WINDOW creates a display window and viewport, and executes the contents of the display list in the virtual display⑨.

13.3.3.3. Calling UISGET_NEXT_OBJECT, UISGET_OBJECT_ATTRIBUTES, and UIS$GET_ROOT_SEGMENT

The program WALK walks the display list and identifies each object. Figure 13-6 shows how each object is returned in the terminal emulation window.

Figure 13-6 Display List Elements

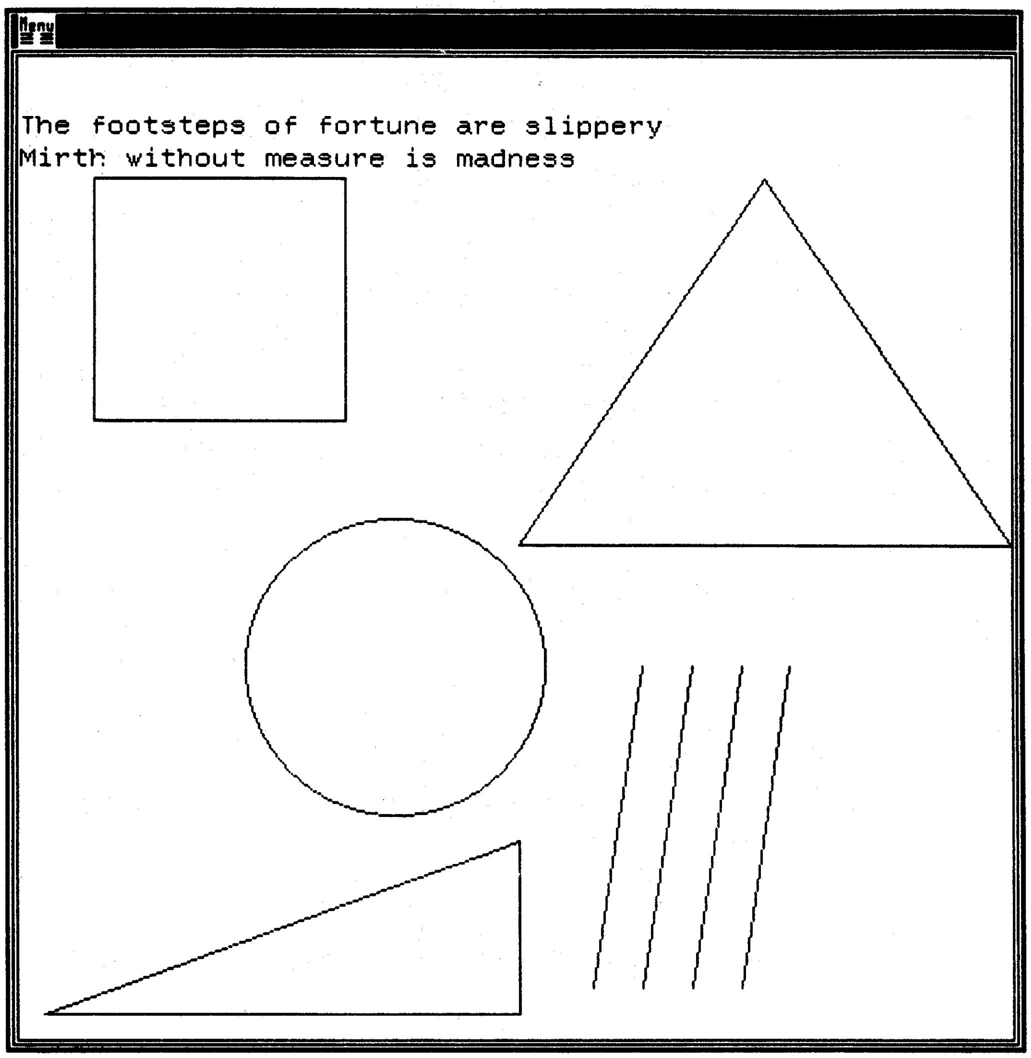


```
$ run walk
DISPLAY LIST ELEMENTS
----------------------------------------
IDENTIFIER          OBJECT TYPE
113992             UIS$C_OBJECT_SEGMENT
115328             UIS$C_OBJECT_ELLIPSE
115575             UIS$C_OBJECT_PLOT
115822             UIS$C_OBJECT_PLOT
116069             UIS$C_OBJECT_PLOT
116316             UIS$C_OBJECT_TEXT
116810             UIS$C_OBJECT_TEXT
117057             UIS$C_OBJECT_LINE
FORTRAN PAUSE
$
```

ZK-5255-86

The program WALK also creates a display window and viewport with the objects in the virtual display as shown in Figure 13-7.

Figure 13-7 Contents of the Display List



```
The footsteps of fortune are slippery
Mirth without measure is madness
```

13.3.3.4 Program Development III

Programming Objectives

To create a display list with a nested segment, traverse upward through the segment path, then search downward through a specified segment.

Programming Tasks

1 Create a virtual display.

2 Create a display window and viewport.

3 Create five levels of nested segments.

4 Print output headings in the emulation window.

5 Beginning at the innermost nested segment, use UIS$GET_PARENT_SEGMENT to obtain and print the parent segment identifier.

6 Print output headings in the emulation window.

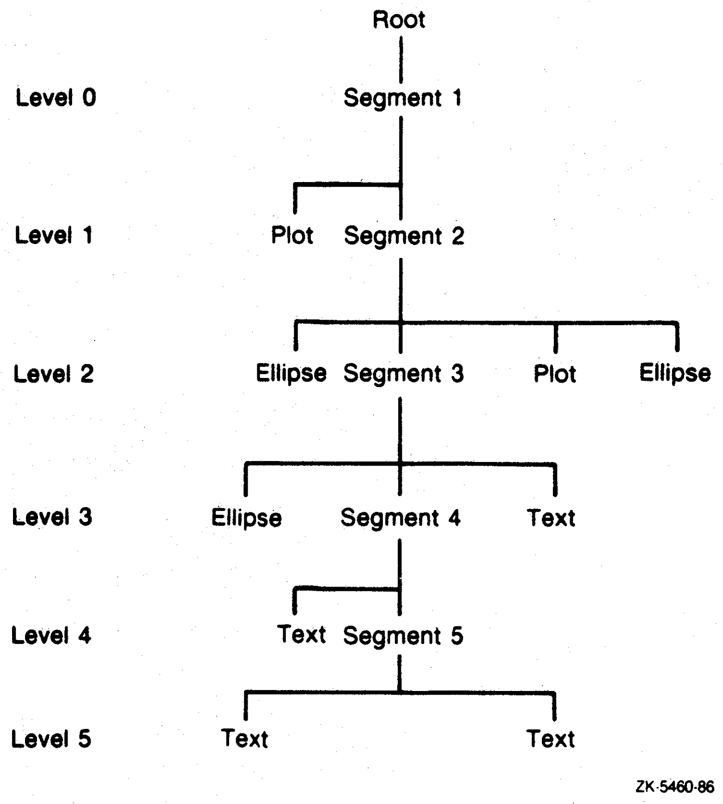
7 Choose a segment to search.

8 Use UIS$GET_NEXT_OBJECT to walk downward through the segment.

9 Call the subroutine DETERMINE to examine and store the objects in arrays by object type.

Figure 13–8 shows the structure of the display list in the program HOP.

Figure 13–8 Display List Structure in Program HOP



To run program HOP, compile the subroutine DETERMINE from the preceding program WALK as a separate module and link it with HOP.

```
PROGRAM HOP
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
COMMON NEXT_ID1,TYPE1

VD_ID2=UIS$CREATE_DISPLAY(-1.0,-1.0,40.0,40.0,15.0,15.0)
```

```
        SEG_ID1=UIS$BEGIN_SEGMENT(VD_ID2)
          CALL UIS$PLOT(VD_ID2,0,0.0,12.0,5.0,12.0,7.5,17.0,10.0,
        2       12.0,15.0,12.0,
        2       12.5,7.5,15.0,0.0,7.5,5.0,0.0,0.0,2.5,7.5,0.0,12.0)
          SEG_ID2=UIS$BEGIN_SEGMENT(VD_ID2)                             1
            CALL UIS$CIRCLE(VD_ID2,0,7.5,8.0,8.0)
             SEG_ID3=UIS$BEGIN_SEGMENT(VD_ID2)
             CALL UIS$ELLIPSE(VD_ID2,0,25.0,8.0,5.0,8.0)
               SEG_ID4=UIS$BEGIN_SEGMENT(VD_ID2)
               CALL UIS$TEXT(VD_ID2,0,'MISERY LOVES COMPANY',
        2       17.0,24.0)
                  SEG_ID5=UIS$BEGIN_SEGMENT(VD_ID2)
                  CALL UIS$TEXT(VD_ID2,0,'ONE SLUMBER INVITES ANOTHER',
        2       1.0,39.0)
                  CALL UIS$NEW_TEXT_LINE(VD_ID2,0)
                  CALL UIS$TEXT(VD_ID2,0,'LIVING WELL IS THE BEST REVENGE')
                  CALL UIS$END_SEGMENT(VD_ID2)
               CALL UIS$END_SEGMENT(VD_ID2)
            CALL UIS$TEXT(VD_ID2,0,'SUCCESS MAKES A FOOL SEEM WISE',
        2       1.0,19.0)
            CALL UIS$END_SEGMENT(VD_ID2)
        CALL UIS$PLOT(VD_ID2,0,20.0,25.0,35.0,25.0,35.0,35.0,20.0,35.0,
        2       20.0,25.0)
        CALL UIS$CIRCLE(VD_ID2,0,10.0,28.0,8.0)
        CALL UIS$END_SEGMENT(VD_ID2)                                    2
        CALL UIS$END_SEGMENT(VD_ID2)
C HOPPING UPWARD ALONG THE SEGMENT PATH
        PRINT 45
 45     FORMAT(T2,'SEGMENT PATH')
        PRINT 55
 55     FORMAT(T1,'------------------------------')
        PRINT 56
 56     FORMAT(T2,'IDENTIFIER',T17,'LEVEL')

        SEG_ID=SEG_ID5              3
        I=5                             4
        PRINT 60,SEG_ID5,I
         DO I=4,1,-1                                 5
        PARENT_ID=UIS$GET_PARENT_SEGMENT(SEG_ID)       6
        SEG_ID=PARENT_ID
        PRINT 60,PARENT_ID,I
 60     FORMAT(T2,I10,T18,I2)
        ENDDO                                       7
C  SEARCHING DOWNWARD THROUGH A NESTED SEGMENT
        PRINT 65
 65     FORMAT(T2,'SEGMENT')
        PRINT 70
 70     FORMAT(T1,'------------------------------')
        PRINT 75
 75     FORMAT(T2,'IDENTIFIER',T17,'OBJECT TYPE')

        NEXT_ID1=UIS$GET_NEXT_OBJECT(SEG_ID2)   8

        DO WHILE(NEXT_ID1 .NE. 0)                 9
        TYPE1=UIS$GET_OBJECT_ATTRIBUTES(NEXT_ID1)
        CALL DETERMINE                            10
        NEXT_ID1=UIS$GET_NEXT_OBJECT(NEXT_ID1,UIS$M_DL_SAME_SEGMENT)     11
        ENDDO                                     12
        WD_ID2=UIS$CREATE_WINDOW(VD_ID2,'SYS$WORKSTATION')

        PAUSE

        END
```

Excluding the root segment, the program HOP contains five levels of nesting. To walk the segment path, start at the innermost segment 3. The counter *I* is initialized to *5* 4, the level of nesting where you start.

A DO loop is declared; the loop ❺ ❼ contains the call to UIS$GET_PARENT_SEGMENT ❻. The **seg_id** argument in UIS$GET_PARENT_SEGMENT is initialized with segment identifier 5 ❸. As each new parent segment identifier is returned, the counter is decremented and, in turn, is used as the **seg_id** argument in the next iteration of the loop.

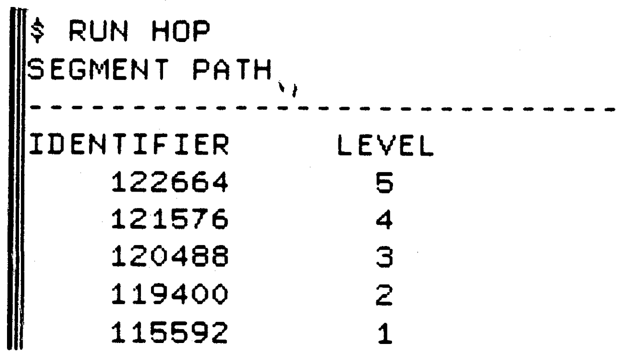
The second purpose of the program is to search a specified segment. To search a segment, use **both** parameters in UIS$GET_NEXT_OBJECT. To start at the beginning of a segment, initialize the **seg_id** to the value of the segment identifier you want to search ❽. When you do this, UIS$GET_NEXT_OBJECT returns the identifier of the next object in the segment. In this example, the second segment is chosen ❶.

Another DO loop is established ❾ ⓬; the loop contains a call to the subroutine DETERMINE. ❿ Note that UIS$GET_NEXT_OBJECT ⓫ now specifies both arguments. The search is performed on the specified segment only. If the flag UIS$M_DL_SAME_SEGMENT is not specified, the search proceeds down to the innermost nested segment.

13.3.3.5 Calling UIS$GET_PARENT_SEGMENT

Segment identifiers are returned beginning with the innermost nested segment as shown in Figure 13–9.

Figure 13–9 Traversing Upward Along the Segment Path



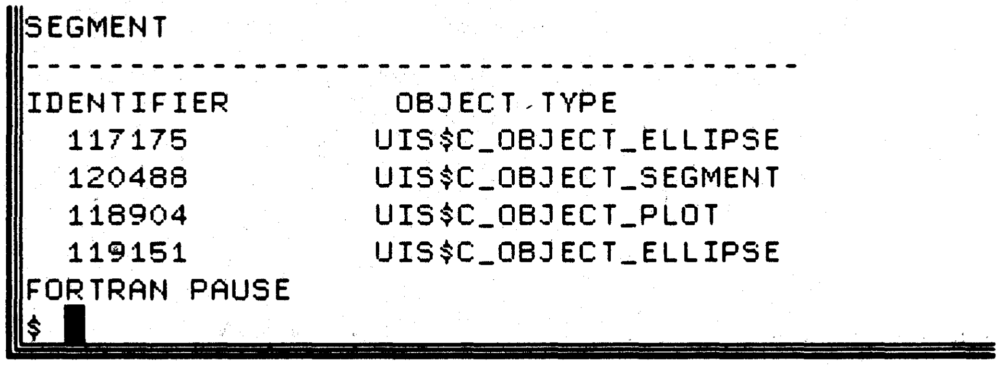
```
$ RUN HOP
SEGMENT PATH
-------------------------------
IDENTIFIER       LEVEL
      122664       5
      121576       4
      120488       3
      119400       2
      115592       1
```

ZK-5295-86

Object identifiers in the second-level segment are displayed as shown in Figure 13–10.

All objects drawn in the virtual display are shown in Figure 13–11.

Figure 13–10 Searching Downward Through a Segment



```
SEGMENT
----------------------------------------
IDENTIFIER         OBJECT-TYPE
  117175          UIS$C_OBJECT_ELLIPSE
  120488          UIS$C_OBJECT_SEGMENT
  118904          UIS$C_OBJECT_PLOT
  119151          UIS$C_OBJECT_ELLIPSE
FORTRAN PAUSE
$
```

ZK-5296-86

13.4 More About Segments

When you use segments in your application programs, you create complex objects that can be edited or searched segment-by-segment. Segments also exhibit special behavior when they encounter attribute blocks.

13.4.1 Programming Options

You can also manipulate segments.

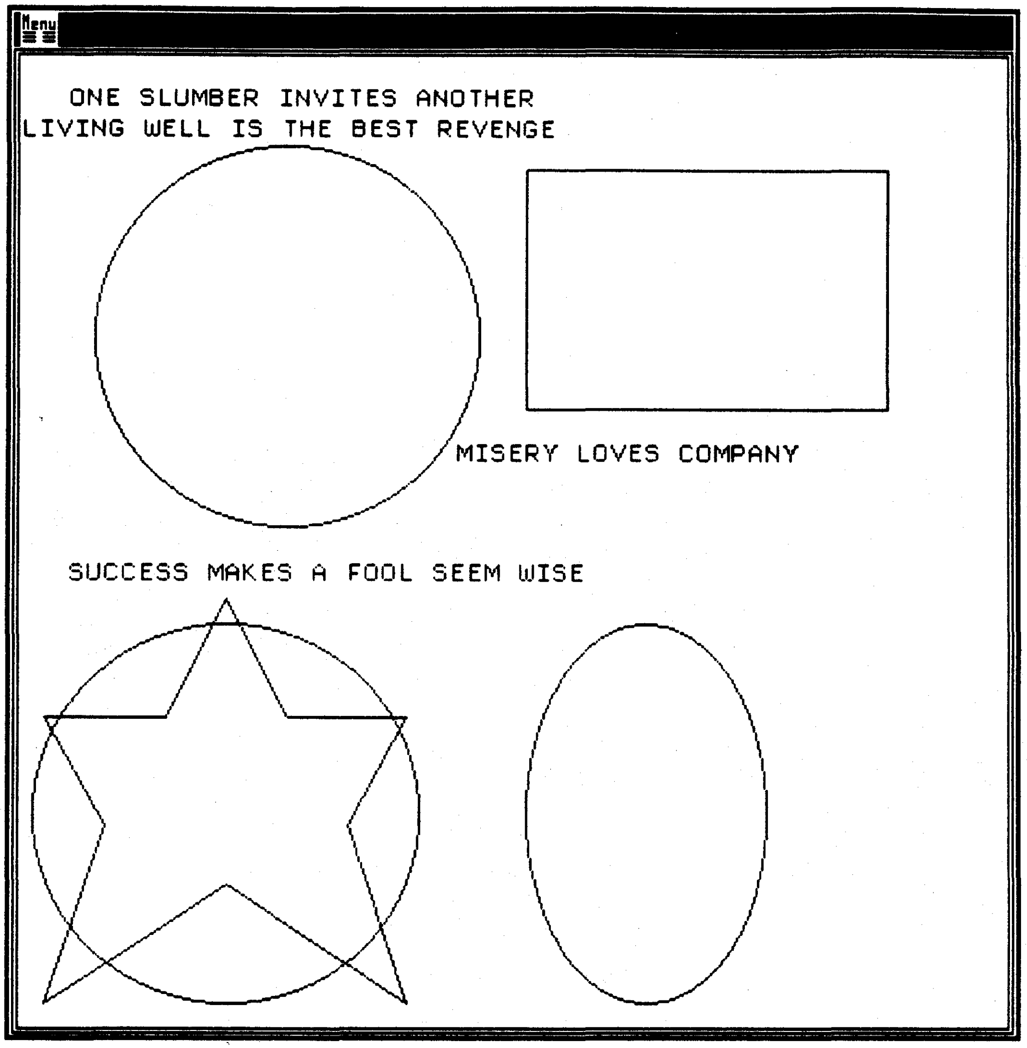
Editing Display Lists

You can edit display lists with or without explicitly defined segments.

NOTE: **Use UIS$SET_INSERTION_POSITION to insert an object between existing objects in a display list.**

The following routines allow you to edit display lists in other ways.

Figure 13–11 Contents of the Display List Drawn in the Virtual Display


Menu
ONE SLUMBER INVITES ANOTHER
LIVING WELL IS THE BEST REVENGE
MISERY LOVES COMPANY
SUCCESS MAKES A FOOL SEEM WISE

ZK 5260 86

Routine	Function
UIS$COPY_OBJECT	Copies an object to another part of the display list
UIS$DELETE_OBJECT	Deletes an object from the display list
UIS$INSERT_OBJECT	Moves an object to another part of the display list
UIS$TRANSFORM_OBJECT	Scales, rotates, and translates an object

Modifying Attribute Blocks Within Segments

A segment can consist of the following:

- Calls to graphics and text output routines
- Attribute routines
- Nested segments

When one attribute block is modified at two different levels of nesting, modifications to the innermost attribute block take precedence over any previous modifications at outer levels. Such attribute block modifications influence graphics and text output (where applicable) at deeper levels of nesting.

When you leave a lower-level nested segment, the original attributes of the parent segment are restored. Therefore, you can change attributes within a segment without affecting a higher-level segment.

13.4.2 Program Development I

Programming Objective

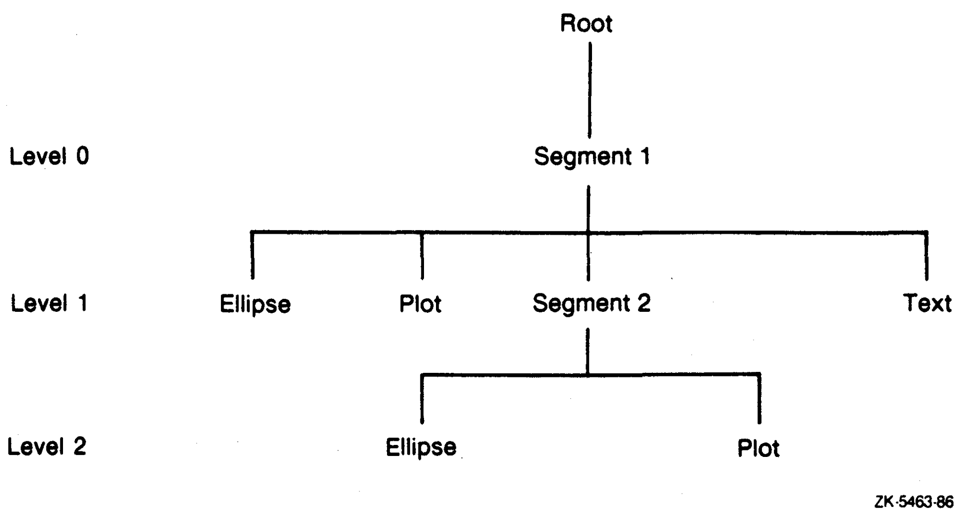
To edit a display list.

Programming Tasks

1. Create a virtual display.
2. Create a series of nested segments containing calls to draw graphic objects.
3. Create a display window and viewport.
4. Delete an object in segment 1.
5. Set the editing pointer to the end of segment 1.
6. Print output headings in the emulation window.
7. Add a line drawing call to the end of segment 1.
8. Verify the contents of segment 1.
9. Position the pointer to the end of segment 2.
10. Add text to segment 2.
11. Verify the contents of segment 2.

Inserting an object in a specific location in the display list affects the order in which objects are drawn in the virtual display, not how an object is drawn. Figure 13-12 shows the pre-edit display list structure in program EDIT_LIST.

Figure 13-12 Pre-Edit Display List Structure



To run program EDIT_LIST, compile subroutine DETERMINE from the program WALK as a separate module and link it with EDIT_LIST.

```
        PROGRAM EDIT_LIST
        IMPLICIT INTEGER(A-Z)
        INCLUDE 'SYS$LIBRARY:UISENTRY'
        INCLUDE 'SYS$LIBRARY:UISUSRDEF'
        COMMON NEXT_ID1,TYPE1

C Create a virtual display
        VD_ID=UIS$CREATE_DISPLAY(1.0,1.0,50.0,50.0,15.0,15.0)
c Create a segment
        SEG_ID1=UIS$BEGIN_SEGMENT(VD_ID)                   1
          CALL UIS$CIRCLE(VD_ID,0,8.0,35.0,7.0)   2
          CURR_ID1=UIS$GET_CURRENT_OBJECT(VD_ID)       3
          CALL UIS$PLOT(VD_ID,0,17.0,27.0,32.0,27.0,24.5,42.0,17.0,27.0)
          CURR_ID2=UIS$GET_CURRENT_OBJECT(VD_ID)
c Create another segment
                SEG_ID2=UIS$BEGIN_SEGMENT(VD_ID)             4
                CALL UIS$ELLIPSE(VD_ID,0,8.0,15.0,5.0,9.0)
                CURR_ID4=UIS$GET_CURRENT_OBJECT(VD_ID)
                CALL UIS$PLOT(VD_ID,0,15.0,8.0,30.0,8.0,
        2       35.0,22.0,20.0,22.0,15.0,8.0)
                CURR_ID5=UIS$GET_CURRENT_OBJECT(VD_ID)
                CALL UIS$END_SEGMENT(VD_ID)
          CALL UIS$TEXT(VD_ID,0,'The ox when weariest treads surest',
        2       5.0,47.0)
          CURR_ID6=UIS$GET_CURRENT_OBJECT(VD_ID)
        CALL UIS$END_SEGMENT(VD_ID)
        WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION')

        PAUSE
c Delete an object from segment 1
           CALL UIS$DELETE_OBJECT(CURR_ID1)      5
```

```
c Set the editing pointer at the end of segment 1
        CALL UIS$SET_INSERTION_POSITION(SEG_ID1,)        ❻
           CALL UIS$PLOT(VD_ID,0,29.0,42.0,44.0,42.0,36.5,27.0,29.0,42.0) ❼

        PRINT 20
 20     FORMAT(T2,'CONTENTS OF SEGMENT 1')
        PRINT 25
 25     FORMAT(T2,'IDENTIFIER',T14,'OBJECT',T22,'TYPE')
        PRINT 30
 30     FORMAT('----------------------------')

c Verify the contents of segment 1
        NEXT_ID1=UIS$GET_NEXT_OBJECT(SEG_ID1)

        DO WHILE(NEXT_ID1 .NE. 0)
        TYPE1=UIS$GET_OBJECT_ATTRIBUTES(NEXT_ID1)
        CALL DETERMINE                                   ❽
        NEXT_ID1=UIS$GET_NEXT_OBJECT(NEXT_ID1,UIS$M_DL_SAME_SEGMENT)
        ENDDO
        PAUSE

c Set the editing pointer at the end of segment 2
        CALL UIS$SET_INSERTION_POSITION(SEG_ID2)       ❾
        CALL UIS$TEXT(VD_ID,0,'Old foxes want no tutors',
        2       5.0,45.0)                                  ❿

        PRINT 40
 40     FORMAT(T2,'CONTENTS OF SEGMENT 2')
        PRINT 45
 45     FORMAT(T2,'IDENTIFIER',T14,'OBJECT',T22,'TYPE')
        PRINT 50
 50     FORMAT('----------------------------')

c Verify the contents of segment 2

        NEXT_ID1=UIS$GET_NEXT_OBJECT(SEG_ID2)

        DO WHILE(NEXT_ID1 .NE. 0)
        TYPE1=UIS$GET_OBJECT_ATTRIBUTES(NEXT_ID1)
        CALL DETERMINE                                   ⓫
        NEXT_ID1=UIS$GET_NEXT_OBJECT(NEXT_ID1,UIS$M_DL_SAME_SEGMENT)
        ENDDO

        PAUSE
        END
```

Two segments are created ❶ ❹. The second segment is nested within the first.

Successive calls to UIS$GET_CURRENT_OBJECT ❸ retrieve an object identifier for each object in both segments. This operation is useful if you need to insert an object in the display list later.

A call to UIS$DELETE_OBJECT ❺ deletes a circle ❷ from segment 1 in the display list.

The editing pointer in the display list is set at the end of segment 1 with UIS$SET_INSERTION_POSITION ❻. A call to UIS$PLOT is added to segment 1 ❼.

A call to the subroutine DETERMINE ❽ verifies the addition in the display list.

The editing pointer in the display list is set at the end of segment 2 with UIS$SET_INSERTION_POSITION ❾. The binary instruction resulting from a call to UIS$TEXT is added to segment 2 ❿.

A call to the subroutine DETERMINE ⓫ verifies the changes in the display list.

Figure 13–13 Post-Edit Structure of the Display List

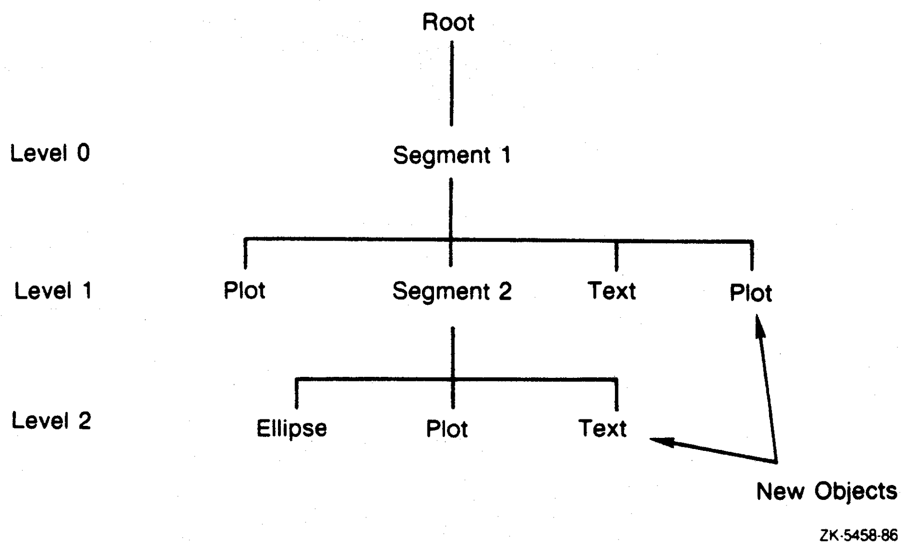


Figure 13–13 shows the post-edit structure of the display list.

13.4.2.1 Calling UIS$SET_INSERTION_POSITION

The original objects, circle, ellipse, triangle, parallelogram, and text, are shown in Figure 13–14.

A triangle and a line of text are added to the virtual display. The circle is deleted from the virtual display as shown in Figure 13–15.

The contents of the segment are written to the emulation window as shown in Figure 13–16.

13.4.2.2 Program Development II

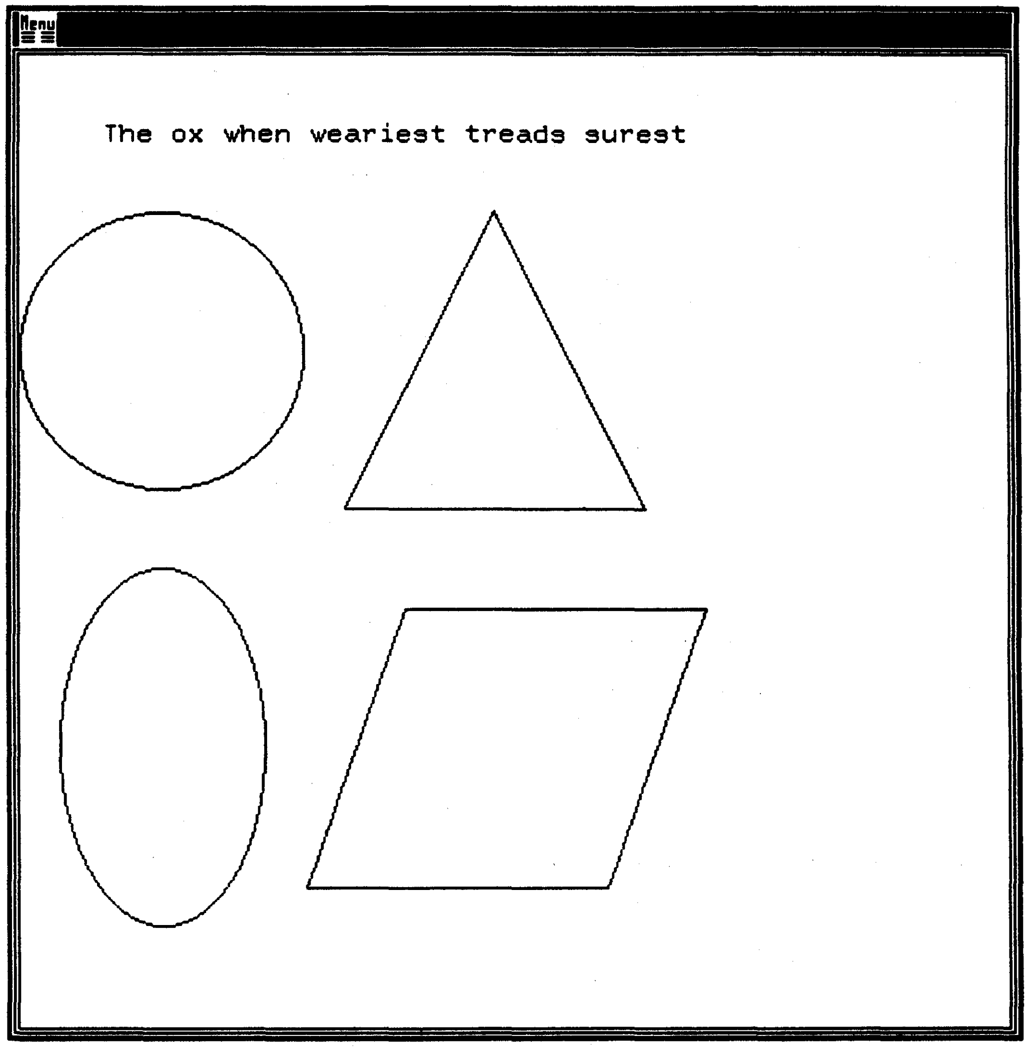
Programming Objective

To draw text at different levels of segmentation.

Programming Tasks

1. Create a virtual display.
2. Create a display window and viewport.
3. Create three levels of nested segments.
4. Modify the font character spacing attributes for each level of nesting.

Figure 13-14 Before Display List Modification



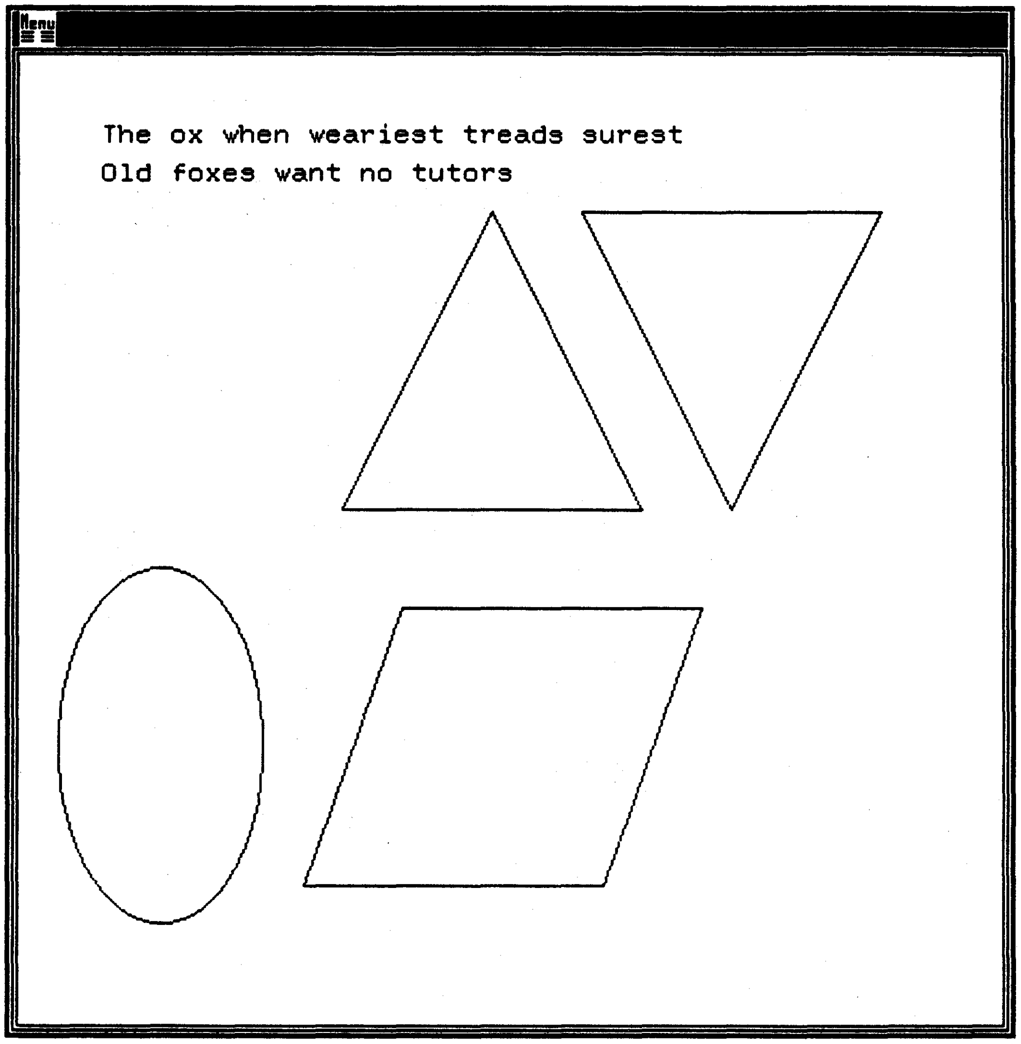
ZK-5261-86

5 Draw text at each level of nesting.

Font names specified in the program are logical names.

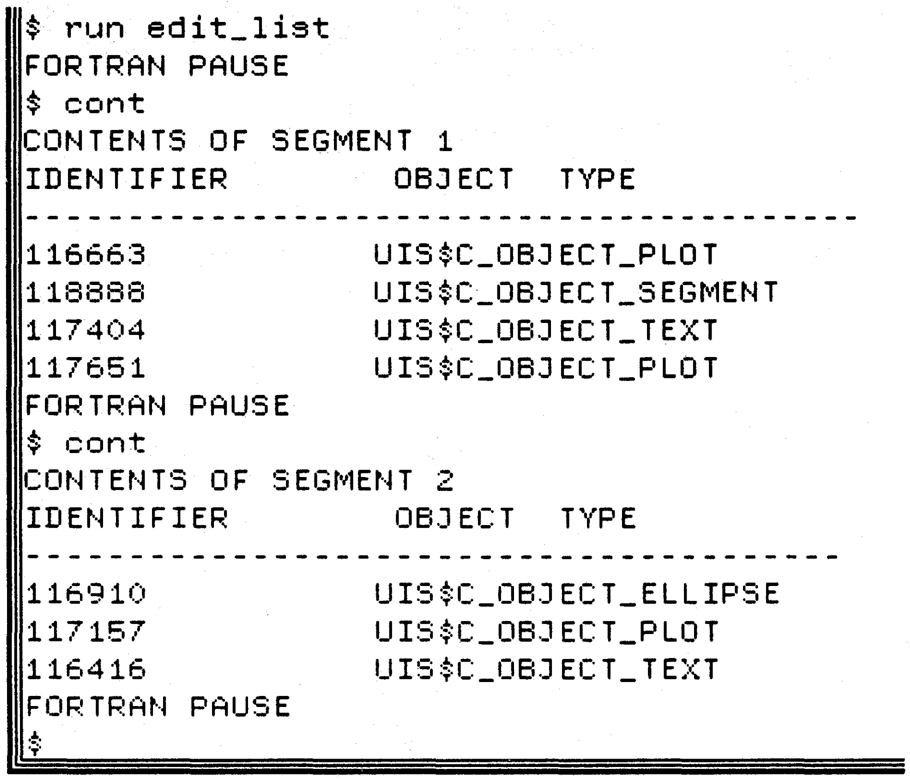
```
PROGRAM SEGMENT
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
VD_ID=UIS$CREATE_DISPLAY(0.0,0.0,30.0,30.0,21.0,5.0)
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION')

CALL UIS$BEGIN_SEGMENT(VD_ID)  ❶
    CALL UIS$SET_FONT(VD_ID,0,1,'MY_FONT_6')  ❷
    CALL UIS$SET_CHAR_SPACING(VD_ID,1,1,0.0,1.0)  ❸
    CALL UIS$TEXT(VD_ID,1,'The resolved mind has no cares',0.0,30.0
```

Figure 13–15 Executing the Modified Display List


The ox when weariest treads surest
Old foxes want no tutors

```
        CALL UIS$BEGIN_SEGMENT(VD_ID)          5
            CALL UIS$SET_FONT(VD_ID,1,1,'MY_FONT_13')  6
            CALL UIS$NEW_TEXT_LINE(VD_ID,1)
            CALL UIS$TEXT(VD_ID,1,'The camel never sees its own hump')
                CALL UIS$BEGIN_SEGMENT(VD_ID)  8
                  CALL UIS$SET_FONT(VD_ID,1,1,'MY_FONT_7')  9
                  CALL UIS$NEW_TEXT_LINE(VD_ID,1)
                  CALL UIS$TEXT(VD_ID,1,'First things first')
                CALL UIS$END_SEGMENT(VD_ID)    10
    PAUSE
            CALL UIS$SET_CHAR_SPACING(VD_ID,1,1,0.0,0.0)  11
            CALL UIS$NEW_TEXT_LINE(VD_ID,1)
            CALL UIS$TEXT(VD_ID,1,'A new broom sweeps clean')     12
        CALL UIS$END_SEGMENT(VD_ID)  13
    CALL UIS$NEW_TEXT_LINE(VD_ID,1,)
    CALL UIS$TEXT(VD_ID,1,'No sun without a shadow')  14
    CALL UIS$END_SEGMENT(VD_ID)     15
```

Figure 13–16 Verifying the Contents of the Display List



```
$ run edit_list
FORTRAN PAUSE
$ cont
CONTENTS OF SEGMENT 1
IDENTIFIER          OBJECT  TYPE
--------------------------------------
116663             UIS$C_OBJECT_PLOT
118888             UIS$C_OBJECT_SEGMENT
117404             UIS$C_OBJECT_TEXT
117651             UIS$C_OBJECT_PLOT
FORTRAN PAUSE
$ cont
CONTENTS OF SEGMENT 2
IDENTIFIER          OBJECT  TYPE
--------------------------------------
116910             UIS$C_OBJECT_ELLIPSE
117157             UIS$C_OBJECT_PLOT
116416             UIS$C_OBJECT_TEXT
FORTRAN PAUSE
$
```

ZK-5262-86

```
        PAUSE
        END
```

The first call to UIS$BEGIN_SEGMENT [1] and the final call to UIS$END_SEGMENT [15] establish the limits of the first-level segment. In this segment, there are two calls to UIS$TEXT [4] [14]. The first call to UIS$TEXT establishes the current position for all first-level text output.

An attribute routine UIS$SET_FONT is called [2] to modify the font attribute. The font MY_FONT_6 is now the current font for all text output in the first-level segment. First-level text is drawn with MY_FONT_6.

The calls to UIS$BEGIN_SEGMENT and UIS$END_SEGMENT [5] [13] establish the limits of the second-level segment nested within the first-level segment. The first call to UIS$SET_FONT [6] in the second-level segment references the same output attribute block number specified in the attribute routine call in the first-level segment [2]. The modifications to attribute block 1 at the second level take precedence over any previous modifications of attribute block 1 at outer levels.

The second-level segment further modifies the font attribute [6]. The font MY_FONT_13 is now the current font for all text output in this second-level segment. The first call to UIS$TEXT within the second-level segment [7] establishes the current position for text output drawn at the second level.

Calls to UIS$TEXT within this segment reference the same attribute block 1.

Once again, calls to UIS$BEGIN_SEGMENT and UIS$END_SEGMENT 8 10 establish the limits of the third level of segmentation nested within the second level. The font MY_FONT_7 is now the current font for all text output in this segment 9.

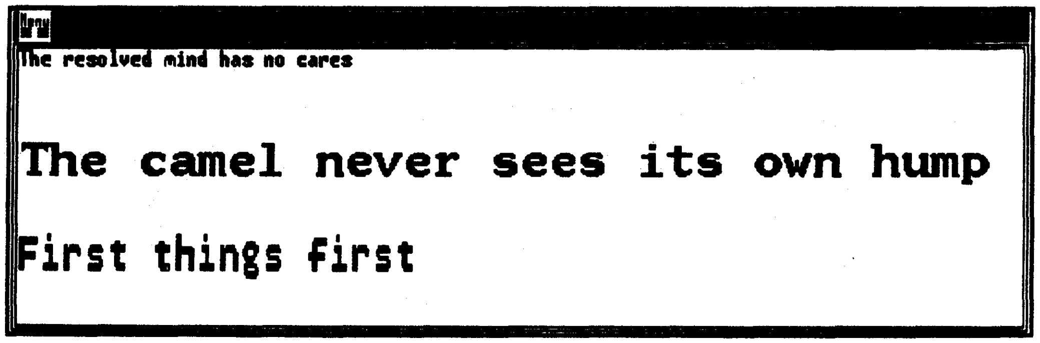
The line-spacing component of the character-spacing attribute is modified twice 3 11. The first call to UIS$SET_CHAR_SPACING increases the line spacing by a factor of *1.* As the program executes, the second text drawing routine call in levels 1 and 2 14 12 require room to avoid overstriking existing lines.

NOTE: **You must invoke the indirect command file SYS$EXAMPLES:DEFFONT.COM before you run the demonstration programs.**

13.4.2.3 Calling UIS$BEGIN_SEGMENT and UIS$END_SEGMENT

As the program SEGMENT sequentially executes each instruction, a text string is drawn in the virtual display at the first, second, and third levels of segmentation as shown in Figure 13–17. Note the font used in text creation.

Figure 13–17 Text Output During Execution

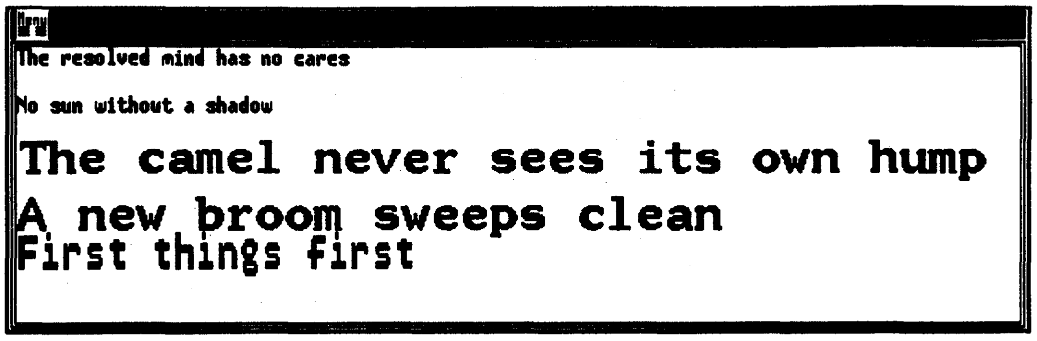


```
The resolved mind has no cares

The camel never sees its own hump

First things first
```

ZK-4559-85

Text strings are created in the reverse order of segmentation—second level and then first level. Note the font used and the order of text string creation shown in Figure 13–18 as compared with the statements in the source program.

Figure 13–18 Final Text Output


The resolved mind has no cares

No sun without a shadow

The camel never sees its own hump

A new broom sweeps clean

First things first

ZK-4560-85

14 Geometric and Attribute Transformations

14.1 Overview

Transformations change the appearance of graphic objects and text. Part I discussed transformations and their possibly distorting effects on graphic objects. In Part II, you have seen the effects of world coordinate transformations when you modify world coordinate space, then redraw graphic objects in the new space. This chapter describes the following types of transformations:

- Two-dimensional geometric transformations
- Attribute transformations

14.2 Geometric Transformations

Two-dimensional geometric transformation of a graphic object involves changing the graphic object angular orientation or shape within the virtual display. It does not modify the coordinate system. Scaling, translation, and rotation transform graphic objects geometrically.

14.2.1 Translating Graphic Objects

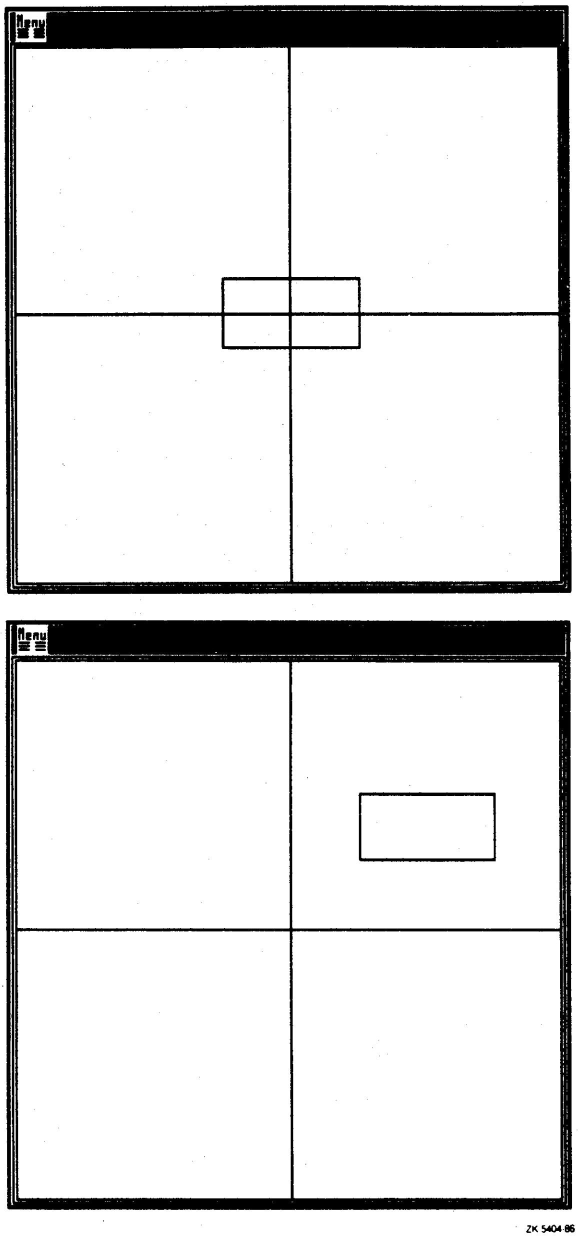
When you translate a graphic object, you move the object to another part of the coordinate space without altering its *x* and *y* axis physical orientation. For example, if a side of a triangle was originally parallel to the *y* axis, it remains parallel to that axis even if the object is moved to another quadrant in the coordinate space. Figure 14–1 shows graphic object translation.

14.2.2 Scaling Graphic Objects

When you scale a graphic object, you stretch or shrink it. There are two types of scaling:

- Simple scaling
- Complex scaling

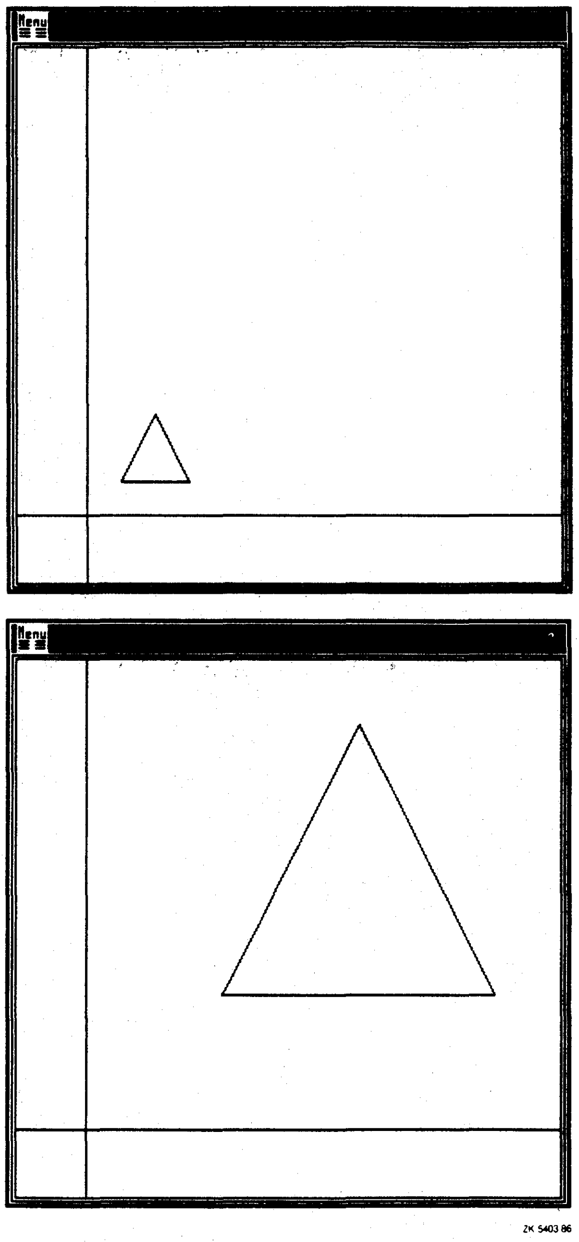
Figure 14–1 Translating a Graphic Object



Simple Scaling of Graphic Objects

When you perform simple scalilng, you execute a single transformation. The position of the newly scaled graphic object in the virtual display is always different from its original position, with one exception: if the object center point is at the origin, the object will not move when scaled. Figure 14–2 shows simple scaling.

Figure 14–2 Simple Scaling



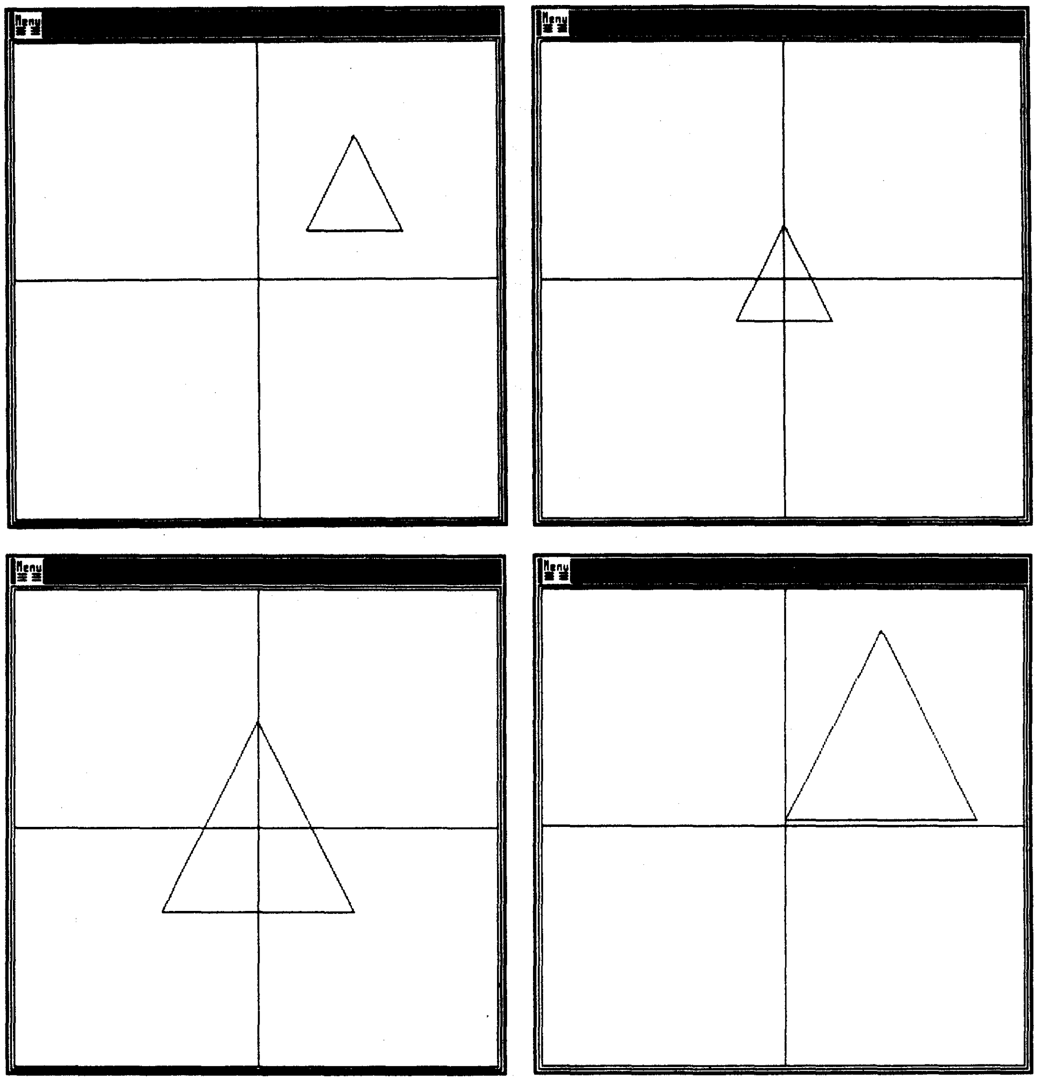
Complex Scaling of Graphic Objects

When you perform complex scaling, the newly scaled object maintains its previous position in the virtual display. Complex scaling involves the following steps:

1. Translate the center of the object to the coordinate system origin.
2. Scale the object.
3. Translate the object to its original position.

Figure 14–3 shows complex scaling.

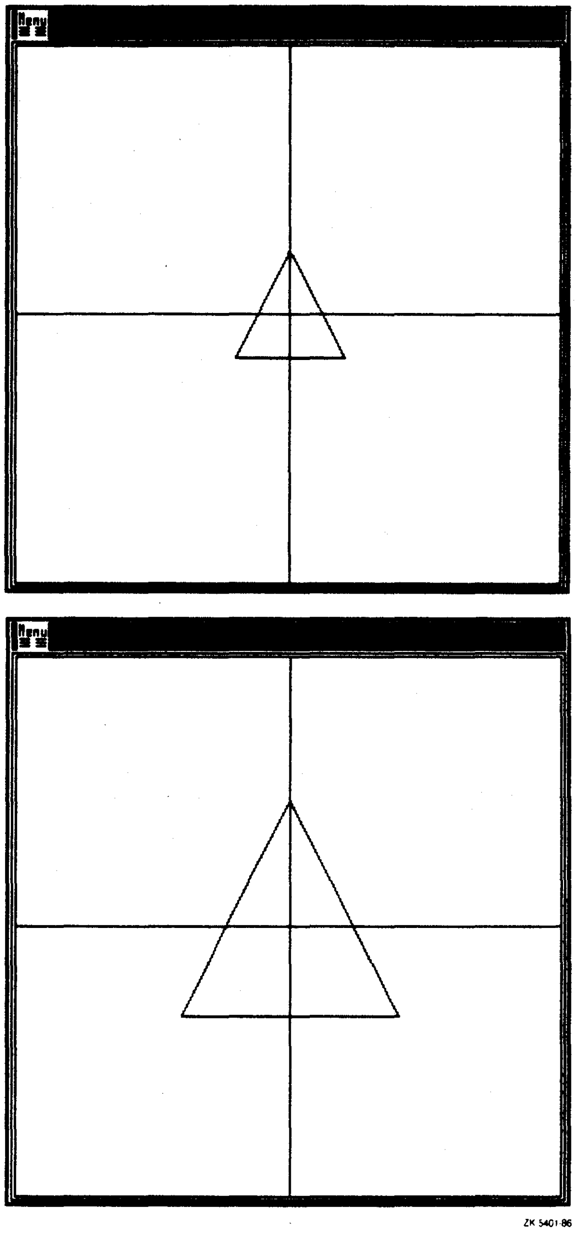
Figure 14–3 Complex Scaling



14.2.2.1 Uniformly Scaled Graphic Objects

Compare uniform scaling to a photographic enlargement of a snapshot. The enlargment renders an object with physical dimensions proportional to the original snapshot. The scaling factor of the width of the object, S_x, equals the scaling factor of the height of the object, S_y. Figure 14–4 shows a uniformly scaled object.

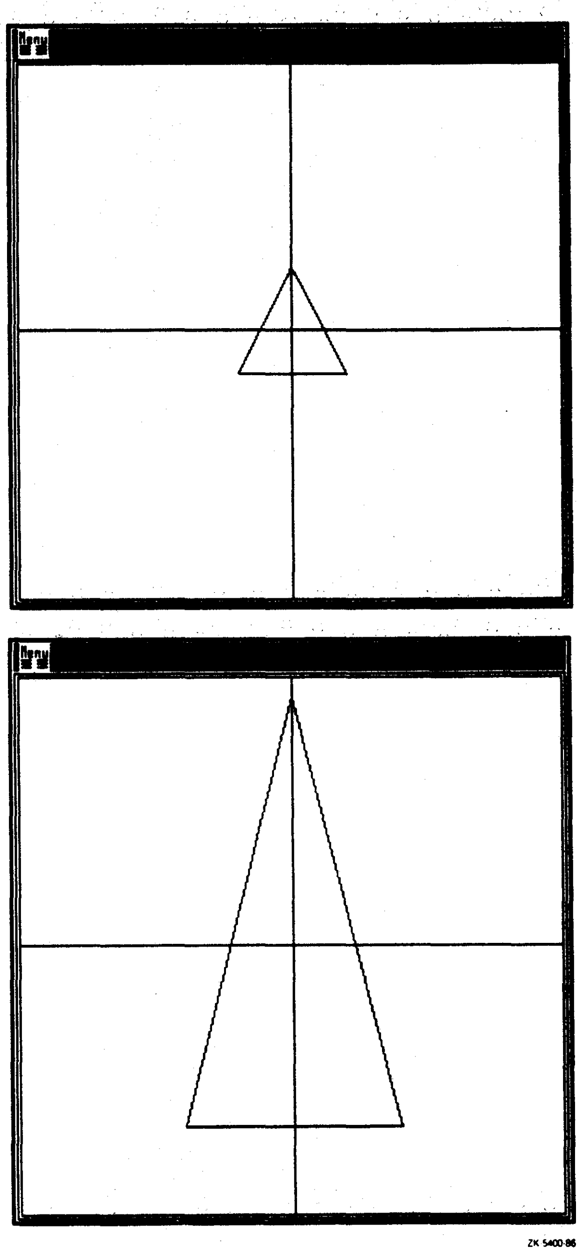
Figure 14–4 Uniformly Scaling a Graphic Object



14.2.2.2 Differentially Scaled Graphic Objects

The height of an object can be increased, while its width remains constant where s_x does not equal s_y. The object is differentially scaled as shown in Figure 14–5.

Figure 14-5 Differentially Scaling a Graphic Object



14.2.3 Rotating Graphic Objects

Generally speaking, rotation changes an object angular orientation in the virtual display. Rotation occurs about the origin of the coordinate system. Positive rotation is a counterclockwise movement.

Simple Rotation of Graphic Objects

Simple rotation involves executing a single transformation—no translation. In simple rotation, the object appears to revolve about the origin. Figure 14–6 shows a rectangle rotating about the origin.

Figure 14–6 Simple Rotation of a Graphic Object

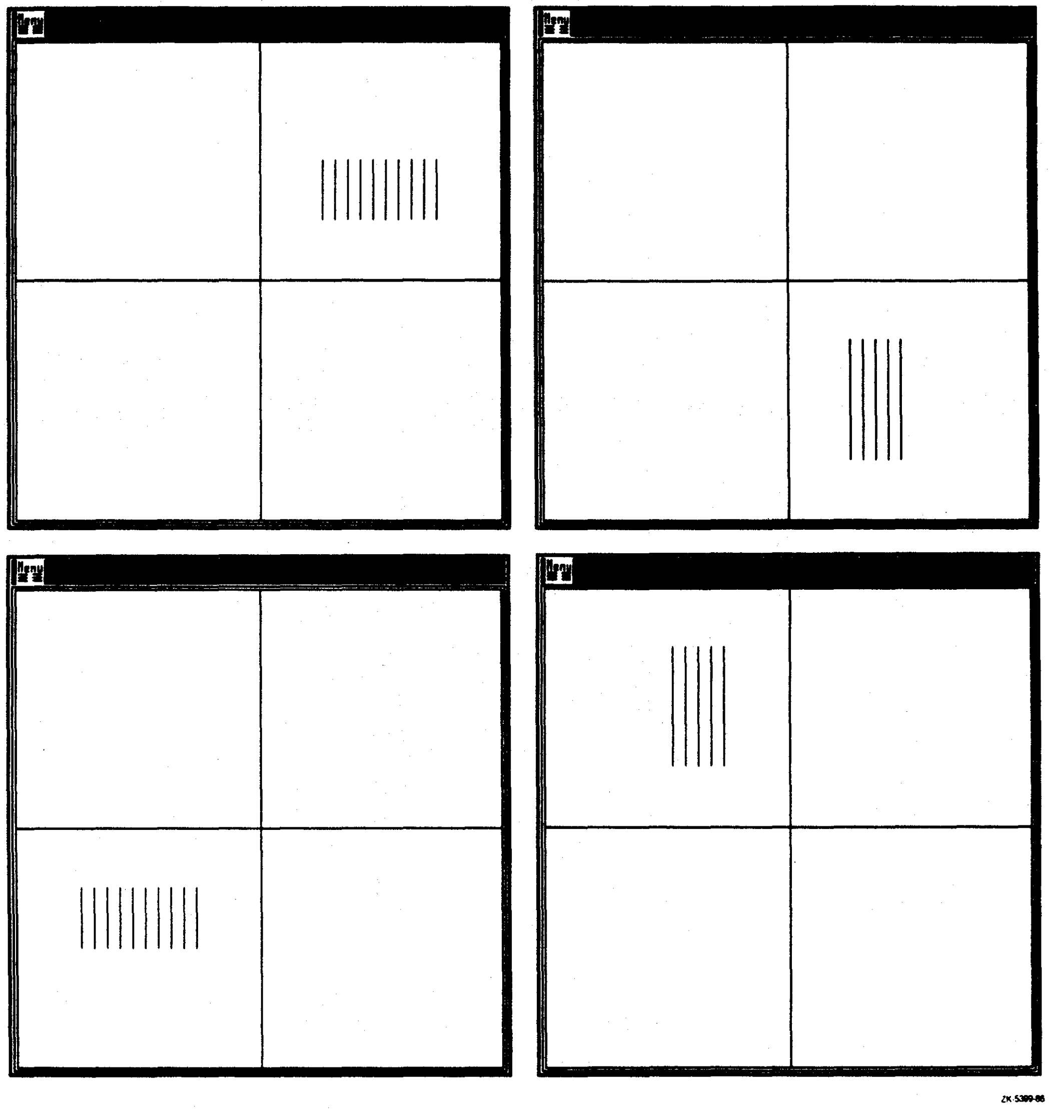
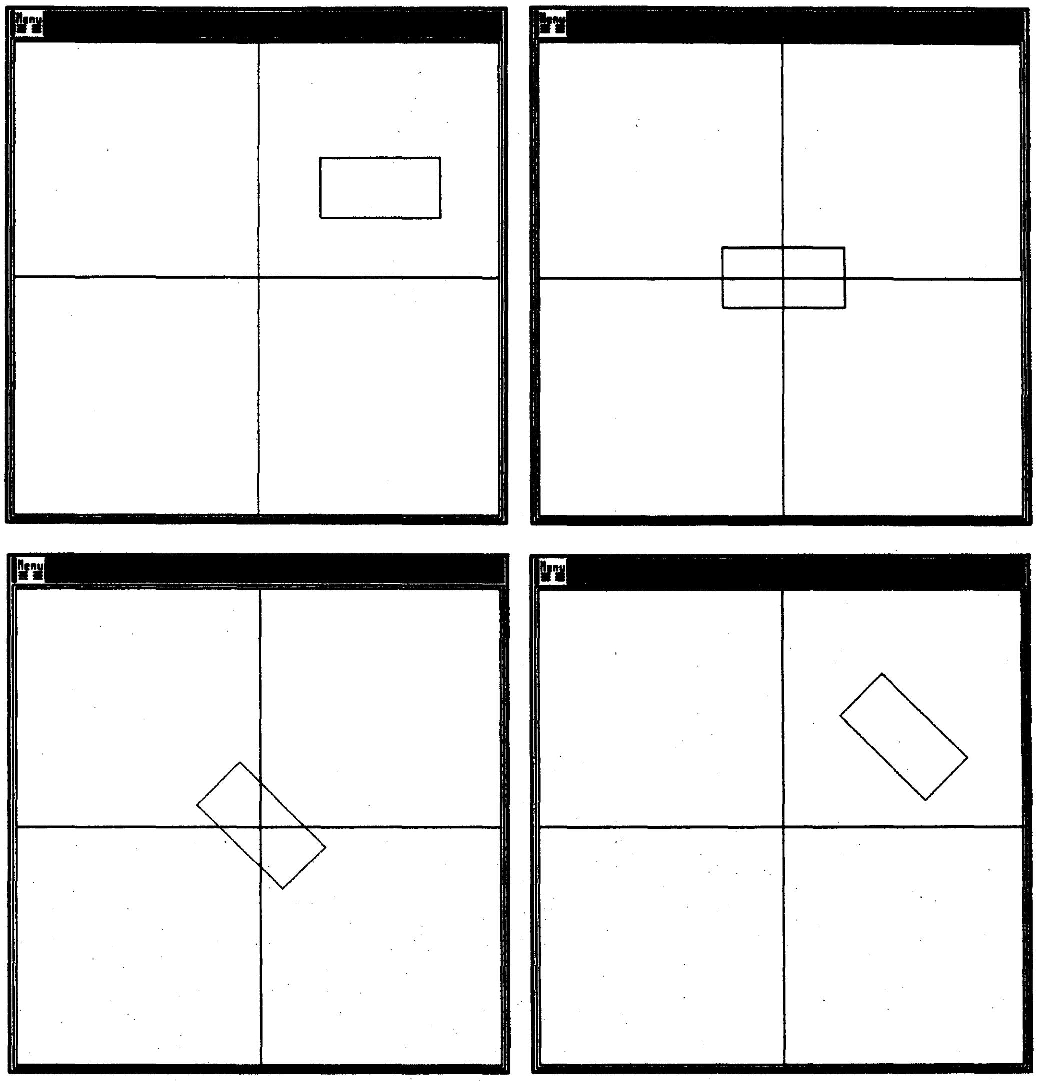


Figure 14–7 Complex Rotation of a Rectangle



Complex Rotation of Graphic Objects

To perform complex rotation, you translate the object to the origin so the origin and reference point share the same coordinate values—(0.0,0.0). The object is rotated and translated to its original position in the virtual display. Figure 14–7 illustrates complex rotation of a rectangle.

14.2.4 Programming Options

There are two types of geometric transformation:

- COPY
- MOVE

Two-Dimensional Geometric Transformation—COPY

You can execute a geometric transformation when you use UIS$COPY_OBJECT to copy the graphic object. The original object remains unchanged.

Two-Dimensional Geometric Transformations—MOVE

You can execute a geometric transformation when you use UIS$TRANSFORM_OBJECT to transform the graphic object in the virtual display. The original object is modified.

14.2.5 Program Development I

Programming Objective

To rotate a graphic object counterclockwise 45 degrees about its center.

Programming Tasks

1. Create a virtual display.
2. Create a display window and viewport.
3. Create a graphic object and obtain its identifier.
4. Declare and load a two-dimensional array with translation values.
5. Execute translation.
6. Load array with rotation values.
7. Execute rotation.
8. Load array with translation values.
9. Execute the translation where the original object is erased and redraw the object in its original position in the coordinate system.

```
PROGRAM GEO_TRANSFORM_ROT
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
REAL*4 MATRIX(2,3)            1

VD_ID=UIS$CREATE_DISPLAY(-20.0,-20.0,20.0,20.0,10.0,10.0)
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION')

CALL UIS$PLOT(VD_ID,0,0.0,20.0,0.0,-20.0)    2
CALL UIS$PLOT(VD_ID,0,-20.0,0.0,20.0,0.0)    3

CALL UIS$PLOT(VD_ID,0,5.0,5.0,15.0,5.0,15.0,10.0,5.0,10.0,
2       5.0,5.0)                                 4

CURRENT_ID=UIS$GET_CURRENT_OBJECT(VD_ID)    5
OBJ_ID=CURRENT_ID

PAUSE
```

```
MATRIX(1,1)=1.0                    ❻
MATRIX(2,1)=0.0
MATRIX(1,2)=0.0
MATRIX(2,2)=1.0
MATRIX(1,3)=-10.0
MATRIX(2,3)=-7.5
CALL UIS$TRANSFORM_OBJECT(OBJ_ID,MATRIX)   ❼

PAUSE

MATRIX(1,1)=COSD(45.0)                   ❽
MATRIX(2,1)=-SIND(45.0)
MATRIX(1,2)=SIND(45.0)
MATRIX(2,2)=COSD(45.0)
MATRIX(1,3)=0.0
MATRIX(2,3)=0.0
CALL UIS$TRANSFORM_OBJECT(OBJ_ID,MATRIX)  ❾

PAUSE
MATRIX(1,1)=1.0                          ❿
MATRIX(2,1)=0.0
MATRIX(1,2)=0.0
MATRIX(2,2)=1.0
MATRIX(1,3)=10.0
MATRIX(2,3)=7.5
CALL UIS$TRANSFORM_OBJECT(OBJ_ID,MATRIX)   ⓫

PAUSE
 END
```

A two-dimensional array is declared ❶.

The *x* and *y* axes are drawn ❷ ❸.

A rectangle is drawn using UIS$PLOT ❹. Call UIS$GET_CURRENT_OBJECT to save its object identifier ❺. The object identifier is used as an argument to the transformation routine.

The rectangle is rotated about its center.

The VAX FORTRAN intrinsic functions SIND and COSD accept degrees as arguments ❽.

The matrix is loaded with values three times ❻ ❽ ❿ to translate, rotate the rectangle about its center, then translate it to its original position in the virtual display.

Each transformation is performed as the original object is erased and redrawn in its new orientation. The rectangle is redrawn with each call to UIS$TRANSFORM_OBJECT ❼ ❾ ⓫.

14.2.6 Calling UIS$TRANSFORMATION_OBJECT

The program GEO_TRANSFORM_ROT translates, rotates, and translates a rectangle with UIS$TRANSFORM_OBJECT. Figure 14-7 illustrates how after each transformation the previous position of the rectangle in the virtual display is erased.

14.2.7 Program Development II

Programming Objectives

To rotate a copy of the graphic object 45 degrees about its center and place the rotated copy in another quadrant.

Programming Tasks

1. Create a virtual display.
2. Create a display window and viewport.
3. Declare and load a two-dimensional array with translation values.
4. Execute the COPY operation and the translation.
5. Load the array with rotation values.
6. Execute rotation.
7. Load the array with translation values.
8. Execute translation.

```
PROGRAM COPY_OBJECT
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
REAL*4 MATRIX(2,3)

VD_ID=UIS$CREATE_DISPLAY(-20.0,-20.0,20.0,20.0,10.0,10.0)
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION')

CALL UIS$PLOT(VD_ID,0,0.0,20.0,0.0,-20.0)
CALL UIS$PLOT(VD_ID,0,-20.0,0.0,20.0,0.0)

CALL UIS$PLOT(VD_ID,0,5.0,5.0,15.0,5.0,10.0,10.0,5.0,5.0)

CURRENT_ID=UIS$GET_CURRENT_OBJECT(VD_ID)
OBJ_ID=CURRENT_ID

PAUSE

MATRIX(1,1)=1.0
MATRIX(2,1)=0.0
MATRIX(1,2)=0.0
MATRIX(2,2)=1.0
MATRIX(1,3)=-10.0
MATRIX(2,3)=-7.5
COPY_ID=UIS$COPY_OBJECT(OBJ_ID,MATRIX)    1

PAUSE

OBJ_ID=COPY_ID                            2

MATRIX(1,1)=COSD(45.0)       3
MATRIX(2,1)=-SIND(45.0)
MATRIX(1,2)=SIND(45.0)
MATRIX(2,2)=COSD(45.0)
MATRIX(1,3)=0.0
MATRIX(2,3)=0.0
CALL UIS$TRANSFORM_OBJECT(OBJ_ID,MATRIX)  4

PAUSE
```

```
MATRIX(1,1)=1.0
MATRIX(2,1)=0.0
MATRIX(1,2)=0.0
MATRIX(2,2)=1.0
MATRIX(1,3)=-10.0                          5
MATRIX(2,3)=7.5
CALL UIS$TRANSFORM_OBJECT(OBJ_ID,MATRIX)

PAUSE
END
```

Except for a few important differences, this program is identical to the previous program GEO_TRANSFORM_ROT.

The first transformation is executed 1. The triangle is copied and translated to the origin of the coordinate space. The coordinates of the center of the triangle match those of the origin. The original triangle in the first quadrant remains unchanged.

The identifier of the transformed object *copy_id* is assigned to the *obj_id* 2. The identifier is used as an argument in the next transformation.

The VAX FORTRAN intrinsic functions SIND and COSD accept degrees as arguments 3.

A call to UIS$TRANSFORM_OBJECT rotates the translated triangle 45 degrees 4. The original object is erased and redrawn in its new orientation.

The final translation of the triangle places it in the second quadrant at a 45-degree angle to the original triangle 5.

14.2.8 Calling UIS$COPY_OBJECT

Transformation of the triangle is similar to that of the rectangle in the previous example. However, the first transformation copies the triangle. Figure 14-8 shows that the triangle remains in the virtual display. However, the rotated copy of the triangle is translated to the second quadrant.

14.3 Attribute Transformations

An attribute transformation occurs when you modify graphic objects and text, but you do not have to know the attribute block of the original objects.

14.3.1 Programming Options

Attribute Transformations

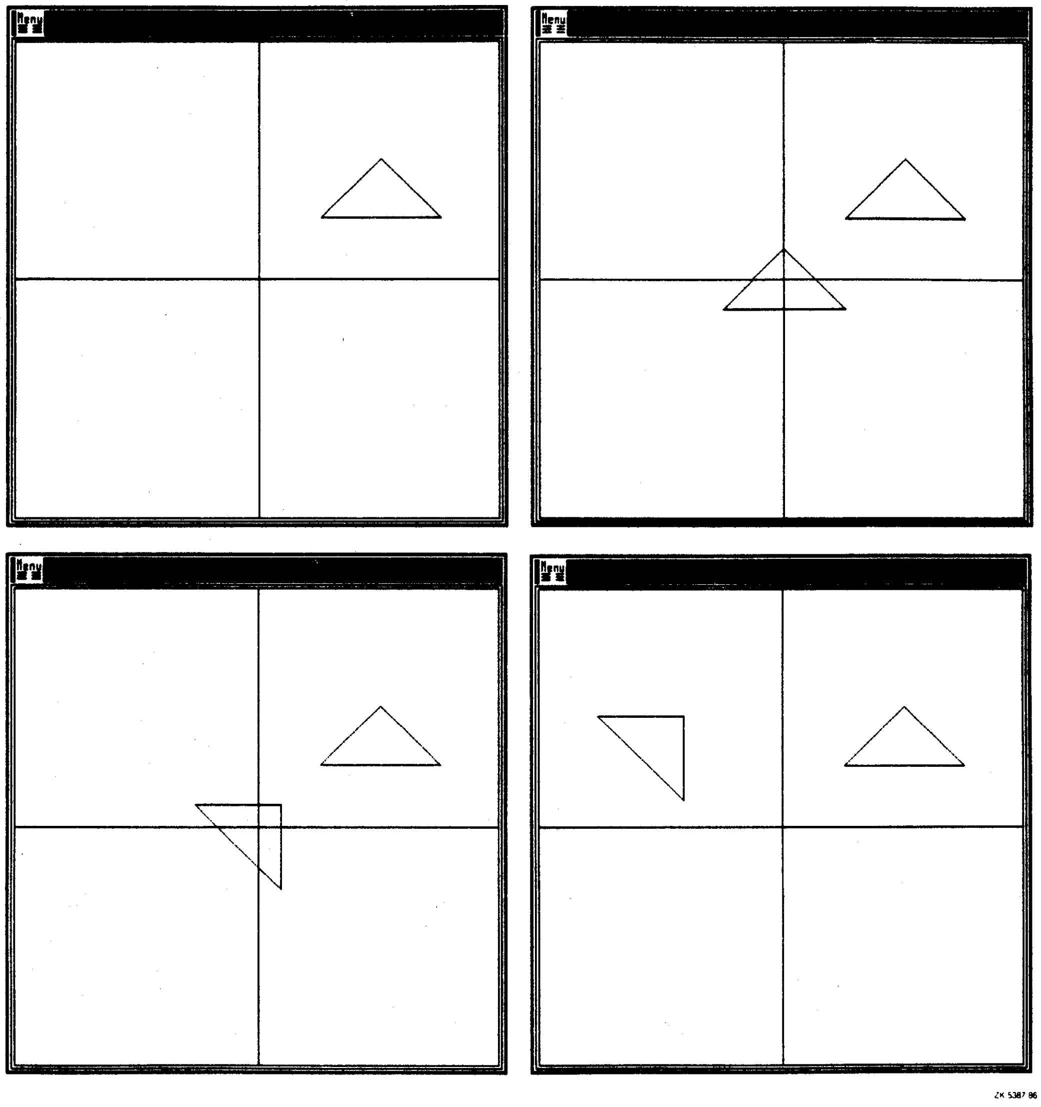
Ordinarily, when you modify the appearance of an existing graphic object, you must perform the follow procedure:

1 Obtain the object identifier.

2 Call UIS$DELETE_OBJECT with the object identifier.

3 Redraw the graphic object or text using the modified attribute block.

Figure 14–8 Complex Rotation of a Triangle



The above procedure requires at least two steps:

- Use UIS$ERASE to erase the virtual display.
- Redraw the object with a modified attribute block.

To modify the attributes of graphic objects and text in a single call, call UIS$COPY_OBJECT or UIS$TRANSFORM_OBJECT and specify the **atb** argument but omit the **matrix** argument. To disable attribute transformation, omit the **atb** argument.

14.3.2 Program Development

Programming Objective

To modify the fill pattern of a circle as a transformation.

Programming Tasks

1 Create a virtual display.

2 Create a display window and a display viewport.

3 Draw a circle using default attributes.

4 Obtain its object identifier.

5 Modify the fill pattern attribute.

6 Transform the circle attributes and draw the modified circle.

```
PROGRAM ATTR_TRANS
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'

VD_ID=UIS$CREATE_DISPLAY(-10.5,-10.5,10.5,10.5,10.0,10.0)
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION')

CALL UIS$CIRCLE(VD_ID,0,0.0,0.0,10.0)
CURRENT_ID=UIS$GET_CURRENT_OBJECT(VD_ID)
OBJ_ID=CURRENT_ID

CALL UIS$SET_FONT(VD_ID,0,1,'UIS$FILL_PATTERNS')  ❶
CALL UIS$SET_FILL_PATTERN(VD_ID,1,1,PATT$C_DOWNDIAG1_7) ❷

PAUSE

CALL UIS$TRANSFORM_OBJECT(OBJ_ID,,1)  ❸

PAUSE
END
```

This program does not declare a matrix. Therefore, the position of any objects will be the same.

The fill pattern attribute is modified ❶ ❷.

The object identifier of the original circle and attribute block number of the newly modified attribute block are arguments in the transformation ❸.

14.3.3 Requesting Attribute Transformations

Because no matrix is specified in the transformation, the resulting transformation does not change object positions within the virtual display. The original circle is erased and the modified circle is placed in its position as shown in Figure 14–9.

If you call UIS$COPY_OBJECT rather than UIS$TRANSFORM_OBJECT, the original circle remains visible in the virtual display. The modified circle is still in the same position. Figure 14–10 shows attributes modified with a copy.

Figure 14–9 Modifying Attributes with a Transformation

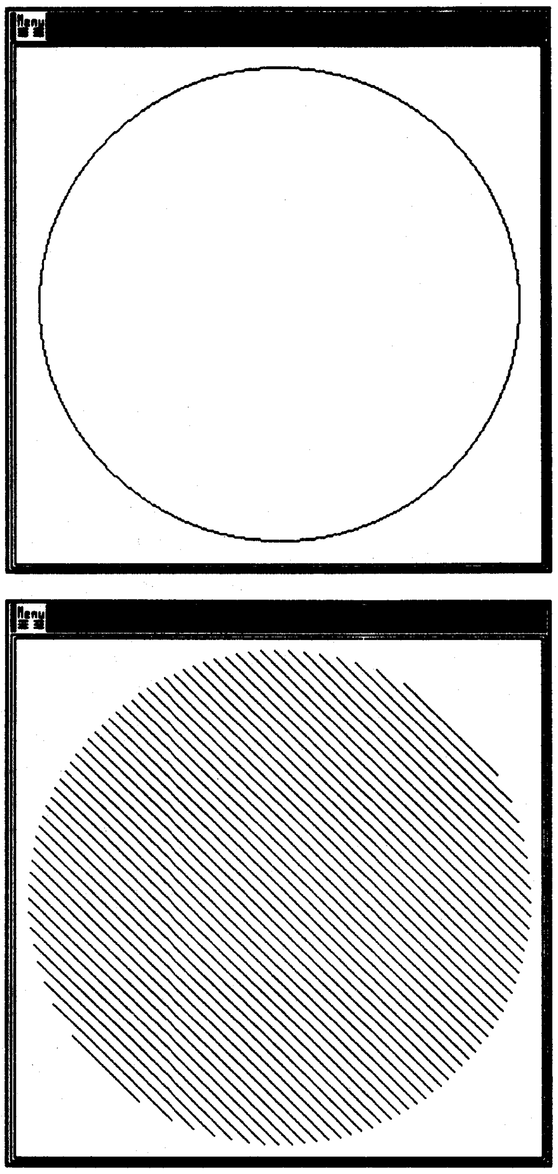
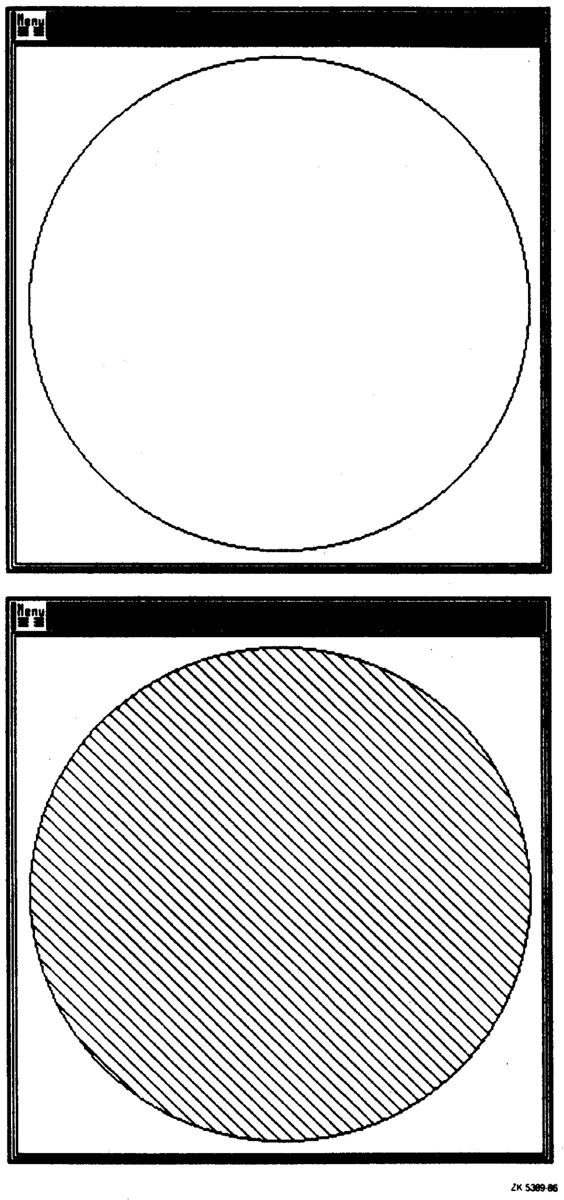


Figure 14–10 Modifying Attributes with a Copy



15 Metafiles and Private Data

15.1 Overview

If you want to reuse a display you produce, you must first store it in a UIS metafile. This chapter details metafile structure and the contents of the binary encoded instructions.

An additional feature allows you to associate data with graphic objects. You can specify a particular graphic object or group of objects within the display to be associated with the user-defined data. This chapter discusses the following topics:

- Extracting data from a display list
- Interpreting the user buffer
- Creating a UIS metafile
- Creating private data

Hardcopy UIS (HCUIS) translates UIS pictures to other formats. See the *VMS Workstation Software Guide to Printing Graphics* for more information about HCUIS.

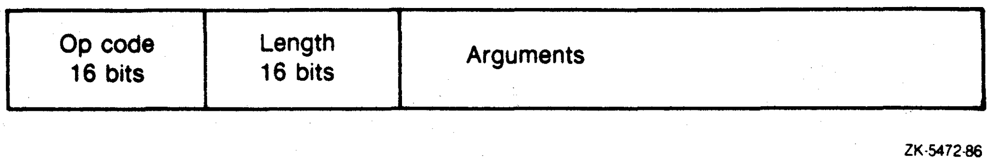
15.2 Display Lists and UIS Metafiles

You design application programs to generate graphic objects on the screen. You should also be concerned with program modularity and efficiency. A new entry is added to the display list for each new object drawn in the virtual display. You should preserve the contents of a display list as *generically encoded* binary instructions for use across many applications. Then you can extract graphics output and attribute modifications from display lists, store them in user-defined buffers as *metafile components,* and store them in files as *metafiles.*

15.2.1 Generic Encoding of Graphics and Attribute Routines

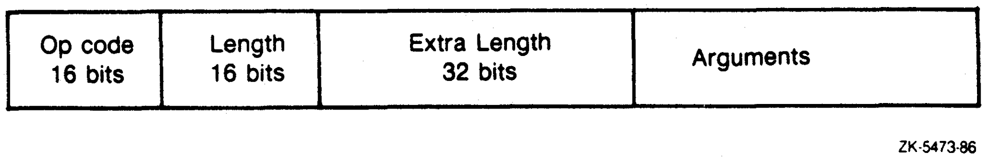
When you draw an object in the virtual display or modify an attribute, a binary encoded instruction is added to the display list of the specified virtual display. Entries in the display list are variable length instructions, encoded as shown in Figure 15–1.

Figure 15–1 Binary Encoded Instruction



If the length of the binary encoded instruction is greater than 32,767 bytes, set the length field equal to GER$C_LENGTH_DIFF and set the extra length equal to the total number of bytes in the instruction. Figure 15–2 describes the format of a display list entry if the length field is greater than 32,767 bytes.

Figure 15–2 Extended Binary Encoded Instruction



15.2.1.1 Normalized Coordinates

The coordinate system used in display lists and to create generically encoded streams is known as *normalized coordinates*. Normalized coordinates are floating point numbers in the range (0.0,0.0) to (max_nc_x,max_nc_y), where (0.0,0.0) refers to lower-left corner of the virtual display and (max_nc_x,max_nc_y) refers to the upper-right corner.

UIS uses normalized coordinates to defer the actual mapping of application world coordinates to device-specific coordinates until the actual output device is known. For example, the device coordinates of a printer might be different from the device coordinates of a raster display.

15.2.1.2 Interpreting the User Buffer

When UIS routine calls are executed, binary encoded instructions are added to the display list. When you extract the contents of a display list and store them in a buffer, you create metafile components—header data, an encoded stream of binary instructions, and trailer data. Each metafile component consists of binary encoded instructions. When you write the contents of the buffer to a file, you create a UIS metafile. A UIS metafile is a *generically encoded* binary stream; that is, all three components exist within a single file that is executable on any VAXstation system. The buffer and metafile contain values that describe the extracted objects. If reexecuted, these encoded instructions cause UIS to recreate the objects drawn in the virtual display. Note that monochrome systems cannot duplicate the color of extracted objects created on color systems.

You can write your own binary encoded instructions and metafiles. First, you must understand how to interpret the contents of the user-defined buffer containing the extracted data.

Opcodes

An opcode is the portion of the binary encoded instruction that specifies the instruction action. Table 15–1 lists the generic encoding symbols and the corresponding opcodes of binary encoded instructions.

Table 15–1 Generic Encoding Symbols and Opcodes

Generic Encoding Symbol	Opcode
Attribute	
GER$C_SET_WRITING_MODE	1
GER$C_SET_WRITING_INDEX	2
GER$C_SET_BACKGROUND_INDEX	3
GER$C_SET_CHAR_SPACING	4
GER$C_SET_CHAR_SLANT	5
GER$C_SET_TEXT_SLOPE	6
GER$C_SET_TEXT_PATH	7
GER$C_SET_TEXT_FORMATTING	11
GER$C_SET_CHAR_ROTATION	12
GER$C_SET_TEXT_MARGINS	13
GER$C_SET_LINE_WIDTH	14
GER$C_SET_LINE_STYLE	15
GER$C_SET_FONT	17
GER$C_SET_ARC_TYPE	26
GER$C_SET_FILL_PATTERN	37
GER$C_SET_CLIP	38
GER$C_SET_CHAR_ENCODING	39
GER$C_SET_CHAR_SIZE	42
Graphics and Text	
GER$C_TEXT	19
GER$C_SET_POSITION	21
GER$C_PLOT	23
GER$C_ELLIPSE	25
GER$C_IMAGE	29
GER$C_ALIGN_POSITION	33
GER$C_LINE	52
Application-specific Private Data	
GER$C_PRIVATE	30
Display List	
GER$C_BEGIN[1]	31

[1]This binary instruction has no arguments.

Table 15-1 (Cont.) Generic Encoding Symbols and Opcodes

Generic Encoding Symbol	Opcode
Display List	
GER$C_END[1]	32
GER$C_BEGIN_DISPLAY	34
GER$C_END_DISPLAY[1]	35
GER$C_VERSION	36
GER$C_IDENTIFICATION	43
GER$C_DATE	44
GER$C_NOP[1]	45
GER$C_PRIVATE_ECO	49
GER$C_DISPLAY_EXTENTS	51
Color	
GER$C_SET_COLORS	47
GER$C_SET_INTENSITIES	48
GER$C_CREATE_COLOR_MAP	50

[1]This binary instruction has no arguments.

Arguments

Figure 15-3 illustrates the format of an argument within a binary instruction that changes attribute settings.

Figure 15-3 Format of Attribute-Related Argument

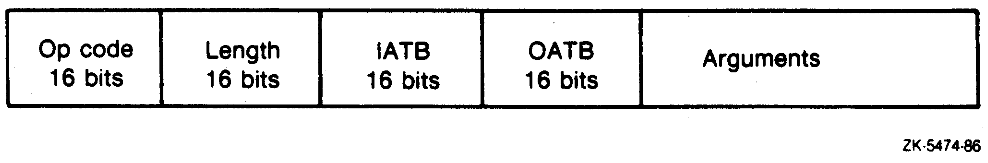


Figure 15–4 illustrates the format of an argument within a binary encoded instruction that produces graphics or text.

Figure 15–4 Format of Graphics- and Text-Related Argument

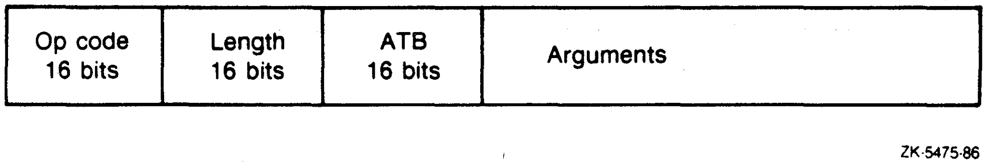


Table 15–2 lists the possible arguments that can appear in a binary encoded instruction.

Table 15–2 Arguments of Binary Encoded Instructions

Opcode	Argument[3]	Data Type	Description
Attributes[1]			
	iatb	word	Input attribute block for set operations
	oatb	word	Output attribute block for set operations
GER$C_SET_ARC_TYPE	arc_type	word	arc type
GER$C_SET_BACKGROUND_INDEX	background_index	word	Background index
GER$C_SET_CHAR_ENCODING	char_encoding_type	word	Character encoding type
GER$C_SET_CHAR_SIZE	char_size_flags	word	Scaling flags
	char_size_enable	bitfield mask	Font ideal size for x
	char_size_def_x	bitfield mask	Font ideal size for y
	char_size_def_y	bitfield mask	Widest char
	char_size_def_char	bitfield mask	
	char_size_example	word	Example character
	char_size_width	F_floating	Character width
	char_size_height	F_floating	Character height
GER$C_SET_CHAR_SLANT	char_slant_angle	F_floating	Character slant angle
GER$C_SET_CHAR_SPACING	char_space_dx char_space_dy	F_floating F_floating	Delta x spacing Delta y spacing

[1]All attribute-related encoding items start with input attribute block (IATB) and output attribute block (OATB) numbers and then contain attribute specific information.

[3]Arguments whose data type is word, longword, or character use the prefix GER$W_, GER$F_, or GER$G, respectively, EXCEPT GER$L_LINE_STYLE and GER$L_IMAGE_SIZE. For example, GERW_IATB, GERF_CHAR_SIZE_WIDTH, or GER$G_FONT_ID_STRING.

Table 15-2 (Cont.) Arguments of Binary Encoded Instructions

Opcode	Argument[3]	Data Type	Description
Attributes[1]			
GER$C_SET_CHAR_ROTATION	char_rotation_angle	F_floating	Character rotation angle
GER$C_SET_CLIP	clip_flags	word	Clipping rectangle
	clip_x1	F_floating	
	clip_y1	F_floating	
	clip_x2	F_floating	
	clip_y2	F_floating	
GER$C_SET_COLORS	color_count	word	Number of indices
	color_index	word	First index
	color_values	longword array	R, G, and B vectors
GER$C_SET_FILL_PATTERN	fill_flags	word	Flags
	fill_index	word	Index
GER$C_SET_FONT	font_id_length	word	Font name length
	font_id_string	character	Font name string
GER$C_SET_INTENSITIES	intensity_count	word	Number of indices
	intensity_index	word	First index
	intensity_values	longword array	I vector
GER$C_SET_LINE_STYLE	line_style	longword	32-bit bitvector
GER$C_SET_LINE_WIDTH	line_width_nc	F_floating	Normalized coordinates
	line_width_dc	F_floating	Pixel coordinates
	line_width_mode	word	Width mode
GER$C_SET_TEXT_FORMATTING	text_format_mode	word	Text formatting mode
GER$C_SET_TEXT_MARGINS	text_margin_x	F_floating	Starting position
	text_margin_y	F_floating	
	text_margin_distance	F_floating	Ending position
GER$C_SET_TEXT_PATH	text_path_major	word	Major path code

[1]All attribute-related encoding items start with input attribute block (IATB) and output attribute block (OATB) numbers and then contain attribute specific information.

[3]Arguments whose data type is word, longword, or character use the prefix GER$W_, GER$F_, or GER$G, respectively, EXCEPT GER$L_LINE_STYLE and GER$L_IMAGE_SIZE. For example, GERW_IATB, GERF_CHAR_SIZE_WIDTH, or GER$G_FONT_ID_STRING.

Table 15-2 (Cont.) Arguments of Binary Encoded Instructions

Opcode	Argument[3]	Data Type	Description
Attributes[1]			
	text_path_minor	word	Minor path code
GER$C_SET_TEXT_ SLOPE	text_slope_angle	F_floating	Angle of text slope
GER$C_SET_ WRITING_MODE	writing_mode	word	Writing mode
GER$C_SET_ WRITING_INDEX	writing_index	word	Writing index
Graphics and Text[2]			
	output_atb	word	ATB for graphics and text operations
GER$C_ELLIPSE	ellipse_x	F_floating	Center point
	ellipse_y	F_floating	
	ellipse_width	F_floating	Radius width and height
	ellipse_height	F_floating	
	ellipse_start_deg	F_floating	Starting and ending degrees
	ellipse_end_deg	F_floating	
GER$C_IMAGE	image_x1	F_floating	Lower-left corner of raster image
	image_y1	F_floating	
	image_x2	F_floating	Upper-right corner of raster image
	image_y2	F_floating	
	image_width	word	Image width in pixels
	image_height	word	Image height in pixels
	image_bpp	word	Bits per pixel
	image_size	longword	Number of bytes in image
	image_data	byte array	Place to store actual data
GER$C_PLOT	plot_count	word	Number of points
	plot_data	longword array	Points
GER$C_TEXT	text_encoding	word	8- or 16-bit encoding
	text_length	word	Text length in bytes
	text_data	character	Text string
GER$C_LINE	line_count	word	Number of points
	line_data	longword array	Points

[1]All attribute-related encoding items start with input attribute block (IATB) and output attribute block (OATB) numbers and then contain attribute specific information.

[2]All output-related encoding items start with an attribute block (ATB) number and are then followed by graphics and text output information.

[3]Arguments whose data type is word, longword, or character use the prefix GER$W_, GER$F_, or GER$G, respectively, EXCEPT GER$L_LINE_STYLE and GER$L_IMAGE_SIZE. For example, GERW_IATB, GERF_CHAR_SIZE_WIDTH, or GER$G_FONT_ID_STRING.

Table 15-2 (Cont.) Arguments of Binary Encoded Instructions

Opcode	Argument[3]	Data Type	Description
Color Map			
GER$C_CREATE_ COLOR_MAP	color_map_attributes	longword	Color map attributes
	color_map_ resident	bitfield mask	
	color_map_no_ bind	bitfield mask	
	color_map_ share	bitfield mask	
	color_map_ system	bitfield mask	
	color_map_name_size	word	
	color_map_size	word	
	color_map_name	character	Virtual color map name
Private Data			
GER$C_PRIVATE	private_facnum	word	Facility number
	private_length	word	Length of data
	private_data	byte array	Data
Metafile			
GER$C_VERSION	version_major	word	Encoding version number
	version_minor	word	
	version_eco	word	
GER$C_ IDENTIFICATION	identification_length	word	
	identification_string	character	
GER$C_DATE	date_length	word	File creation date
	date_string	character	
GER$C_PRIVATE_ ECO	private_eco_facnum	word	
	private_eco_major	word	
	private_eco_minor	word	
	private_eco_eco	word	
Miscellaneous			
GER$C_DISPLAY_ EXTENTS	extent_minx	F_floating	Extent rectangle
	extent_miny	F_floating	
	extent_maxx	F_floating	
	extent_maxy	F_floating	

[3]Arguments whose data type is word, longword, or character use the prefix GER$W_, GER$F_, or GER$G, respectively, EXCEPT GER$L_LINE_STYLE and GER$L_IMAGE_SIZE. For example, GERW_IATB, GERF_CHAR_SIZE_WIDTH, or GER$G_FONT_ID_STRING.

Table 15-2 (Cont.) Arguments of Binary Encoded Instructions

Opcode	Argument[3]	Data Type	Description
Miscellaneous			
GER$C_SET_POSITION	text_pos_x	F_floating	Text position
	text_pos_y	F_floating	
GER$C_ALIGN_POSITION	align_pos_atb	word	Attribute block
	align_pos_x	F_floating	Position
	align_pos_y	F_floating	
GER$C_BEGIN_DISPLAY	display_wc_minx	f_floating	Dimensions of virtual display
	display_wc_miny	f_floating	
	display_wc_maxx	f_floating	
	display_wc_maxy	f_floating	
	display_width	f_floating	
	display_height	f_floating	
GER$C_END_DISPLAY	No arguments		

[3]Arguments whose data type is word, longword, or character use the prefix GER$W_, GER$F_, or GER$G, respectively, EXCEPT GER$L_LINE_STYLE and GER$L_IMAGE_SIZE. For example, GERW_IATB, GERF_CHAR_SIZE_WIDTH, or GER$G_FONT_ID_STRING.

15.2.2 Creating UIS Metafiles

UIS metafiles are encoded binary instructions that are *generically encoded* when you use UIS$EXTRACT_OBJECT or UIS$EXTRACT_REGION to extract them from a display list. UIS metafiles consist of the following components:

- Header information
- Generically encoded binary instructions
- Trailer information

The header and trailer are special binary instructions that indicate the beginning and end of a UIS metafile. The generic encoding of UIS metafiles allows you to store the extracted contents of the display list in a buffer or file. Table 15-3 lists the parts of a UIS metafile.

Table 15-3 Structure of UIS Metafiles

Generic Encoding Symbol	Function
Header Information	
GER$C_VERSION	Level of generic encoding syntax. The version always appears first.
GER$C_IDENTIFICATION	User-specified optional identification string.
GER$C_DATE	Optional and user-specified.
GER$C_PRIVATE_ECO[1,2]	Optional and user-specified.
GER$C_CREATE_COLOR_ MAP	Used by UIS$EXECUTE_DISPLAY.
GER$C_SET_COLORS	Used by UIS$EXECUTE_DISPLAY.
GER$C_BEGIN_DISPLAY	Dimensions of the virtual display to be created by UIS$EXECUTE_DISPLAY.
Encoded Binary Instructions[2]	
GER$C_DISPLAY_ EXTENTS[3]	Define bounds of an extent rectangle used in UIS$EXTRACT_REGION.
Segment	Express the hierarchical structure within a display list and identify the attributes associated with a segment.
Attribute	Allow the modification of any attribute in any attribute block. A generic encoding opcode exists for each attribute.
Graphics and text	Contain the data necessary to draw graphic objects.
Application-specific	Associate data with a user-specified facility.
Trailer	
GER$C_END_DISPLAY	Ends the UIS metafile.

[1]Engineering Change Order

[2]See Table 15-1 for the generic symbols in each of these categories of binary encoded instructions.

[3]Generated only by UIS$EXTRACT_REGION.

15.2.3 Structure of a UIS Metafile

A UIS metafile consists of three components:

- Header information
- Binary instructions
- Trailer information

Figure 15–5 illustrates the structure of a UIS metafile containing a single extracted graphic object. Note that attribute modification instructions precede the object and private data instructions follow it. Also, if the extracted object were previously within a segment, segmentation instructions must surround it in the metafile.

Figure 15–5 Structure of UIS Metafile

Header Information	GER$C_VERSION	Length	Arguments		
	GER$C_IDENTIFICATION	Length	Arguments		
	GER$C_DATE	Length	Arguments		
	GER$C_BEGIN_DISPLAY	Length	Arguments		
Beginning Segmentation Instruction	GER$C_BEGIN	Length	No arguments		
Attribute Modification Instructions	GER$C_SET_FONT	Length	IATB	OATB	Arguments
	GER$C_SET_FILL_PATTERN	Length	IATB	OATB	Arguments
Extracted Graphic Object	GER$C_ELLIPSE	Length	ATB	Arguments	
Private Data	GER$C_PRIVATE	Length	Arguments		
	GER$C_PRIVATE	Length	Arguments		
Ending Segmentation Instruction	GER$C_END	Length	No Arguments		
Trailer Information	GER$C_END_DISPLAY	Length	No arguments		

ZK 5476 86

Private data is discussed later in this chapter.

15.2.4 Programming Options

With UIS metafiles, you can save display screen output for reexecution at a later time.

Creating UIS Metafiles

You can extract an object or the contents of a region within a virtual display and store the data in a buffer or file as a metafile. Use the following procedure:

1. Use UIS$EXTRACT_HEADER, UIS$EXTRACT_OBJECT or UIS$EXTRACT_REGION, and UIS$EXTRACT_TRAILER without the buffer length and buffer address parameters to determine the size of the buffer you need to store the header information, binary encoded stream, and trailer.
2. Call UIS$EXTRACT_HEADER, UIS$EXTRACT_OBJECT or UIS$EXTRACT_REGION, and UIS$EXTRACT_TRAILER with the previously omitted parameters to extract the header information, binary encoded instructions, and trailer and to store the data in three buffers.
3. Use VAX FORTRAN OPEN and WRITE statements to write the contents of the buffers to an external file.

Executing the Metafile

Use UIS$EXECUTE to write UIS metafiles extracted and stored in a buffer to the same virtual display.

UIS$EXECUTE_DISPLAY creates a new virtual display and executes the metafile in the new display space. However, you must call UIS$CREATE_WINDOW to view the graphic object in the virtual display.

15.2.5 Program Development I

Programming Objectives

To extract the contents of a region in the virtual display and create a UIS metafile.

Programming Tasks

1. Initialize variables.
2. Create a virtual display.
3. Draw graphic objects in the virtual display.
4. Create a display window and viewport.
5. Determine the size of each part of the metafile.
6. Allocate the space in buffers for each part of the metafile.
7. Extract the contents of the specified region in a buffer.

8 Write the contents of the buffer to an external file.

```
        PROGRAM EXTRACT
        IMPLICIT INTEGER(A-Z)
        INCLUDE 'SYS$LIBRARY:UISENTRY'
        INCLUDE 'SYS$LIBRARY:UISUSRDEF'
        DATA RETLEN1,RETLEN2,RETLEN3/3*0/
        VD_ID=UIS$CREATE_DISPLAY(1.0,1.0,30.0,30.0,20.0,20.0)

c Draw some objects
        CALL UIS$PLOT(VD_ID,0,7.0,10.0,16.0,10.0,7.0,15.0,
        2       7.0,10.0)                                 1
        CALL UIS$ELLIPSE(VD_ID,0,20.0,20.0,9.0,5.0)          2
        CALL UIS$TEXT(VD_ID,0,'Haste and wisdom are things far odd',
        2       11.0,15.0)                                   3
c Create a display window
        WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION')

        PAUSE

c Find out how much space to allocate for each part of the metafile
        CALL UIS$EXTRACT_HEADER(VD_ID,,,RETLEN1)  4
        CALL UIS$EXTRACT_REGION(VD_ID,,,,,,RETLEN2)         5
        CALL UIS$EXTRACT_TRAILER(VD_ID,,,RETLEN3)  6

c Virtual memory is allocated for the buffers
        STATUS=LIB$GET_VM(RETLEN1,ENCODED1)                 7
        IF (.NOT.STATUS) CALL LIB$STOP(%VAL(STATUS)) 8
        STATUS=LIB$GET_VM(RETLEN2,ENCODED2)                 9
        IF (.NOT.STATUS) CALL LIB$STOP(%VAL(STATUS)) 10
        STATUS=LIB$GET_VM(RETLEN3,ENCODED3)                 11
        IF (.NOT.STATUS) CALL LIB$STOP(%VAL(STATUS)) 12
        RETLEN=RETLEN1+RETLEN2+RETLEN3

        TYPE *,'HEADER DATA',RETLEN1,' BYTES'    13
        TYPE *,'BINARY INSTRUCTION',RETLEN2,' BYTES' 14
        TYPE *,'TRAILING DATA',RETLEN3,' BYTES'    15

        TYPE *,'NO. OF BYTES ALLOCATED = ',RETLEN     16

        PAUSE

C Extract the data and store it in a buffer
        CALL UIS$EXTRACT_HEADER(VD_ID,RETLEN1,%VAL(ENCODED1)) 17
        CALL UIS$EXTRACT_REGION(VD_ID,,,,,RETLEN2,%VAL(ENCODED2))        18
        CALL UIS$EXTRACT_TRAILER(VD_ID,RETLEN3,%VAL(ENCODED3))    19

c Write the contents of the buffer to an external file
        OPEN(UNIT=10,FILE='$DISK:[MY_DIR]METAFILE.DAT',STATUS='NEW') 20

c Call subroutine to write the contents of the buffer
        CALL BUFFERWRITE(%VAL(ENCODED1),RETLEN1,10)      21
        CALL BUFFERWRITE(%VAL(ENCODED2),RETLEN2,10)        22
        CALL BUFFERWRITE(%VAL(ENCODED3),RETLEN3,10)     23

c Close the external file
        CLOSE(UNIT=10,STATUS='SAVE')                        24

        END

        SUBROUTINE BUFFERWRITE(BUFFER,LENGTH,LUN)    25
        IMPLICIT INTEGER(A-Z)
        BYTE BUFFER(LENGTH)
        WRITE(LUN,500)BUFFER                        26
 500    FORMAT(T3,I7)

        RETURN
        END
```

Calls to UIS$PLOT, UIS$ELLIPSE, and UIS$TEXT 1 2 3 draw objects in the virtual display.

Next, determine how much space to allocate for the buffers that hold the header data, binary encoded stream, and trailing data. 4 5 6. The variables *retlen1*, *retlen2*, and *retlen3* receive the length of the header data, binary encoded stream, and trailing data.

Allocate virtual memory for the buffers and store the address of each buffer in the pointers *encoded1*, *encoded2*, and *encoded3* using LIB$GET_VM. 7 9 11. Perform a test for completion status of each Run-Time Library call 8 10 12.

Type the length of the header data, encoded stream, and trailing data as well as the total number of bytes allocated 16 in the emulation window 13 14 15.

Extract the contents of the display list with UIS$EXTRACT_HEADER, UIS$EXTRACT_REGION, and UIS$EXTRACT_TRAILER; store them at the location indicated by pointers *encoded1*, *encoded2*, and *encoded3* 17 18 19. Use the VAX FORTRAN built-in function %VAL to evaluate the pointers *encoded1*, *encoded2*, and *encoded3* in terms of the actual data they store—the addresses of the starting point of each buffer.

An external file is opened with the VAX FORTRAN OPEN statement for program output 20.

The pointer *encoded* is implicitly declared as a longword integer. Therefore, you cannot simply write the data to the file PRIVATE.DAT.

The subroutine BUFFERWRITE is called 21 22 23 three times to perform this task. Three arguments are passed in the call 25—buffer address, buffer size, and the VAX FORTRAN logical unit number of the output device. An array BUFFER is constructed from this data.

The subroutine BUFFERWRITE writes the contents of BUFFER to the UIS metafile PRIVATE.DAT 26. First the header data is stored in the metafile, then the binary encoded stream; finally, the trailing data is written to PRIVATE.DAT.

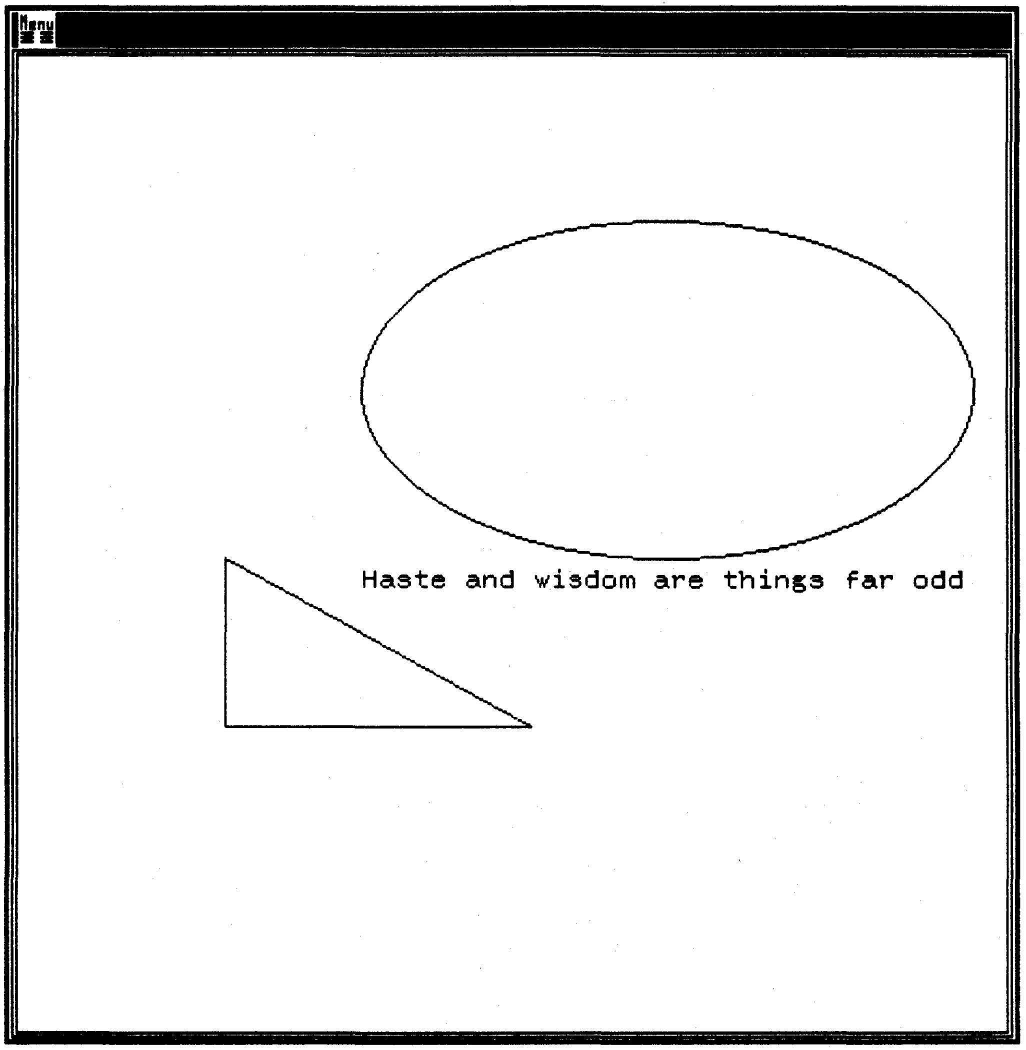
Before the program terminates, the VAX FORTRAN CLOSE statement closes the file 24.

15.2.5.1 Calling UIS$EXTRACT_HEADER, UIS$EXTRACT_REGION, and UIS$EXTRACT_TRAILER

A triangle, an ellipse, and text are drawn in a virtual display as shown in Figure 15-6.

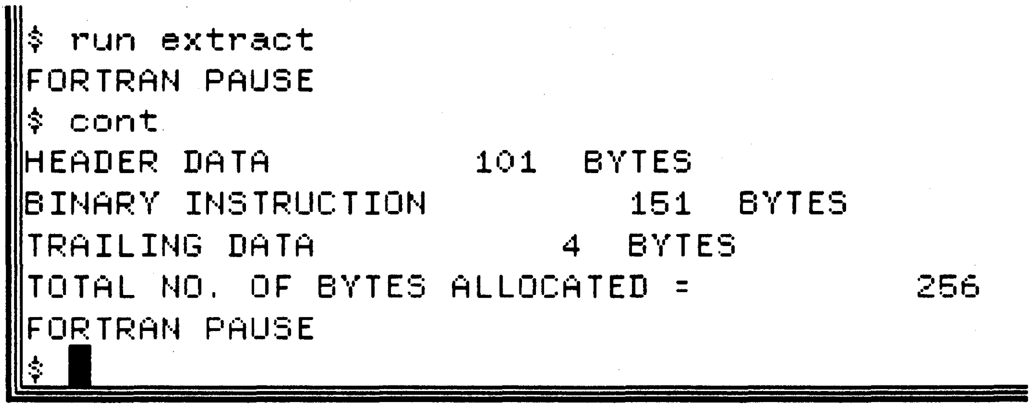
The terminal emulation window shown in Figure 15-7 shows buffer size information for metafile components.

Figure 15-6 Original Objects Drawn in the Virtual Display



ZK-5264-86

Figure 15-7 After Buffer Execution



```
$ run extract
FORTRAN PAUSE
$ cont
HEADER DATA          101  BYTES
BINARY INSTRUCTION          151  BYTES
TRAILING DATA            4  BYTES
TOTAL NO. OF BYTES ALLOCATED =          256
FORTRAN PAUSE
$
```

ZK 5265 86

15.3 Display Lists and Private Data

Display lists are created when graphics routines are executed. Application-specific or *private data* can be bound to graphic objects. The binary encoded instructions in the display list point to internal buffers that contain private data.

15.3.1 Using Private Data

Use private data to include application-specific information with the graphic objects displayed on the workstation screen. The nature of this information is entirely at your discretion. For example, an application that draws a vertical bar graph and plots relative humidity over a 24-hour period might create data on an hourly basis. The private data, here indicating temperature and wind speed, might be associated with each vertical bar. Private data is not displayed on the workstation screen and is not available unless extracted into a buffer or metafile and executed. You can attach private data to any graphic object in the virtual display.

15.3.2 Programming Options

To construct a program that reads data from an external file and uses it as private data.

Creating Private Data

Use UIS$PRIVATE to create private data.

Extracting Private Data

With the following procedure, you can use UIS$EXTRACT_PRIVATE to extract private data and store it in a buffer.

1 Use UIS$EXTRACT_HEADER, UIS$EXTRACT_PRIVATE, and UIS$EXTRACT_TRAILER without the buffer length and buffer address parameters to determine what size buffer you need to store the header information, binary encoded stream, and trailer.

2 Call UIS$EXTRACT_HEADER, UIS$EXTRACT_PRIVATE, and UIS$EXTRACT_TRAILER with the previously omitted parameters to extract the private data and store it in a buffer.

3 Use the VAX FORTRAN OPEN statement to write the contents of the buffer to an external file.

Deleting Private Data

Use UIS$DELETE_PRIVATE to delete private data associated with a graphic object.

15.3.3 Program Development II

Programming Objectives

1 To append private data to an object in the display list.

2 To extract the private data.

3 To create a UIS metafile containing the private data instruction.

Programming Tasks

1 Declare an array to receive the private data from an external file.

2 Type the contents of the array to verify it.

3 Create private data and append it to the last object in the display list.

4 Determine how large the buffers must be.

5 Allocate memory for the buffers.

6 Extract the private data.

7 Write the contents of the buffers to an external file.

NOTE: **Before you run this program, modify the file specifications in the OPEN statements and construct a data file similar to DATA.DAT. Your data file must be located in the same directory as the executable demonstration program file.**

```
          PROGRAM PRIVATE
          IMPLICIT INTEGER(A-Z)
          INCLUDE 'SYS$LIBRARY:UISENTRY'
          INCLUDE 'SYS$LIBRARY:UISUSRDEF'
          BYTE PRIV(1:23)                   ❶
c Construct a descriptor
          INTEGER*4 PRIV_DESC(2)                      ❷
          PRIV_DESC(1)=23
          PRIV_DESC(2)=%LOC(PRIV)                     ❸
```

```
c Open external file containing private data
        OPEN(UNIT=8,FILE='$DISK:[MY_DIR]DATA.DAT',STATUS='OLD') 4

c Read data into array
        READ(8,50)PRIV            5
 50     FORMAT(A7)

        CLOSE(UNIT=8,STATUS='SAVE')
        VD_ID=UIS$CREATE_DISPLAY(1.0,1.0,30.0,30.0,15.0,15.0)   6

c draw the hot air balloon
        CALL UIS$SET_FONT(VD_ID,0,2,'MY_FONT_5')
        INDEX=87
        CALL UIS$SET_FILL_PATTERN(VD_ID,2,2,INDEX)

        CALL UIS$CIRCLE(VD_ID,2,12.0,20.0,8.0)
        CALL UIS$LINE(VD_ID,2,10.0,12.0,10.0,8.0,14.0,12.0,14.0,8.0,
    2           10.0,10.0,14.0,10.0,10.0,8.0,14.0,8.0)

c draw house
        CALL UIS$PLOT(VD_ID,0,15.0,8.0,29.0,8.0,22.0,13.0,
    2           15.0,8.0)
        CALL UIS$LINE(VD_ID,0,15.0,8.0,15.0,0.0,29.0,8.0,29.0,0.0)

c draw door
        CALL UIS$PLOT(VD_ID,0,21.0,0.0,21.0,4.0,23.0,4.0,23.0,0.0)

C create windows
        CALL UIS$PLOT(VD_ID,0,17.0,2.0,17.0,6.0,19.0,6.0,19.0,2.0,
    2           17.0,2.0)
        CALL UIS$LINE(VD_ID,0,17.0,4.0,19.0,4.0,18.0,2.0,18.0,6.0)

        CALL UIS$PLOT(VD_ID,0,25.0,2.0,25.0,6.0,27.0,6.0,27.0,2.0,
    2           25.0,2.0)
        CALL UIS$LINE(VD_ID,0,25.0,4.0,27.0,4.0,26.0,2.0,26.0,6.0)

c create chimney
        CALL UIS$LINE(VD_ID,0,26.0,11.0,28.0,11.0,26.0,11.0,26.0,10.0,
    2           28.0,11.0,28.0,9.0)

c create smoke
        CALL UIS$ELLIPSE(VD_ID,0,27.0,13.0,2.5,1.0)
        CALL UIS$ELLIPSE(VD_ID,0,27.25,16.0,2.25,1.0)
        CALL UIS$ELLIPSE(VD_ID,0,27.5,19.0,2.0,1.0)
        CALL UIS$ELLIPSE(VD_ID,0,27.75,22.0,1.75,1.0)
        CALL UIS$ELLIPSE(VD_ID,0,28.0,25.0,1.5,1.0)
        CALL UIS$ELLIPSE(VD_ID,0,28.25,28.0,1.25,1.0)
        CURR_ID=UIS$GET_CURRENT_OBJECT(VD_ID)    7

c type out buffer containing private data
        TYPE *,PRIV                              8

c Create private data
        FACNUM = 1
        CALL UIS$PRIVATE(vd_id,FACNUM,PRIV_DESC)    9

        CALL UIS$SET_LINE_WIDTH(VD_ID,0,3,15.0)
        CALL UIS$PLOT(VD_ID,3,1.0,29.0,4.0,11.0)

        CALL UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION')

        PAUSE

c Determine size of buffer
        CALL UIS$EXTRACT_HEADER(VD_ID,,,RETLEN1)    10
        CALL UIS$EXTRACT_PRIVATE(CURR_ID,,,RETLEN2)    11
        CALL UIS$EXTRACT_TRAILER(VD_ID,,,RETLEN3)    12

        RETLEN=RETLEN1+RETLEN2+RETLEN3

        TYPE *,'BUFFER SIZE FOR HEADER INFO',RETLEN1,'BYTES'  13
        TYPE *,'BUFFER SIZE REQUIRED',RETLEN2,' BYTES'   14
        TYPE *,'BUFFER SIZE FOR TRAILING INFO',RETLEN3,'BYTES 15

C Allocate the virtual memory for the buffer
```

```
        STATUS=LIB$GET_VM(RETLEN1,EXT_PRIV1)   16
        IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))   17
        STATUS=LIB$GET_VM(RETLEN2,EXT_PRIV2)   18
        IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))   19
        STATUS=LIB$GET_VM(RETLEN3,EXT_PRIV3)   20
        IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))   21

c Extract and store private data in buffer
        CALL UIS$EXTRACT_HEADER(VD_ID,RETLEN1,%VAL(EXT_PRIV1))        22
        CALL UIS$EXTRACT_PRIVATE(CURR_ID,RETLEN2,%VAL(EXT_PRIV2))  23
        CALL UIS$EXTRACT_TRAILER(VD_ID,RETLEN3,%VAL(EXT_PRIV3))  24

        CALL BUFFERTYPE(%VAL(EXT_PRIV2),RETLEN2) 25

C Open an external file
        OPEN(UNIT=11,FILE='$DISK:[MY_DIR]PRIVATE.DAT',STATUS='NEW',  26
    2           FORM='FORMATTED')

c Write the contents of the buffer
        CALL BUFFERWRITE(%VAL(EXT_PRIV1),RETLEN1,11)           27
        CALL BUFFERWRITE(%VAL(EXT_PRIV2),RETLEN2,11)           28
        CALL BUFFERWRITE(%VAL(EXT_PRIV3),RETLEN3,11)           29

C Close the file
        CLOSE(UNIT=11,STATUS='SAVE')

        PAUSE

        END
        SUBROUTINE BUFFERWRITE(BUFFER,LENGTH,LUN)   30
        IMPLICIT INTEGER(A-Z)
        BYTE BUFFER(LENGTH)

        WRITE(LUN,500)BUFFER                      31
 500    FORMAT(T3,I7)

        RETURN
        END

        SUBROUTINE BUFFERTYPE(BUFFER,length)  32
        IMPLICIT INTEGER(A-Z)
        BYTE BUFFER(length)

        TYPE *,buffer                 33

        RETURN
        END
```

A data file DATA.DAT of private data is constructed. It consists of a sentence. Because each character requires a byte of storage, the total number of characters in the data file is specified as the upper bound of array PRIV 1 as well as the buffer length in the descriptor you must construct for UIS$PRIVATE 2.

An external file DATA.DAT is opened 4 and read into the array PRIV 5.

A circle, a triangle, and text are drawn in the virtual display 6.

UIS$GET_CURRENT_OBJECT retrieves the identifier of the last object drawn in the virtual display 7.

The array PRIV is typed out to verify its contents 8.

UIS$PRIVATE associates the sentence contained in the array PRIV with the objects drawn in the virtual display 9. Note that the location of the array PRIV is passed by descriptor 3.

To extract the data and store it in a buffer as a UIS metafile, you must first determine how much space the header data, binary encoded private data, and trailing data will occupy. To do this, call UIS$EXTRACT_HEADER, UIS$EXTRACT_PRIVATE, and UIS$EXTRACT_TRAILER without the **buflen** and **bufaddr** arguments 10 11 12.

Type out the variables *retlen1*, *retlen2*, and *retlen3* to reveal the size of each part of the display list [13] [14] [15].

Call LIB$GET_VM to allocate virtual memory for three buffers using the value of *retlen1*, *retlen2*, and *retlen3* and to store the location of each buffer in the pointers *ext_priv1*, *ext_priv2*, and *ext_priv3* [16] [18] [20]. A test for completion status is performed for each Run-Time Library call [17] [19] [21].

If you do not use LIB$GET_VM, you have to declare explicitly an array with an actual length in the beginning of the program. However, at that point in the program, you have no idea how large such an array should be.

Call UIS$EXTRACT_HEADER, UIS$EXTRACT_PRIVATE, and UIS$EXTRACT_TRAILER with the omitted parameters to extract the header data, binary encoded private data, and the trailing data and to store them in separate buffers [22] [23] [24]. Because *ext_priv1*, *ext_priv2*, and *ext_priv3* are pointers, use the VAX FORTRAN built-in function %VAL to obtain the actual data they store.

To look at the contents of the user buffer before you write the contents to an external file, you cannot simply type the data in the user buffer because the pointer *ext_priv* is implicitly declared a longword integer and functions as a pointer.

Call subroutine BUFFERTYPE to reference the pointer *ext_priv2* and the size of the buffer [25]. Two arguments are passed in the call—the pointer name and the size of the buffer. The subroutine BUFFERTYPE reads the data from the location to which *ext_priv2* points [32] and writes the data in the terminal emulation window [33].

The file PRIVATE.DAT is opened [26].

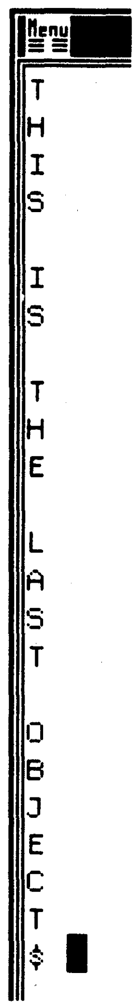
The subroutine BUFFERWRITE [30] is called three times to write the header, private, and trailer data to the external file [27] [28] [29]. Three arguments are passed in the call—buffer address, buffer size, and the VAX FORTRAN logical unit number of the output device. An array BUFFER is declared from this data and an association with an external file is established.

The subroutine BUFFERWRITE writes the contents of BUFFER to the file PRIVATE.DAT [31]. The file is closed and saved.

15.3.3.1 Calling UIS$PRIVATE and UIS$EXTRACT_PRIVATE

Figure 15–8 shows the sample containing character string private data in the external file DATA.DAT

Figure 15–8 Private Data

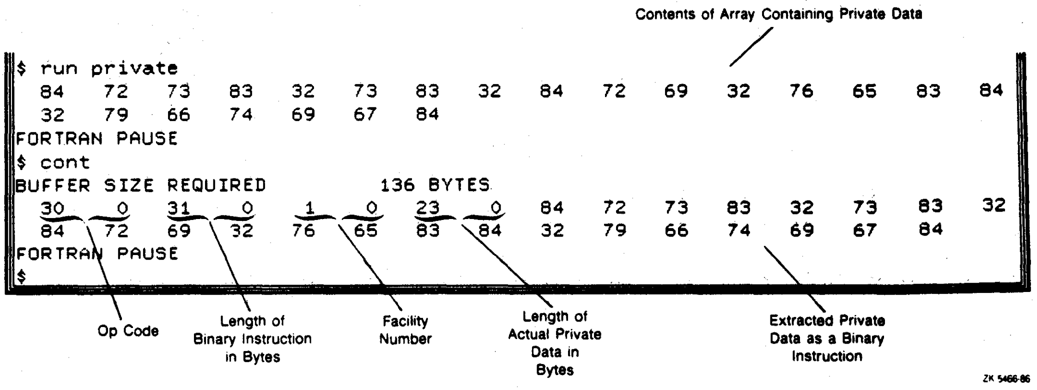


ZK-5454-86

Figure 15–9 shows the contents of the array PRIV read from the external file DATA.DAT. Note that each number is an ASCII code. The required buffer size is also shown. In addition, the extracted generically encoded binary private data instruction is shown as metafile opcodes and ASCII codes.

The private data is appended to the last ellipse drawn—the smallest cloud of smoke rising from the chimney shown in Figure 15–10.

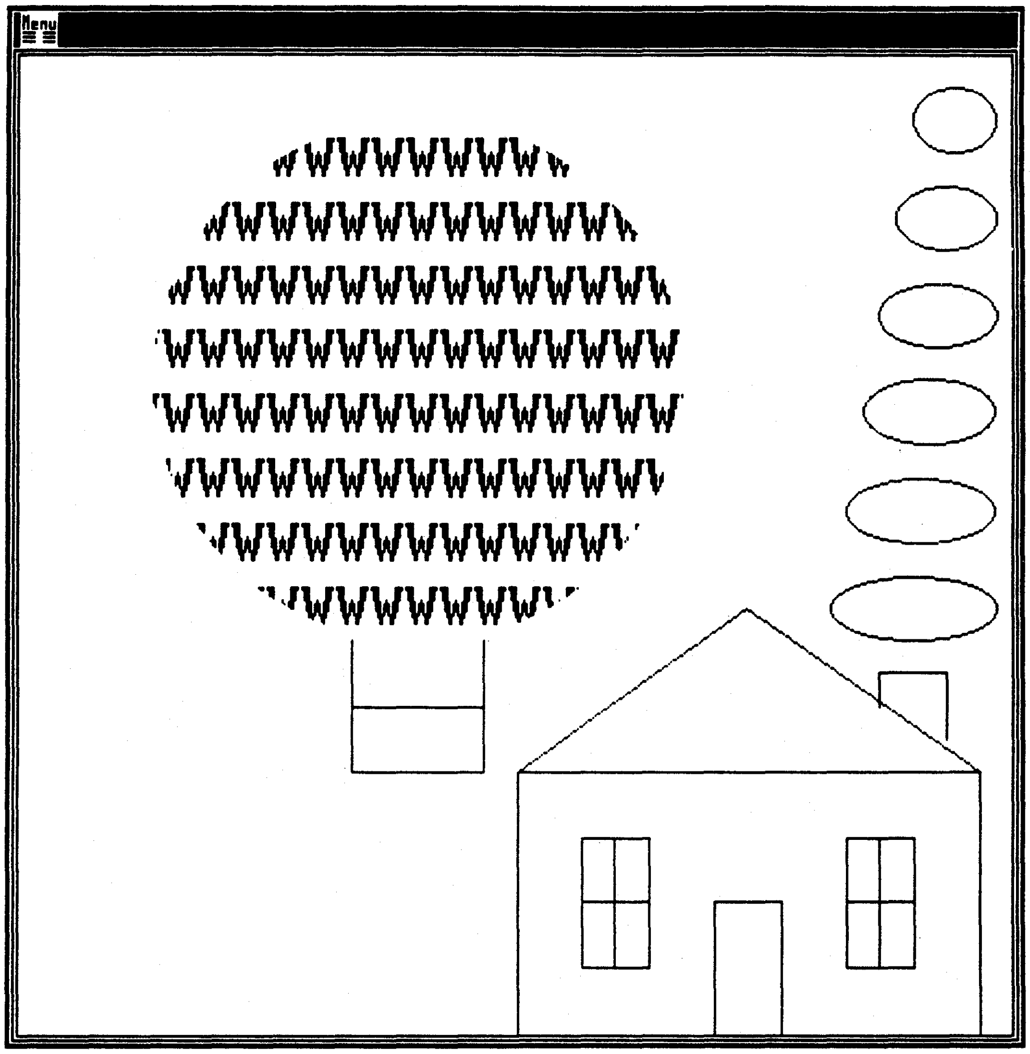
Figure 15–9 Verifying the Contents of the Temporary Array and User Buffer



```
$ run private
  84   72   73   83   32   73   83   32   84   72   69   32   76   65   83   84
  32   79   66   74   69   67   84
FORTRAN PAUSE
$ cont
BUFFER SIZE REQUIRED          136 BYTES
  30    0   31    0    1    0   23    0   84   72   73   83   32   73   83   32
  84   72   69   32   76   65   83   84   32   79   66   74   69   67   84
FORTRAN PAUSE
$
```

Contents of Array Containing Private Data
Op Code
Length of Binary Instruction in Bytes
Facility Number
Length of Actual Private Data in Bytes
Extracted Private Data as a Binary Instruction
ZK 5466-86

Figure 15–10 Hot Air Balloon



16 Programming in Color

16.1 Overview

To change the appearance of graphic objects of text, you can modify the settings in attribute block 0. Depending on the VAXstation color system you have, you can also draw graphic objects in over 16 million colors. This chapter discusses the following topics:

- Using color and intensity routines
- Setting entries in virtual color maps
- Creating shareable color maps
- Using color map segments
- Using color and intensity inquiry routines

This chapter is informative for VAXstation programmers with either an intensity or color environment.

16.2 Color and Intensity Routines

Your application uses color and intensity routines to draw graphic objects in color or shades of gray. These routines create and load the virtual color map and color map segment structures that hold the application color values. Color and intensity routines perform the following tasks:

- Create and delete virtual color maps
- Load virtual color map entries with color values
- Create and delete color map segments
- Load entries in color map segments

Color map segments are described later in this chapter.

16.2.1 Programming Options

When your application includes a range of color or intensities, use one or more of the UIS routines listed in Table 16–1.

Table 16–1 Color and Intensity Routines

Routine	Function
Virtual Color Maps	
UIS$CREATE_COLOR_MAP	Creates a virtual color map
UIS$DELETE_COLOR_MAP	Deletes a virtual color map

Table 16-1 (Cont.) Color and Intensity Routines

Routine	Function
Loading Virtual Color Map Entries	
UIS$SET_COLOR	Sets a single RGB color value in a virtual color map
UIS$SET_COLORS	Sets multiple RGB color values in a virtual color map
UIS$SET_INTENSITY	Sets a single intensity value in a virtual color map
UIS$SET_INTENSITIES	Sets multiple RGB color values in a virtual color map
Color Map Segments	
UIS$CREATE_COLOR_MAP_SEG	Creates a color map segment
UIS$DELETE_COLOR_MAP_SEG	Deletes a color map segment

16.2.2 Step 1—Creating a Virtual Color Map

In a color or an intensity environment, you must use UIS$CREATE_COLOR_MAP to create a virtual color map, which is a storage location similar to an artist's palette. Within the color map, you store color values in locations known as entries. The virtual color map varies according to the needs of your application. Specify the virtual color map *attributes* as you see fit.

16.2.3 Step 2—Setting Virtual Color Map Attributes

Some virtual color map attributes are required and some are optional. You **must** specify the size of the virtual color map; that is, how many color map values it will hold. You can specify name, access, and residency for the virtual color map if you wish.

Virtual Color Map Size

As with any storage location, size is a consideration. For every color your application uses, you need an entry in the virtual color map. You can specify a maximum size of 32,768 entries.

Access to Virtual Color Maps

To determine who or what process has access to a virtual color map, designate it *private* (no other processes have access to it) or *shareable* (some or all processes can share it).

Virtual Color Map Residency

You can also explicitly specify *residency*, which allows you to dedicate color resources to your application. Use this feature carefully, because it precludes sharing hardware color resources among applications.

16.2.4 Step 3—Setting Entries in the Virtual Color Map

Depending on color environment, your application must now load color values into the color map entries with UISSET_COLOR, UISSET_COLORS, UIS$SET_INTENSITIES, or UIS$SET_INTENSITY.

Color and intensity values are expressed as floating-point numbers between *0.0* and *1.0*. The color subsystem uses the red green blue (RGB) color model. Colors that result from color values with percentages of red, green, and blue are not always readily apparent from the value chosen. Therefore, as you write your application, you should use human-interface color setup menus to determine the appropriate RGB color component values.

Setting Single Entries

For an application with only a few colors or intensities, you might need only a small virtual color map. In this case, use UIS$SET_COLOR or UIS$SET_INTENSITY to load color map entries each time.

Setting Multiple Entries

If your virtual color map is large, you can arrange your color map values in an array with a single call to UIS$SET_COLORS or UIS$SET_INTENSITIES.

16.2.5 Program Development I

Programming Objective

To create and load a color map with single entries.

Programming Tasks

1 Establish a size for the virtual color map.

2 Create the virtual color map.

3 Create a virtual display.

4 Create a display window and viewport.

5 Use UIS$SET_COLOR to load a single color map entry with one color value.

```
PROGRAM SINGLE_ENTRY
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
REAL J,K
DATA J/17.0/              ❶
DATA K/16/                ❷
DATA VCM_SIZE/8/
VCM_ID=UIS$CREATE_COLOR_MAP(VCM_SIZE)    ❸
VD_ID=UIS$CREATE_DISPLAY(1.0,1.0,40.0,40.0,15.0,15.0,VCM_ID)  ❹
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','WINDOW #1')
```

```
CALL UIS$SET_COLOR(VD_ID,0,0.40,0.30,0.0)  ❺
CALL UIS$SET_COLOR(VD_ID,1,0.5,0.5,0.5)      ❻
CALL UIS$SET_COLOR(VD_ID,2,0.5,0.25,0.5)   ❼
CALL UIS$SET_COLOR(VD_ID,3,0.0,0.7,0.3)    ❽
CALL UIS$SET_COLOR(VD_ID,4,0.25,0.25,0.9)    ❾
CALL UIS$SET_COLOR(VD_ID,5,0.90,0.5,0.0)       ❿
CALL UIS$SET_COLOR(VD_ID,6,0.80,0.30,0.0)    ⓫
CALL UIS$SET_COLOR(VD_ID,7,0.35,0.65,0.95)  ⓬
CALL UIS$SET_WRITING_INDEX(VD_ID,0,9,2) ⓭
CALL UIS$SET_WRITING_INDEX(VD_ID,0,10,3)  ⓮
CALL UIS$SET_WRITING_INDEX(VD_ID,0,11,4)  ⓯
CALL UIS$SET_WRITING_INDEX(VD_ID,0,12,5)  ⓰
CALL UIS$SET_WRITING_INDEX(VD_ID,0,13,6)    ⓱
DO I=9,13,1
CALL UIS$CIRCLE(VD_ID,I,J,20.0,10.0)    ⓲
J=J+2.0
ENDDO

PAUSE

DO I=9,13
CALL UIS$CIRCLE(VD_ID,I,21.0,K,10.0)    ⓳
K=K+2.0
ENDDO
PAUSE

END
```

The counters *j* and *k* are declared and initialized ❶ ❷.

An eight-entry virtual color map is created with no attributes specified ❸.

The virtual color map is associated with the virtual display in UIS$CREATE_DISPLAY ❹ during creation of the virtual display.

Each color value is loaded into a virtual color map with successive calls to UIS$SET_COLOR ❺ ❻ ❼ ❽ ❾ ❿ ⓫ ⓬.

The default writing color attribute setting in attribute block 0 is modified such that five new default writing colors are associated with a virtual color map entry ⓭ ⓮ ⓯ ⓰ ⓱.

The **atb** argument in the call to UIS$CIRCLE within the DO loop references the modified attribute block. As a result, five circles are drawn horizontally ⓲, each with a different default writing color.

Five circles are drawn vertically ⓳, the same colors as the horizontally drawn circles.

16.2.6 Program Development II

Programming Objectives

To create and load a color map with more than one entry at a time.

Programming Task

1 Load the arrays with color component values.

2 Establish color map size.

3 Use UIS$SET_COLORS to load eight color map entries in a single call.

```
PROGRAM MULTIPLE_ENTRY
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
REAL J,K
REAL R_VECTOR(8),G_VECTOR(8),B_VECTOR(8)       ❶
DATA J/17.0/                                    ❷
DATA K/16/                                            ❸
DATA R_VECTOR/0.40,0.50,0.50,0.0,0.25,0.90,0.80,0.35/   ❹
DATA G_VECTOR/0.30,0.50,0.25,0.70,0.25,0.50,0.30,0.65/   ❺
DATA B_VECTOR/0.0,0.50,0.50,0.30,0.90,0.0,0.0,0.95/    ❻
DATA VCM_SIZE/8/
VCM_ID=UIS$CREATE_COLOR_MAP(VCM_SIZE)          ❼
VD_ID=UIS$CREATE_DISPLAY(1.0,1.0,40.0,40.0,15.0,15.0,VCM_ID)  ❽
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','COLOR')

CALL UIS$SET_COLORS(VD_ID,0,8,R_VECTOR,G_VECTOR,B_VECTOR)  ❾

CALL UIS$SET_WRITING_INDEX(VD_ID,0,9,2)
CALL UIS$SET_WRITING_INDEX(VD_ID,0,10,3)
CALL UIS$SET_WRITING_INDEX(VD_ID,0,11,4)
CALL UIS$SET_WRITING_INDEX(VD_ID,0,12,5)
CALL UIS$SET_WRITING_INDEX(VD_ID,0,13,6)
DO I=9,13,1
CALL UIS$CIRCLE(VD_ID,I,J,20.0,10.0)
J=J+2.0
ENDDO

PAUSE
DO I=9,13
CALL UIS$CIRCLE(VD_ID,I,21.0,K,10.0)
K=K+2.0
ENDDO

PAUSE

END
```

Three arrays are declared ❶ to hold eight R, G, and B color component values each.

The counters *j* and *k* are declared and initialized ❷ ❸.

The arrays R_VECTOR, G_VECTOR, and B_VECTOR are loaded with color component values ❹ ❺ ❻.

An eight-entry virtual color map is created ❼ and associated with a newly created virtual display ❽.

The R, G, and B color component values stored in the arrays are loaded in the virtual color map using a single call to UIS$SET_COLORS ❾.

The remaining portions of the program are identical to the previous program SINGLE_ENTRY.

16.2.6.1 Program Development III

Programming Objective

To create a shareable color map.

Programming Tasks

1. Load arrays containing color component values.
2. Create the color map attributes list, specifying the shareable attribute.
3. Create a virtual display, specifying a name for the color map.
4. Create a display window and display viewport.
5. Load color values into the color map.
6. Program 2 must perform steps 2 through 4 and reference the name of the color map specified in Program 1.

```
PROGRAM SHAREABLE_MAP
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
REAL J,K                                 1
REAL R_VECTOR(8),G_VECTOR(8),B_VECTOR(8)
INTEGER*4 VCM_ATTRIBUTES(3)                 2
DATA J/17.0/                                     3
DATA K/16/                                          4
DATA R_VECTOR/0.40,0.50,0.50,0.0,0.25,0.90,0.80/
DATA G_VECTOR/0.30,0.50,0.25,0.70,0.25,0.50,0.30/
DATA B_VECTOR/0.0,0.50,0.50,0.30,0.90,0.0,0.0/
DATA VCM_SIZE/8/
VCM_ATTRIBUTES(1)=VCMAL$C_ATTRIBUTES              5
VCM_ATTRIBUTES(2)=VCMAL$M_SHARE                     6
VCM_ATTRIBUTES(3)=VCMAL$C_END_OF_LIST                  7

VCM_ID=UIS$CREATE_COLOR_MAP(VCM_SIZE,'LIVING_COLOR',VCM_ATTRIBUTES)
VD_ID=UIS$CREATE_DISPLAY(1.0,1.0,40.0,40.0,15.0,15.0,VCM_ID)    9
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','PROCESS #1')

CALL UIS$SET_COLORS(VD_ID,0,8,R_VECTOR,G_VECTOR,B_VECTOR)

CALL UIS$SET_WRITING_INDEX(VD_ID,0,9,2)     10
CALL UIS$SET_WRITING_INDEX(VD_ID,0,10,3)
CALL UIS$SET_WRITING_INDEX(VD_ID,0,11,4)
CALL UIS$SET_WRITING_INDEX(VD_ID,0,12,5)
CALL UIS$SET_WRITING_INDEX(VD_ID,0,13,6)
DO I=9,13,1
CALL UIS$CIRCLE(VD_ID,I,J,20.0,10.0)
J=J+2.0
ENDDO

VD_ID2=UIS$CREATE_DISPLAY(1.0,1.0,40.0,40.0,15.0,15.0,VCM_ID)   11
WD_ID2=UIS$CREATE_WINDOW(VD_ID2,'SYS$WORKSTATION','WINDOW #2')
CALL UIS$SET_WRITING_INDEX(VD_ID2,0,9,2)     12
CALL UIS$SET_WRITING_INDEX(VD_ID2,0,10,3)
CALL UIS$SET_WRITING_INDEX(VD_ID2,0,11,4)
CALL UIS$SET_WRITING_INDEX(VD_ID2,0,12,5)
CALL UIS$SET_WRITING_INDEX(VD_ID2,0,13,6)

DO I=9,13,1
CALL UIS$CIRCLE(VD_ID2,I,21.0,K,10.0)
K=K+2.0
ENDDO

PAUSE

END
```

The counters *j* and *k* are declared and initialized 1 3 4.

An integer array VCM_ATTRIBUTES is declared to have three elements ❷.

The array elements are assigned attribute values defined by UIS constants ❺ ❻ ❼. The structure contains an attribute code followed by a longword value for that attribute. The final element contains a longword 0 to terminate the list.

An eight-entry virtual color map is created with UIS$CREATE_COLOR_MAP and the array VCM_ATTRIBUTES is used as an argument ❽.

The newly created virtual display references the virtual color map ❾. Objects drawn in the virtual display can use this virtual color map.

Different default writing colors are defined ❿ as in previous programs, to highlight and differentiate the objects drawn.

A second virtual display is created ⓫. The second call to UIS$CREATE_DISPLAY references the same virtual color map identifier as the first. Both virtual displays share the use of color value assignments in this virtual color map.

You must call UIS$SET_WRITING_INDEX ⓬ again to change the default setting of the writing color so that objects will be the same colors as those drawn in the first virtual display.

Here is a portion of a second program that uses the virtual color map LIVING_COLOR in the program SHAREABLE_MAP.

```
        .
        .
        .
             PROGRAM SECOND_PROGRAM
        .
        .
        .
             INTEGER*4 VCM_ATTRIBUTES(3)  ❶
             DATA VCM_SIZE/8/      ❷

             VCM_ATTRIBUTES(1)=VCMAL$C_ATTRIBUTES
             VCM_ATTRIBUTES(2)=VCMAL$M_SHARE
             VCM_ATTRIBUTES(3)=VCMAL$C_END_OF_LIST
             VCM_ID=UIS$CREATE_COLOR_MAP(VCM_SIZE,'LIVING_COLOR',VCM_ATTRIBUTES)
             VD_ID2=UIS$CREATE_DISPLAY(1.0,1.0,35.0,35.0,10.0,10.0,VCM_ID)

             WD_ID2=UIS$CREATE_WINDOW(VD_ID2,'SYS$WORKSTATION','PROCESS #2)
        .
        .
        .
```

An array of virtual color map attributes specifies the same attributes as those indicated in the preceding program SHAREABLE_MAP ❶. The application SECOND_PROGRAM must declare the virtual color map size ❷ as this argument is required in UIS$CREATE_COLOR_MAP.

The shareable color map is referenced by name in a call to UIS$CREATE_COLOR_MAP ❸.

16.3 Color Map Segments

Use color map segments to control binding the virtual color map to the hardware color map.

16.3.1 Programming Options

You can use UIS$CREATE_COLOR_MAP_SEG and UIS$DELETE_COLOR_MAP_SEG to create and delete color map segments.

16.3.2 Program Development

The program COLOR_SEG is a portion of a longer program and shows how to bind your virtual color map to the hardware color map.

```
      PROGRAM COLOR_SEG
      IMPLICIT INTEGER(A-Z)
      INCLUDE 'SYS$LIBRARY:UISENTRY'
      INCLUDE 'SYS$LIBRARY:UISUSRDEF'
      INTEGER*4 VCM_ATTRIBUTES(3)  [1]
      DATA VCM_SIZE,PLACEMENT_DATA/8,16/          [2]
  .
  .
  .
      VCM_ATTRIBUTES(1)=VCMAL$C_ATTRIBUTES     [3]
      VCM_ATTRIBUTES(2)=VCMAL$M_NOBIND             [4]
      VCM_ATTRIBUTES(3)=VCMAL$C_END_OF_LIST                [5]
      VCM_ID=UIS$CREATE_COLOR_MAP(VCM_SIZE,,VCM_ATTRIBUTES)       [6]
      CMS_ID=UIS$CREATE_COLOR_MAP_SEG(VCM_ID,'SYS$WORKSTATION',
      2         UIS$C_COLOR_EXACT,PLACEMENT_DATA)   [7]

      VD_ID=UIS$CREATE_DISPLAY(1.0,1.0,30.0,30.0,10.0,10.0,VCM_ID)
      WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION')
  .
  .
  .
```

Two declarations are established—an array VCM_ATTRIBUTES is declared [1] and the virtual color map size is initialized to *8* [2].

Because the color map segment is created with exact placement, the **placement_data** argument of UIS$CREATE_COLOR_MAP_SEG must be initialized to the starting index in the hardware color map where binding is to occur.

The elements of array VCM_ATTRIBUTES are assigned an attribute code [3], an attribute value VCMAL$M_NOBIND [4], and a terminating value [5].

UIS$CREATE_COLOR_MAP is called before any other UIS routine.

16.3.3 Calling UIS$CREATE_COLOR_MAP_SEG

No special graphics effects are displayed on the VAXstation screen.

16.4 Color and Intensity Inquiry Routines

As mentioned in Chapter 12, inquiry routines provide an application with status information. Several UIS color and intensity routines return information to the application about color setup, virtual color map, and hardware color map. This information can be direct input to your application.

16.4.1 Programming Options

Your application can use one or more inquiry routines. Table 16-2 lists color and intensity inquiry routines.

Table 16-2 Color and Intensity Inquiry Routines

Routine	Information Returned
Virtual Color Map	
UIS$GET_COLOR	Single RGB value from a virtual color map
UIS$GET_COLORS	Multiple RGB values from a virtual color map
UIS$GET_INTENSITIES	Multiple intensity values from a virtual color map
UIS$GET_INTENSITY	Single intensity value from a virtual color map
Hardware Color Map	
UIS$GET_HW_COLOR_ INFO	Device type; number of indexes; number of colors; bits of precision for R, G, and B values; reserved entries; and regeneration characteristics.
Color Value Conversion	
UIS$HLS_TO_RGB	Converts HLS color values to RGB color values
UIS$HSV_TO_RGB	Converts HSV color values to RGB color values
UIS$RGB_TO_HLS	Converts RGB color values to HLS color values
UIS$RGB_TO_HSV	Converts RGB color values to HSV color values
Workstation Standard Colors	
UIS$GET_WS_COLOR	Workstation standard RGB color value
UIS$GET_WS_ INTENSITY	Workstation standard intensity value
Color Setup	
UIS$GET_ BACKGROUND_INDEX	Window background index
UIS$GET_WRITING_ INDEX	Window foreground index
UIS$GET_WRITING_ MODE	Writing mode

16.4.2 Program Development I

Programming Objective

To retrieve hardware color map information.

Programming Tasks

1 Create a virtual color map.

2 Create a virtual display.

3 Create a display window and viewport.

4 Obtain the number of color map indices, possible colors, maps, bits of precision for each color component, and reserved entries.

```
        PROGRAM GET_INFO
        IMPLICIT INTEGER(A-Z)
        INCLUDE 'SYS$LIBRARY:UISENTRY'
        INCLUDE 'SYS$LIBRARY:UISUSRDEF'
        REAL J,K
        REAL R_VECTOR(8),G_VECTOR(8),B_VECTOR(8)
        REAL RETR_VECTOR(8),RETG_VECTOR(8),RETB_VECTOR(8)
        INTEGER*4 VCM_ATTRIBUTES(3)
        DATA VCM_SIZE/8/
 DATA J/17.0/
        DATA K/16/
        DATA R_VECTOR/0.40,0.50,0.50,0.0,0.25,0.90,0.80,0.35/
        DATA G_VECTOR/0.30,0.50,0.25,0.70,0.25,0.50,0.30,0.65/
        DATA B_VECTOR/0.0,0.50,0.50,0.30,0.90,0.0,0.0,0.95/
        VCM_ATTRIBUTES(1)=VCMAL$C_ATTRIBUTES
        VCM_ATTRIBUTES(2)=VCMAL$M_SHARE
        VCM_ATTRIBUTES(3)=VCMAL$C_END_OF_LIST

        VCM_ID=UIS$CREATE_COLOR_MAP(VCM_SIZE,VCM_ATTRIBUTES)
        VD_ID=UIS$CREATE_DISPLAY(1.0,1.0,40.0,40.0,15.0,15.0,VCM_ID)
        WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','COLOR')

        CALL UIS$SET_COLORS(VD_ID,0,8,R_VECTOR,G_VECTOR,B_VECTOR)
        CALL UIS$SET_WRITING_INDEX(VD_ID,0,9,2)
        CALL UIS$SET_WRITING_INDEX(VD_ID,0,10,3)
        CALL UIS$SET_WRITING_INDEX(VD_ID,0,11,4)
        CALL UIS$SET_WRITING_INDEX(VD_ID,0,12,5)
        CALL UIS$SET_WRITING_INDEX(VD_ID,0,13,6)

        CALL UIS$GET_COLORS(VD_ID,0,8,RETR_VECTOR,RETG_VECTOR,RETB_VECTOR)

        TYPE 50
 50     format(T8,'RED',T18,'GREEN',T30,'BLUE')
        TYPE 100,RETR_VECTOR,RETG_VECTOR,RETB_VECTOR
 100    FORMAT(F11.3,F11.3,F11.3)

        CALL UIS$GET_HW_COLOR_INFO(,,
        2       INDICES,COLORS,MAPS,RBITS,GBITS,BBITS,,RES_INDICES) ❷

        TYPE 150,INDICES,COLORS
 150    FORMAT(T2,'NO. OF INDICES=',I3,T22,'NO. OF COLORS=',I8)
        TYPE 200,MAPS
 200    FORMAT(T2,'NO.OF MAPS=',i3)

        TYPE 225,RBITS,GBITS,BBITS
 225    FORMAT(T2,'NO. OF BITS OF PRECISION',T28,'RED',I3,T37,'GREEN',I3,
        2       T48,'BLUE',I3)

        TYPE 250,RES_INDICES
 250    FORMAT(T2,'NO. OF RESERVED ENTRIES',I3)
        TYPE*,'VCM Indexes Used In Virtual Display 1'

        DO I=9,13,1
        CALL UIS$CIRCLE(VD_ID,I,J,20.0,10.0)
        INDEX=UIS$GET_WRITING_INDEX(VD_ID,I)    ❸
        TYPE*,INDEX
        J=J+2.0
        ENDDO
        VD_ID2=UIS$CREATE_DISPLAY(1.0,1.0,40.0,40.0,15.0,15.0,VCM_ID)
        WD_ID2=UIS$CREATE_WINDOW(VD_ID2,'SYS$WORKSTATION','WINDOW #2')
```

```
      CALL UIS$SET_WRITING_INDEX(VD_ID2,0,9,2)
      CALL UIS$SET_WRITING_INDEX(VD_ID2,0,10,3)
      CALL UIS$SET_WRITING_INDEX(VD_ID2,0,11,4)
      CALL UIS$SET_WRITING_INDEX(VD_ID2,0,12,5)
      CALL UIS$SET_WRITING_INDEX(VD_ID2,0,13,6)
      TYPE*,'VCM Indexes Used In Virtual Display 2'
      DO I=9,13
      CALL UIS$CIRCLE(VD_ID2,I,21.0,K,10.0)
      INDEX=UIS$GET_WRITING_INDEX(VD_ID2,I)    ❹
      TYPE*,INDEX
      K=K+2.0
      ENDDO

      PAUSE

      END
```

A great deal of information is returned from only three inquiry routines. A call to UIS$GET_COLORS ❶ returns the R, G, and B color component values in the color map entries of the virtual color map.

A call to UIS$GET_HW_COLOR_INFO ❷ returns the number of precision binary bits for R, G, and B color map values; it also returns the total number of hardware color map and reserved entries.

Writing color information must be returned from two program locations. The first call to UIS$GET_WRITING_INDEX within the DO loop ❸ returns all the default writing indices as they are being used in the first virtual display. The second call to UIS$GET_WRITING_INDEX ❹ returns each writing index used to draw graphic objects in the second virtual display.

16.4.2.1 Calling UISGET_COLORS, UISGET_HW_COLOR_INFO, UIS$GET_WRITING_INDEX

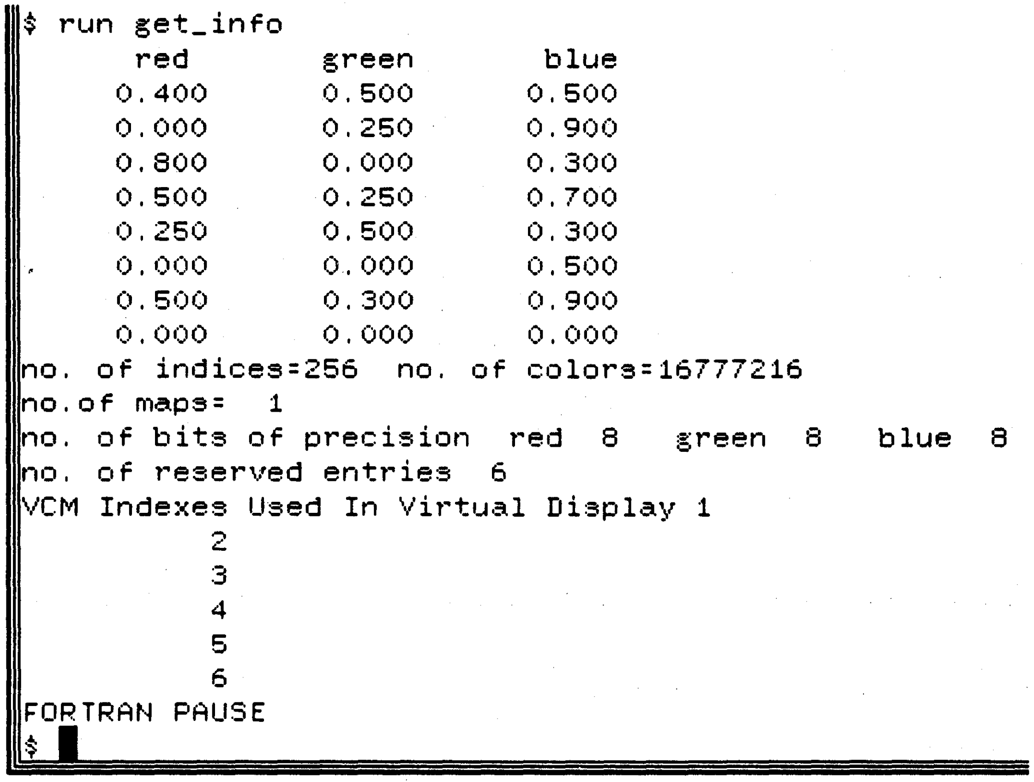
Figure 16-1 shows the information returned in the user emulation window.

16.4.3 Program II—Creating an HSV Color Wheel

NOTE: To abort the demonstration program, type CTRL/C, then EXIT RET. If you are running another graphics process at an independent emulator window, the process will not continue after you exit the COLOR_WHEEL demonstration program. This is known as a side effect.

```
      PROGRAM COLOR_WHEEL
c
c This program draws a color wheel once and then continually
c changes its appearance by updating the virtual color map.
c
      IMPLICIT INTEGER*4(A-Z)
      PARAMETER DISPLAY_SIZE=4.0*2.54
      REAL*4 R,G,B,H,L,S,V,START_DEG,END_DEG
      REAL*4 R_VECTOR(0:255),G_VECTOR(0:255),B_VECTOR(0:255)
      INCLUDE 'SYS$LIBRARY:UISUSRDEF'
```

Figure 16–1 Different Types of Information Returned from Inquiry Routines


```
$ run get_info
      red         green       blue
     0.400        0.500       0.500
     0.000        0.250       0.900
     0.800        0.000       0.300
     0.500        0.250       0.700
     0.250        0.500       0.300
     0.000        0.000       0.500
     0.500        0.300       0.900
     0.000        0.000       0.000
no. of indices=256  no. of colors=16777216
no.of maps=  1
no. of bits of precision  red  8   green  8   blue  8
no. of reserved entries  6
VCM Indexes Used In Virtual Display 1
           2
           3
           4
           5
           6
FORTRAN PAUSE
$
```

ZK-5453-86

```
c
c Find out some information about the workstation color characteristics
c
        CALL UIS$GET_HW_COLOR_INFO(,,INDICES,,MAPS,,,,,RES_INDICES,REGEN)
c
c Only attempt to run this program on color map hardware systems.
c
        IF (MAPS .EQ. 0 .OR. REGEN .NE. UIS$C_DEV_RETRO) STOP
c
c Make the virtual color map size dependent upon the available
c hardware, but no greater than 64 entries
c
        MAP_SIZE=MIN(INDICES-RES_INDICES, 64)
        VCM_ID=UIS$CREATE_COLOR_MAP(MAP_SIZE)
c
c Create the virtual display and a single window
c
        VD_ID=UIS$CREATE_DISPLAY(0.0, 0.0, 1.0, 1.0,
        1                DISPLAY_SIZE, DISPLAY_SIZE, VCM_ID)
        WD_ID=UIS$CREATE_WINDOW(VD_ID, 'SYS$WORKSTATION')
```

```
c
c Establish some attributes for drawing
c
        CALL UIS$SET_ARC_TYPE(VD_ID, 0, 1, UIS$C_ARC_PIE)
        CALL UIS$SET_FONT(VD_ID, 1, 1, 'UIS$FILL_PATTERNS')
        CALL UIS$SET_FILL_PATTERN(VD_ID, 1, 1, PATT$C_FOREGROUND)
c
c Set window background to black and draw wedges of a circle.
c The initial colors of the wedges are determined by traversing
c 360 degrees around the HSV color model, varying H, while S and
c V are both 1.0.
c
        CALL UIS$SET_COLOR(VD_ID, 0, 0.0, 0.0, 0.0)
        DO I=1,MAP_SIZE-1
            START_DEG=(I-1)*(360.0/FLOAT(MAP_SIZE-1))
            END_DEG=START_DEG+(360.0/FLOAT(MAP_SIZE-1))
            CALL UIS$HSV_TO_RGB(START_DEG, 1.0, 1.0, R, G, B)
            CALL UIS$SET_COLOR(VD_ID, I, R, G, B)
            CALL UIS$SET_WRITING_INDEX(VD_ID, 1, 1, I)
            CALL UIS$CIRCLE(VD_ID, 1, 0.5, 0.5, 0.4, START_DEG, END_DEG)
        END DO
c
        V=1.0
c
c The next set of sequential and nested loops
c traverse the HSV color model cone.
c
100     CONTINUE
c
c Vary S from 1.0 to 0.0 in 0.01 increments
c
        DO IS=99,0,-1
        S=FLOAT(IS)/100.0
c
        DO I=1,MAP_SIZE-1
            START_DEG=(I-1)*(360.0/FLOAT(MAP_SIZE-1))
            IF (S .EQ. 0.0) START_DEG=UIS$C_COLOR_UNDEFINED
            CALL UIS$HSV_TO_RGB(START_DEG, S, V,
        1       R_VECTOR(I), G_VECTOR(I), B_VECTOR(I))
        END DO          ! I
        CALL UIS$SET_COLORS(VD_ID, 1, MAP_SIZE-1,
        1       R_VECTOR(1), G_VECTOR(1), B_VECTOR(1))
c
        end do          ! s=1.0,0.0
c
c Vary V from 1.0 to 0.0 in 0.01 increments
c
        DO IV=99,0,-1
        V=FLOAT(IV)/100.0
c
        DO I=1,MAP_SIZE-1
            START_DEG=(I-1)*(360.0/FLOAT(MAP_SIZE-1))
            IF (S .EQ. 0.0) START_DEG=UIS$C_COLOR_UNDEFINED
            CALL UIS$HSV_TO_RGB(START_DEG, S, V,
        1       R_VECTOR(I), G_VECTOR(I), B_VECTOR(I))
        END DO          ! I
        CALL UIS$SET_COLORS(VD_ID, 1, MAP_SIZE-1,
        1       R_VECTOR(1), G_VECTOR(1), B_VECTOR(1))
c
        END DO          ! V=1.0,0.0
```

```
c
c Vary V from 0.0 to 1.0 in 0.01 increments
c
        DO IV=1,100, 1
        V=FLOAT(IV)/100.0
c
        DO I=1,MAP_SIZE-1
            START_DEG=(I-1)*(360.0/FLOAT(MAP_SIZE-1))
            IF (S .EQ. 0.0) START_DEG=UIS$C_COLOR_UNDEFINED
            CALL UIS$HSV_TO_RGB(START_DEG, S, V,
        1       R_VECTOR(I), G_VECTOR(I), B_VECTOR(I))
        END DO          ! I
        CALL UIS$SET_COLORS(VD_ID, 1, MAP_SIZE-1,
        1       R_VECTOR(1), G_VECTOR(1), B_VECTOR(1))
c
        END DO          ! V=0.0,1.0
c
c Vary S from 0.0 to 1.0 in 0.01 increments
c
        DO IS=1,100,1
        S=FLOAT(IS)/100.0
c
        DO I=1,MAP_SIZE-1
            START_DEG=(I-1)*(360.0/FLOAT(MAP_SIZE-1))
            IF (S .EQ. 0.0) START_DEG=UIS$C_COLOR_UNDEFINED
            CALL UIS$HSV_TO_RGB(START_DEG, S, V,
        1       R_VECTOR(I), G_VECTOR(I), B_VECTOR(I))
        END DO          ! I
        CALL UIS$SET_COLORS(VD_ID, 1, MAP_SIZE-1,
        1       R_VECTOR(1), G_VECTOR(1), B_VECTOR(1))
c
        END DO          ! S=0.0,1.0
c
c Repeat HSV color cone traversal indefinitely
c
        GOTO 100
c
        END
```

17 Asynchronous System Trap Routines

17.1 Overview

Frequently, an application program relies on certain run-time events to trigger execution of an application-specific task. Such run-time events can range from a power failure to a missed keystroke. Several UIS routines *enable* this type of behavior for the duration of the program or until the enabling UIS routine is explicitly disabled. Such routines enable the use of asynchronous system trap (AST) routines. This chapter discusses AST routines and how they can be used to perform the following tasks:

- Creating a virtual keyboard
- Using a pointer
- Creating a pointer pattern
- Shrinking a display viewport to an icon
- Resizing a display window
- Closing a display window

AST routines are not limited to the tasks listed here.

17.1.1 Using AST Routines

Certain UIS routines associate, or *bind,* a specific run-time event or action to a subroutine. When this binding occurs, control passes from the main program to a user-written subroutine that then performs some application-specific task. When the subroutine completes execution, control is transferred to the next statement in the main program. However, the association between the run-time event and the execution of the subroutine remains in effect.

If the action occurs again during program execution, the subroutine is recalled. The process executing the main program is suspended when the run-time event occurs and until the subroutine completes execution. Thus, execution of the subroutine occurs asynchronously with respect to execution of the main program. The user-written subroutine is known as an *asynchronous system trap* routine or AST routine.

You can code the AST routine in two ways:

- Within the main program according to the particular programming language conventions
- Separately as a module in a library

If you code the AST routine separately, you must compile and link the library modules with your program.

17.1.2 AST-Enabling Routines

Several UIS routines enable AST routine execution whenever a particular run-time event occurs. The actual event might involve the keyboard or pointer, or the occurrence of a program-related event such as the movement or resizing of a window. *AST-enabling* routines reference AST routines in their argument lists. Table 17–1 lists each AST-enabling routine and the event that triggers AST routine execution.

Table 17–1 AST-Enabling Routines

Routine	Event
UIS$SET_ADDOPT_AST	An additional option is chosen using the human interface.
UIS$SET_BUTTON_AST	A button is depressed or released on a pointer device.
UIS$SET_CLOSE_AST	A display window is deleted with the human interface.
UIS$SET_TB_AST	A digitizer is moved within a specified data region on the tablet.
UIS$SET_EXPAND_ICON_AST	An icon is expanded to display viewport with the user interface.
UIS$SET_GAIN_KB_AST	A virtual keyboard is bound to a physical keyboard.
UIS$SET_KB_AST	A key is pressed.
UIS$SET_LOSE_KB_AST	A virtual keyboard is disconnected from a physical keyboard.
UIS$SET_MOVE_INFO_AST	A window is moved in the virtual display.
UIS$SET_POINTER_AST	A pointer moves into or exits an area of the virtual display.
UIS$SET_RESIZE_AST	A display window is resized with the human interface.
UIS$SET_SHRINK_ICON_TO_AST	A display viewport is shrunk with the human interface.

17.2 Keyboard and Pointer Devices

Keyboard and pointer devices are resources for use within your application program. The keyboard and pointer are mentioned here to illustrate routines that use input from such workstation peripheral devices during application program execution. You can use keyboard and pointer devices effectively in conjunction with AST routines.

17.2.1 Using AST Routines with Virtual Keyboards

You can use your keyboard as a virtual device with characteristics transportable from virtual display to virtual display. In this way, you can create an unlimited number of virtual devices (subject to system and process resources) with different characteristics and associate each with any virtual display you choose.

To use AST routines with virtual keyboards, follow these steps with the routines listed in Table 17–2.

1 Create virtual keyboard(s).

  You can create an unlimited number of virtual keyboards with UIS$CREATE_KB.

2 Bind the virtual keyboard to a display screen.

  Once you create a virtual keyboard, you must bind it to a specified display window with UIS$ENABLE_VIEWPORT_KB or UIS$ENABLE_KB. These routines also define how the physical and virtual keyboards are assigned to each other.

  If your display screen contains one or more display viewports and you have assigned virtual keyboards to their associated display windows, to keep track of which viewport is active use the CYCLE key to move from viewport to viewport through the assignment list.

  A viewport is active when the KB icon background color on the viewport is highlighted. The physical keyboard is now assigned to a virtual keyboard. The virtual keyboard and all enabled characteristics can then be used with the physical keyboard. You can bind more than one display window to the same virtual keyboard. In this case, all KB icons are highlighted simultaneously when you assign windows to a physical keyboard.

3 Enable the AST routines.

  Now that the virtual keyboard is created and bound to a display window, you must use UIS$SET_KB_AST to associate the keystroke with the action taken by a subroutine.

Table 17-2 AST Routines and Descriptions

AST ROUTINE	Description
UIS$CREATE_ KB	Creates virtual keyboards
UIS$ENABLE_ VIEWPORT_KB	Adds the display window to the assignment list. Use the CYCLE key to move from viewport to viewport.
UIS$ENABLE_ KB	Places the display window at the top of the assignment list. Makes the viewport active. Use the CYCLE key to move to other viewports.
UIS$DISABLE_ VIEWPORT_KB	Removes a display window from the assignment list. Use UIS$ENABLE_VIEWPORT_KB or UIS$ENABLE_KB to make the viewport active.
UIS$DISABLE_ KB	Places a display window at the bottom of the assignment list. Press the CYCLE key to make the viewport active.
UIS$SET_KB_ AST	Associates a keystroke with subroutine action.

17.2.2 Controlling Keyboards

After you create the virtual keyboard, your application can verify successful connection with the physical keyboard. You can be notified when such connections are made or broken.

Connecting and disconnecting virtual keyboards can occur many times within your application program. Whenever a virtual keyboard is disconnected or lost, you might want your program to initiate some action through a subroutine. For example, when a virtual keyboard is disconnected, UIS$SET_GAIN_KB_AST and UIS$SET_LOSE_KB_AST enable AST routines that allow your program to perform housekeeping functions such as deleting unused virtual keyboards, display windows, and display viewports.

The following routines control the keyboard:

- UIS$SET_KB_ATTRIBUTES—Assigns characteristics to specific virtual keyboards.
- UIS$TEST_KB—Verifies the connection between a specified virtual keyboard and the physical keyboard.
- UIS$DELETE_KB—Deletes a virtual keyboard.

17.2.3 Program Development

Programming Objective

To type keyboard characters directly to the virtual display using AST routines.

Programming Tasks

1 Declare the subroutine and the appropriate variables to be included in the COMMON statement.

2 Create a virtual display.

3 Create a virtual keyboard.

4 Create a display window and viewport.

5 Bind the virtual keyboard to the display window.

6 Use UIS$SET_KB_AST to enable keyboard AST routines.

7 Create a subroutine to send each keystrike to the virtual display.

```
PROGRAM AST
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
LOGICAL*1 KEYBUF(4)
EXTERNAL KEYSTRIKE  1
COMMON KB_ID,VD_ID,KEYBUF,WD_ID,COUNT    2

VD_ID=UIS$CREATE_DISPLAY(1.0,1.0,31.0,31.0,20.0,5.0)
KB_ID=UIS$CREATE_KB('SYS$WORKSTATION')    3

WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','KEYBOARD AST')
CALL UIS$ENABLE_VIEWPORT_KB(KB_ID,WD_ID)    4

CALL UIS$SET_ALIGNED_POSITION(VD_ID,1,1.0,30.0)

COUNT=0

CALL UIS$SET_KB_AST(KB_ID,KEYSTRIKE,0,KEYBUF)    5
CALL SYS$HIBER()    6

END

SUBROUTINE KEYSTRIKE     7
IMPLICIT INTEGER(A-Z)
INCLUDE 'SYS$LIBRARY:UISENTRY'
INCLUDE 'SYS$LIBRARY:UISUSRDEF'
LOGICAL*1 KEYBUF(4)
COMMON KB_ID,VD_ID,KEYBUF,WD_ID,COUNT    8
STRUCTURE/TEXT/     9
INTEGER*2 BUFLEN,BUFCODE
INTEGER*4 BUFADR
END STRUCTURE

RECORD/TEXT/DESC     10

DESC.BUFLEN=1
DESC.BUFADR=%LOC(KEYBUF)  11
STATUS=UIS$TEST_KB(KB_ID)    12

CALL UIS$SET_FONT(VD_ID,1,2,'MY_FONT_13')

IF ((COUNT .EQ. 60) .OR. (KEYBUF(1) .EQ. 13)) THEN  13

CALL UIS$NEW_TEXT_LINE(VD_ID,2)

COUNT=0
ELSE
CALL UIS$TEXT(VD_ID,2,DESC)
COUNT=COUNT+1
END IF

RETURN

END
```

Use the EXTERNAL statement to declare the name of the AST routine KEYSTRIKE ❶. The EXTERNAL statement defines the symbolic name of the routine as an address. You can then use the routine name as an argument in a parameter list, as in the **astadr** argument of an AST-enabling routine.

The COMMON statement allows certain variables used in both program units (the main program and the subroutine) to share the same storage area ❷ ❸. You can use either the COMMON statement or the **astprm** argument in the AST-enabling routine to pass data to the AST routine.

The virtual keyboard is created ❸ and bound to a display window ❹.

UIS$SET_KB_AST is the AST-enabling routine that references the subroutine KEYSTRIKE ❺ ❼. Note that there is no separate call to the subroutine KEYSTRIKE.

Whenever you press a key, the ASCII character code for that character is stored in the variable *keybuf* ❺ and subroutine KEYSTRIKE is executed. The subroutine KEYSTRIKE is an AST routine.

The subroutine KEYSTRIKE retrieves the character code stored in the variable *keybuf*. The data structure TEXT, a character string descriptor, is created ❾. DESC, the variable that denotes a record, is defined with the same structure as TEXT ❿. The address of *keybuf* is assigned to a longword in the descriptor ⓫. The subroutine KEYSTRIKE uses UIS$TEXT to write the character to the virtual display.

After the AST routine completes execution, control returns to the next statement in the main program. The next statement calls the SYS$HIBER system service ❻, which allows the process to remain inactive until the next time the AST routine is executed (when a key is pressed).

The AST routine KEYSTRIKE also verifies that the virtual and physical keyboards are connected ⓬.

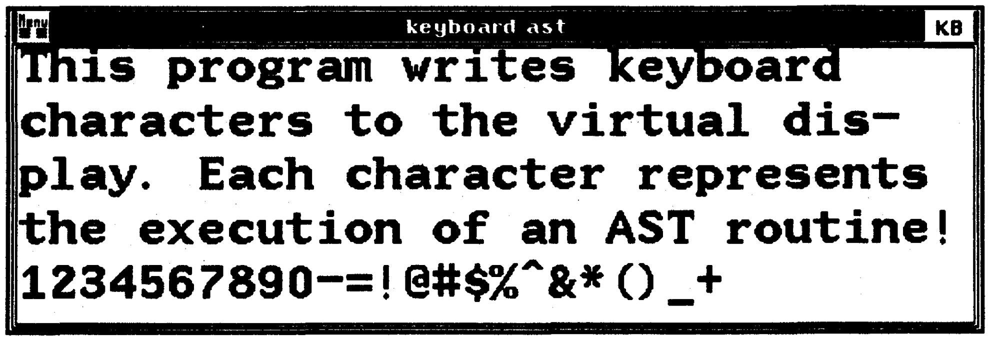
Whenever column 60 is reached or the RETURN key is pressed ⓭, text output moves to the next line. The ASCII character code for the RETURN key is 13.

17.2.4 Calling Keyboard Routines

The program AST creates a viewport to which characters are written as shown in Figure 17–1.

To display characters after you run the AST program, make sure the pointer device is located within the bounds of the display viewport. Press the leftmost button on the mouse. The keyboard AST display is now activated. At this point, you can begin typing the characters and they are displayed on the viewport.

Figure 17–1 Writing Characters to a Display Viewport



```
This program writes keyboard
characters to the virtual dis-
play. Each character represents
the execution of an AST routine!
1234567890-=!@#$%^&*()_+
```

ZK-4561-85

17.2.5 Using AST Routines with Pointer Devices

Pointer routines allow the pointer to act as an input device to your application program. Typically, application programs use such data to keep track of the location of the pointer device in the virtual display or the location of a specified rectangle in the virtual display. AST routines provide an effective way to use pointers in this manner.

17.2.5.1 Mouse

You can use the mouse with AST routines to return to the application status information about mouse location.

17.2.5.2 Tablet

The digitizer is another pointing device. The tablet consists of a puck or stylus and a tablet. You can use a digitizer only with a tablet. You cannot use a mouse as a data digitizer. If you attempt to digitize with a mouse, UIS will report an error.

Digitizing with a Tablet

To digitize with a tablet, establish a region on the tablet called the data rectangle, where digitizing is active. If you do not specify a data rectangle, the whole tablet is used.

Only one data digitizing region can be active at a time.

The pointer position on the tablet is available to the digitizing AST routine. If the pointer is within the data rectangle, the AST routine is executed.

Only one image can own the tablet at a time. When a process connects to the tablet, the system hardware cursor is turned off and the connected process receives all the input from the tablet device. The process must use a software cursor to track the pointer in a window. The process owns the tablet until it makes a call to UIS$ENABLE_TB to disconnect itself from the tablet.

Terminating Data Digitizing

Only the process that issues the data digitizing request can change or cancel the request. If the process is deleted and the channel deassigned, data digitizing is immediately canceled if a request is still outstanding.

Only one data digitizing region can be active at a time. Attempts by other processes to initiate fail if another process has already declared a digitizing region.

17.2.5.3 Step 1—Create an AST Routine

You must write a program that includes an AST subroutine that performs a task. Typically, AST subroutines perform inquiry functions and return pointer information such as location to the main program. Table 12–1 lists pointer routines. You are not restricted to using AST routines in this manner. For example, you can use AST routines with pointers to create menus.

17.2.5.4 Step 2—Enable the AST Routine

The AST routine executes whenever a specific run-time event occurs. To enable this behavior, you must include an AST-enabling routine in the main program. Table 17–3 lists pointer AST-enabling routines.

Table 17–3 Pointer AST-Enabling Routines

Routine	Run-Time Event
UIS$SET_BUTTON_AST	The button on the pointer device is depressed.
UIS$SET_TB_AST	The digitizer is moved within a specified data region on the tablet.
UIS$SET_POINTER_AST	The pointer is moved into a specified region of virtual display.

17.2.6 Programming Options

Many graphics applications use the pointer position and movement to draw objects on the display screen. Graphics routines use this information to generate objects.

Pointer Movement

Many application programs must know where the pointer is. For example, the program might need to perform some type of action whenever the pointer moves within certain regions of the virtual display. Use the AST-enabling routine UIS$SET_POINTER_AST whenever pointer movement is important.

Pointer Position

Your application might need to establish pointer position in world coordinates. In addition, UIS$SET_POINTER_POSITION returns a status value.

Pointer Pattern

You can change the appearance of the pointer cursor with UIS$SET_POINTER_PATTERN. Normally, this cursor appears as an arrow on the display screen. The pointer cursor, or *pattern*, represents bit settings within an array of 16 words. To choose your own pointer pattern, for each word in the array, assign a value that sets the desired bits for the new pattern.

Optionally, you can request that the pointer be bound to the region specified in the UIS$SET_POINTER_PATTERN call. When this region is unoccluded, the pointer pattern cannot exit after it has been positioned within the region. The cursor can leave the bound region if it becomes occluded.

Tablet Information

Currently, the routines UIS$GET_TB_INFO and UIS$GET_TB_POSITION return information about tablet characteristics and position, respectively.

17.2.7 Program Development

Programming Objective

To change the default pointer pattern to a cross-hair.

Programming Tasks

1 Declare the subroutine and the appropriate variables in the COMMON statement.

2 Create a virtual display.

3 Create a display window and viewport.

4 Use UIS$SET_POINTER_AST to enable the pointer AST routine.

5 Create a subroutine that defines the new cursor pattern.

```
        PROGRAM PATTERN
        IMPLICIT INTEGER(A-Z)
        EXTERNAL FIGURE  1
        INCLUDE 'SYS$LIBRARY:UISENTRY'
        INCLUDE 'SYS$LIBRARY:UISUSRDEF'
        COMMON VD_ID,WD_ID
        VD_ID=UIS$CREATE_DISPLAY(-1.0,-1.0,30.0,30.0,20.0,20.0)
        WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION',
    2           'POINTER PATTERN') 2

        CALL UIS$SET_POINTER_AST(VD_ID,WD_ID,FIGURE,0)  3

        CALL SYS$HIBER()

        END
        SUBROUTINE FIGURE
        IMPLICIT INTEGER(A-Z)
        INTEGER*2 CURSOR(16)  4
        INCLUDE 'SYS$LIBRARY:UISENTRY'
        INCLUDE 'SYS$LIBRARY:UISUSRDEF'
        COMMON VD_ID,WD_ID

        DATA CURSOR/7*896,65535,8*896/   5

        CALL UIS$SET_POINTER_PATTERN(VD_ID,WD_ID,CURSOR,,8,8) 6

        RETURN
        END
```

In this program, neither world coordinates nor dimensions of the display viewport are specified for the display window ❷. As a result, the display window maps the entire virtual display space and the display viewport size defaults to the dimensions specified in UIS$CREATE_DISPLAY.

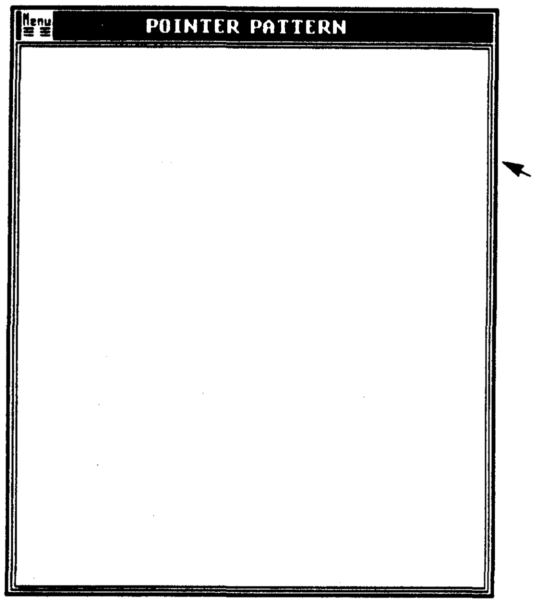
The subroutine FIGURE is called whenever the pointer lies within the specified area of the display window. In the main program, the subroutine is declared as an external procedure ❶. UIS$SET_POINTER_AST, the AST-enabling routine for the pointer devices, is called ❸. Because no rectangle is specified, the subroutine FIGURE is executed whenever the pointer is within the display window.

The array CURSOR is declared in the subroutine FIGURE ❹ and contains 16 elements. Each array element is declared as a word and is, therefore, 16 bits long. Imagine the array as a 16 by 16-bit pattern, or matrix. Each array element ❺ is assigned a value that sets certain bits in the matrix to *1*. The matrix represents the bitmap image of the new cursor pattern. The call to UIS$SET_POINTER_PATTERN references the new cursor pattern and the exact bit in the new cursor pattern used to calculate current pointer positon ❻.

17.2.8 Calling UIS$SET_POINTER_AST and UIS$SET_POINTER_PATTERN

When you run the program PATTERN, the display viewport is created. The pointer lies outside the display viewport, and the default pointer pattern is in effect as shown in Figure 17-2.

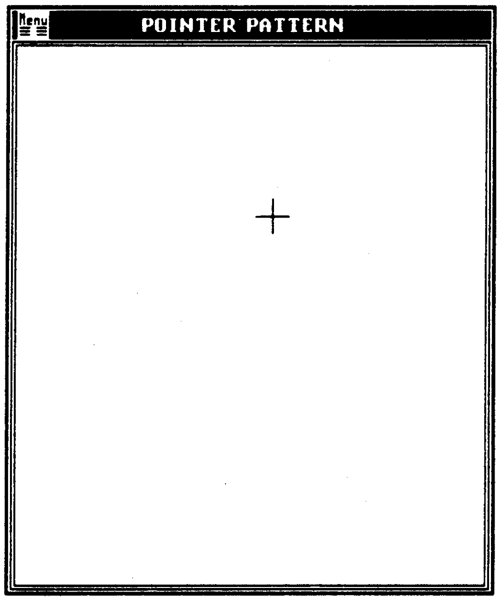
Figure 17-2 Default Pointer Pattern



ZK-4614-85

The process that executes the main program is hibernating, or waiting for you to move the pointer. As you can see in Figure 17-3, when you move the pointer within the display window, the pointer pattern changes from an arrow to a cross.

Figure 17-3 New Pointer Pattern



ZK-4562-85

17.3 Manipulating Display Windows and Viewports

Default Shrinking Operation

By default, you use the Window Options Menu to shrink viewports. When you choose the Shrink to an Icon menu item, UIS$SHRINK_TO_ICON is called. To expand icons to viewports with the user interface, place the cursor in the icon and press the pointer button.

Default Resizing and Closing Operations

By default, you use the Window Options Menu to resize and close display windows. When you choose the Change the Size menu item, you call UIS$RESIZE_WINDOW, which accepts the world coordinate values of the newly resized window.

You also use the Window Options Menu to close display windows. When you choose the Delete menu item, you call UIS$CLOSE_WINDOW, which, in turn, calls SYS$EXIT system service. SYS$EXIT performs image rundown and deletes the process that owns the image.

17.3.1 Using AST Routines to Modify the Window Options Menu

Certain UIS routines can override the default actions listed in the Window Options Menu and enable user-written shrinking, expanding, resizing, and closing AST routines that are activated whenever you choose the Shrink to an Icon, Change the size, or Delete menu items.

Using AST routines to modify the Window Options Menu requires two steps:

1. Create an AST routine.
2. Enable the AST routine.

17.3.1.1 Step 1—Create an AST Routine

To override one of the default actions in the Window Options Menu, you must write a program that includes an AST routine. When you execute the program and initiate the action through the user interface, the default action is no longer performed automatically.

You can code your AST routine to perform any action. Often, you modify the action of a menu item by adding additional actions to the default. To do so, include in your AST routine a call to UIS$RESIZE_WINDOW in addition to code to perform any other special features. When the program executes, the AST routine performs the resize as well as any other additional actions. Table 17-4 lists the task you want to perform and the corresponding UIS routine you should include in your subroutine.

Table 17-4 Tasks and Corresponding UIS Routines

Routine	Task
UIS$CLOSE_WINDOW	Close or delete a window
UIS$EXPAND_ICON	Expand an icon[1]
UIS$RESIZE_WINDOW	Resize a viewport
UIS$SHRINK_TO_ICON	Shrink a viewport

[1]Not listed in the Window Options Menu.

17.3.1.2 Step 2—Enable the AST Routine

Your AST routine should execute whenever you want to override the default features listed in the Window Options Menu. To execute the AST routine, a run-time event must occur to trigger it. Therefore, you must include an appropriate AST-enabling routine in your main program. Table 17-5 lists window AST-enabling routines that trigger AST routine execution for various run-time events.

Table 17–5 AST-Enabling Routines that Trigger AST Routine Execution

Routine	Run-Time Event
UIS$SET_CLOSE_AST	The Delete menu item is chosen using the user interface.
UIS$SET_EXPAND_ICON_AST	The pointer pattern is placed on an icon and the pointer button is depressed.
UIS$SET_RESIZE_WINDOW_AST	The Change the Size menu item is chosen using the user interface
UIS$SET_SHRINK_TO_ICON_AST	The Shrink to an Icon menu item is chosen using the user interface.

17.3.2 Programming Options

Table 17–6 lists programming options for which you can also enable AST routine execution.

Table 17–6 AST Routine Execution Programming Options

Routine	Programming Option
UIS$SET_SHRINK_TO_ICON_AST	Shrink viewports to icons
UIS$SET_EXPAND_ICON_AST	Expand icons to viewports
UIS$RESIZE_WINDOW	Resize display windows
UIS$SET_CLOSE_AST	Close display windows

Shrinking Viewports to Icons

To override the default display viewport shrinking operation, call the AST-enabling routine UIS$SET_SHRINK_TO_ICON_AST in your main program. Your AST routine will contain UIS$SHRINK_TO_ICON, which specifies icon attributes. Shrinking viewports to icons occurs as a four-step process as follows:

1. Initiate the shrinking operation with the user interface.
2. Use the invisible attribute to move the viewport offscreen.
3. The subroutine creates a small virtual display and viewport with no banner, the actual icon.
4. The subroutine using UIS$SHRINK_TO_ICON associates the icon name with the virtual display identifier of the offscreen viewport.

Expanding Icons to Display Viewports

To override the default icon expansion operation, call the AST-enabling routine UIS$SET_EXPAND_ICON_AST in your main program. Include UIS$EXPAND_ICON in your AST routine to specify viewport attributes.

Resizing Display Windows

To override the default display window resize operation, call the AST-enabling routine UIS$SET_RESIZE_AST in your main program. Resizing occurs as a three-step process as follows:

1 Initiate the resizing operation with the user interface.

2 The user interface returns values to the addresses specified in UIS$SET_RESIZE_AST.

3 The AST routine is called.

Your AST routine includes a call to UIS$RESIZE_WINDOW, which redefines the default resize behavior as follows:

- Absolute position—You can specify an absolute position, that is, a device coordinate position on the physical screen for the newly resized display viewport.
- Size—You can specify the dimensions of all newly resized display viewports. All subsequent display viewports are created with these dimensions.
- World coordinate space—You can specify the world coordinate space as the original display window. Typically, the coordinates you specify here match the world coordinates of the original display window. However, is not always the case. If your original display window views a portion of the virtual display, you can view more or less of the virtual display depending on the world coordinate range you specify.

Closing Display Windows

To override the default close display window operation, call the AST-enabling routine UIS$SET_CLOSE_AST in your main program.

Instructions that you include in your AST routine override the default window closing behavior. Closing display windows occurs as a two-step process as follows:

1 Choose the Delete menu item in the Window Option Menu.

2 Call the AST routine.

17.3.3 Program Development

Programming Objective

To modify the display window shrinking, expanding, resizing, and closing operations listed in the Window Options Menu, whenever the Shrink to Icon, Change the Size, or Delete menu item is chosen.

Programming Tasks

1 Declare the subroutines and the appropriate variables in the COMMON statement.

2 Create a virtual display.

3 Create a display window and viewport.

4 Draw two ellipses and a circle.

5. Use UIS$SET_SHRINK_TO_ICON_AST and UIS$SET_EXPAND_ICON_AST to enable viewport shrinking and icon expansion AST routines.
6. Use UIS$SET_RESIZE_AST UIS$SET_CLOSE_AST to enable window resizing and closing AST routines.
7. Create viewport shrinking and icon expansion AST routines.
8. Create window resizing and closing AST routines.

NOTE: Before you run the OVERRIDE demonstration program, invoke the indirect command file DEFFONT.COM to define fonts. Also, you must link the routines RESIZER, SHRINKER, EXPANDER, and CLOSER with the main module OVERRIDE.

```
      PROGRAM OVERRIDE
      IMPLICIT INTEGER(A-Z)
      EXTERNAL RESIZER,SHRINKER,EXPANDER,CLOSER
      INCLUDE 'SYS$LIBRARY:UISENTRY'
      INCLUDE 'SYS$LIBRARY:UISUSRDEF'
      COMMON VD_ID,VD_ID2,WD_Id,WD_ID2,NEW_ABS_X,NEW_ABS_Y

      VD_ID=UIS$CREATE_DISPLAY(0.0,0.0,50.0,50.0,10.0,10.0)  1
      WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','USER')    2

      CALL UIS$ELLIPSE(VD_ID,0,0.0,20.0,15.0,20.0)

      CALL UIS$CIRCLE(VD_ID,1,40.0,20.0,25.0)

      CALL UIS$ELLIPSE(VD_ID,0,80.0,20.0,15.0,20.0)

      CALL UIS$SET_SHRINK_TO_ICON_AST(WD_ID,SHRINKER)  3

      CALL UIS$SET_EXPAND_ICON_AST(WD_ID,EXPANDER)  4

      CALL UIS$SET_CLOSE_AST(WD_ID,CLOSER,0)     5

      CALL UIS$SET_RESIZE_AST(VD_ID,WD_ID,RESIZER,0,NEW_ABS_X,NEW_ABS_Y,
     2        NEW_WIDTH,NEW_HEIGHT,NEW_WC_X1,NEW_WC_Y1,NEW_WC_X2,
     2        NEW_WC_Y2)                                               6

      CALL SYS$HIBER()                   7

      TYPE *,'DISPLAY WINDOW HAS BEEN SUCCESSFULLY CLOSED'     8

      END

      SUBROUTINE RESIZER              9
      IMPLICIT INTEGER(A-Z)
      COMMON VD_ID,VD_ID2,WD_ID,wd_id2,NEW_ABS_X,NEW_ABS_Y

      CALL UIS$RESIZE_WINDOW(VD_ID,WD_ID,NEW_ABS_X,NEW_ABS_Y,,,,,,)  10

      RETURN
      END

      SUBROUTINE SHRINKER          11
      IMPLICIT INTEGER(A-Z)
      INCLUDE 'SYS$LIBRARY:UISENTRY'
      INCLUDE 'SYS$LIBRARY:UISUSRDEF'
      COMMON VD_ID,VD_ID2,WD_ID,WD_ID2,NEW_ABS_X,NEW_ABS_Y

      STRUCTURE/AWAY/                        12
      INTEGER*4 CODE1
      INTEGER*4 ATTR1
      INTEGER*4 CODE2
      INTEGER*4 ATTR2
      INTEGER*4 END_LIST
      END STRUCTURE

      RECORD/AWAY/WINDOW                 13
```

```
WINDOW.CODE1=WDPL$C_PLACEMENT            14
WINDOW.ATTR1=WDPL$M_INVISIBLE              15
WINDOW.CODE2=WDPL$C_END_OF_LIST         16
CALL UIS$MOVE_VIEWPORT(WD_ID,WINDOW)     17

WINDOW.CODE1=WDPL$C_ATTRIBUTES          18
WINDOW.ATTR1=WDPL$M_NOBANNER          19
WINDOW.CODE2=WDPL$C_END_OF_LIST                  20

VD_ID2=UIS$CREATE_DISPLAY(0.0,0.0,5.0,5.0,2.54,2.54)  21
WD_ID2=UIS$CREATE_WINDOW(VD_ID2,'SYS$WORKSTATION',,
2        ,,,,,,WINDOW)                                  22
CALL UIS$SET_FONT(VD_ID2,0,2,'MY_FONT_5')
CALL UIS$TEXT(VD_ID2,2,'USER',0.5,3.5)                23

ICON_FLAGS=UIS$M_ICON_DEF_BODY                  24
CALL UIS$SHRINK_TO_ICON(WD_ID,WD_ID2,ICON_FLAGS)          25

RETURN
END

SUBROUTINE EXPANDER
IMPLICIT INTEGER(A-Z)
COMMON VD_ID,VD_ID2,WD_ID,WD_ID2,NEW_ABS_X,NEW_ABS_Y

CALL UIS$EXPAND_ICON(WD_ID,WD_ID2)

RETURN
END

SUBROUTINE CLOSER                        26
IMPLICIT INTEGER(A-Z)
COMMON VD_ID,VD_ID2,WD_ID,WD_ID2,NEW_ABS_X,NEW_ABS_Y

CALL UIS$ERASE(VD_ID)
CALL UIS$DELETE_WINDOW(WD_ID)                  27

CALL UIS$DELETE_DISPLAY(VD_ID)           28

CALL SYS$WAKE(,)                   29

RETURN
END
```

The main program OVERRIDE creates a virtual display 1 and a display window 2. The world coordinate space of the display window is a portion of the virtual display; the display window contains only those objects in the virtual display that lie within it.

A circle is drawn between two ellipses in the virtual display and appears in the display window and its associated display viewport.

Four AST-enabling routines, UIS$SET_SHRINK_TO_ICON_AST, UIS$SET_EXPAND_ICON_AST, UIS$SET_CLOSE_AST and UIS$SET_RESIZE_AST, 3 4 5 6 are called. The main program executes until the call to SYS$HIBER is reached 7.

Use the pointer to invoke the Window Options Menu from the MENU icon in the viewport WINDOW. Choose the menu item Change the Size. Perform the following procedure:

1 Move the pointer to one of the flashing dots on the border of the viewport.

2 Press the button and the border of the display viewport is highlighted.

3 Hold the button down and move the pointer until the stretchy box is the desired size and release the pointer button.

The call to UIS$RESIZE_WINDOW 10 in the subroutine RESIZER 9 modifies the default resize behavior. UIS$RESIZE_WINDOW specifies the world coordinates of the existing virtual display as the world coordinates for all newly resized display windows. Therefore, a newly resized window always displays the entire virtual display space. If the aspect ratios of the virtual display and the resized display viewport are not equal, graphic objects are scaled.

The subroutine SHRINKER 11 modifies the default shrinking behavior. The window attributes data structure AWAY is created 12. A record WINDOW is defined to have the structure of AWAY 13. The fields of record WINDOW are assigned values 14 15 16. Note the use of the invisible placement attribute. A call to UIS$MOVE_VIEWPORT 17 references the display window identifier of the existing viewport and the current window attributes. The viewport is moved offscreen.

New window attribute values are assigned 18 19 20 to the fields of the record WINDOW.

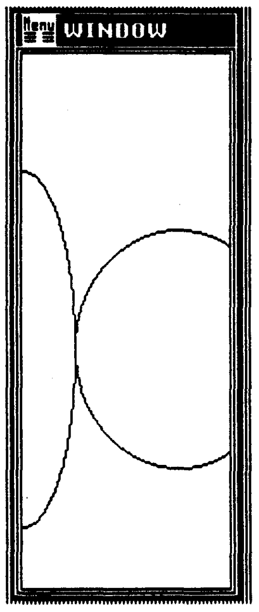
A virtual display and display window are created for the icon 21 22. UIS$TEXT draws the character string in the icon 23. The flag UIS$M_ICON_DEF_BODY sets the appropriate bit in the mask **icon_flags** 24. When this bit is set, the area of the icon becomes a button AST region (for later icon expansion). UIS$SHRINK_TO_ICON 25 associates the display window identifiers of the existing viewport and the icon.

The subroutine CLOSER 26 overrides the default window closing behavior by deleting the display window 27, display viewport, and the virtual display 28. The process that owns the main program is awakened 29. The main program continues execution with the next statement after the call to SYS$HIBER 8, types the message "Display window has been successfully closed," and terminates.

17.3.4 Calling UIS$SET_RESIZE_AST

When the main program executes, a display window and its associated display viewport appear on the display screen as shown in Figure 17–4.

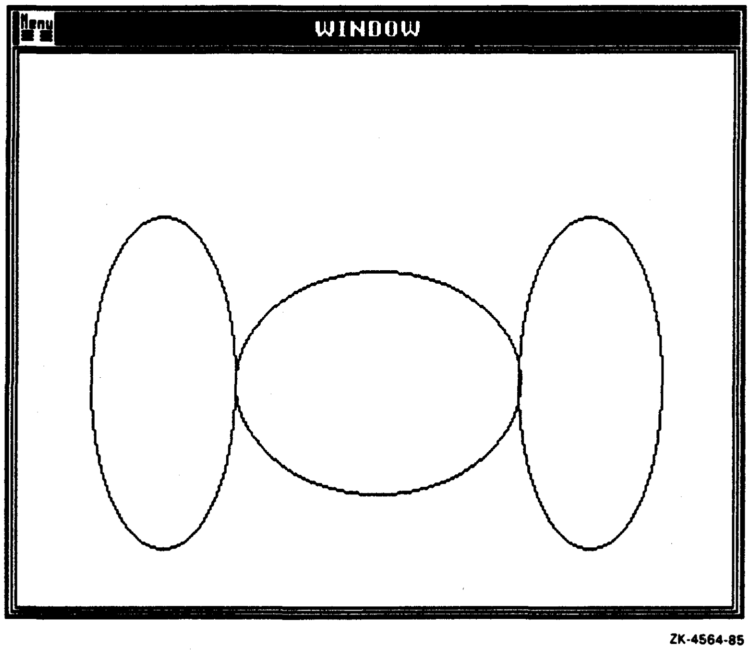
Figure 17–4 Unresized Window and Viewport



ZK-4563-85

When you select the menu item Change the Size, the display window and viewport are resized as shown in Figure 17–5.

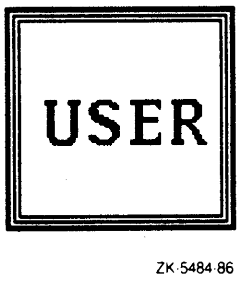
Figure 17–5 Resized Window and Viewport



17.3.5 Calling UIS$SET_SHRINK_TO_ICON_AST

When you select the menu item Shrink to Icon, the display viewport is replaced with a user-defined icon as shown in Figure 17–6.

Figure 17–6 Icon



17.3.6 Calling UIS$SET_CLOSE_AST

When you select the menu item Delete, the display viewport, window, and virtual display are deleted and the message "Display window has been successfully closed" is written to the terminal emulation window.

Part III UIS Routine Descriptions

18 UIS Routine Descriptions

18.1 Overview

Each UIS and UISDC routine in Parts III and IV of this book is documented in a structured format. This section discusses the main headings of this format, the information presented under each heading, and the format used to present the information.

The purpose of this section, therefore, is to explain where to find information and how to read it correctly, not how to use it.

Some main headings in the routine template contain information that requires no further explanation beyond what is given in Table 18–1. However, the following main headings contain information that does require additional discussion; this discussion takes place in the remaining subsections of this section.

- Format heading
- Returns heading
- Arguments heading

Table 18–1 lists the main headings in the UIS routines template.

Table 18–1 Main Headings in the Routine Template

Main Heading	Required or Optional	Description
Routine Name	Required	The routine entry point name is usually, though not always, followed by the English name of the routine.
Routine Overview	Required	The routine overview appears directly below the routine name and explains, usually in one or two sentences, what the routine does.
Format	Required	The format gives the routine entry point name and the routine argument list.
Returns	Required	The returns heading explains what information the routine will return.
Arguments	Required	The arguments heading gives detailed information about each argument.

Table 18-1 (Cont.) Main Headings in the Routine Template

Main Heading	Required or Optional	Description
Description	Optional	The description section contains information about specific actions the routine can take, such as: interaction between routine arguments; operation of the routine within the context of VAX/VMS; user privileges needed to call the routine; system resources used by the routine; and user quotas that can affect the operation of the routine. Note that restrictions on the use of the routine are always discussed first; for example, any required user privileges or necessary system resources are explained first. For some simple routines, a description section is not necessary because the routine overview carries the needed information.
Examples	Optional	This section contains programming examples that illustrate the use of the routine. An explanation of the example is also given. **Note:** All examples have been tested and should run when compiled (or assembled) and linked.
Screen Output	Optional	The screen output heading contains either an actual display produced by the routine or information that the routine normally returns to the program. Note that in many instances screen output contains annotations that serve only to explain the information returned. For example, UIS$GET_POSITION returns information about the current text position along the actual path. This information is displayed and described as an example of the kind of data that can be returned. In many cases, for example, the inquiry routines, the displayed information is formatted with headings and annotations for presentation in this manual only.
Illustration	Optional	The illustration heading contains artwork that describes how to use the routine, how the routine functions, or what kind of information to expect from it. The illustrations *might* or *might not* be annotated.

18.1.1 Format Heading

The following types of information can be present in the format heading:

- Procedure call format

- Explanatory text

Procedure Call Format

The procedure call format ensures that a routine call conforms to the procedure call mechanism described in the *VAX Procedure Calling and Condition Handling Standard;* for example, an entry mask is created, registers are saved, and so on.

Procedure call formats can appear in many forms. The following four examples illustrate the meaning of syntactical elements such as brackets and commas. General rules of syntax governing how to use procedure call formats are listed in Table 18-2.

Example 1

This example illustrates the standard representation of optional arguments and best describes the use of commas as delimiters. Arguments enclosed within square brackets are optional, but if an optional argument other than a trailing optional argument is omitted, you must include a comma as a delimiter for the omitted argument.

```
ENTRY-POINT-NAME    arg1 [,[arg2 [,arg3]]
```

Typically, VAX RMS system routines use this format where up to three arguments can appear in the argument list.

Example 2

When the argument list contains three or more optional arguments, the syntax does not provide enough information. If the optional arguments **arg3** and **arg4** are omitted and the trailing argument **arg5 is** specified, commas **must** be used to delimit the positions of the omitted arguments.

```
ENTRY-POINT-NAME     arg1 ,arg2 ,[arg3] ,nullarg [,arg4] [,arg5]
```

Typically, VAX/VMS system services, utility routines, and VAX Run-Time Library routines contain call formats with more than three arguments.

Example 3

In the following call format example, the trailing four arguments are optional as a group; that is, either you specify **arg2**, **arg3**, **arg4**, and **arg5** or none of them. Therefore, if you do not specify optional arguments, you do not have to use commas to delimit unoccupied positions.

However, if you specify a hypothetical required argument or if you specify a separate optional argument after **arg5**, you must use commas when **arg2**, **arg3**, **arg4**, and **arg5** are omitted.

```
ENTRY-POINT-NAME     arg1 [,arg2 ,arg3 ,arg4 ,arg5]
```

Example 4

In the following example, you can specify **arg2** and omit **arg3**. Whenever you specify **arg3**, however, you **must** specify **arg2**.

```
ENTRY-POINT-NAME  arg1 [,arg2 [,arg3]]
```

Explanatory Text

Explanatory text can follow one or both of the above formats. Explanatory text is present only when it is needed to clarify the format. For example, if the arguments are optional, the call format indicates that by enclosing them in brackets ([ ]). However, brackets alone cannot convey all the important information that might apply to optional arguments. For example, in some routines with many optional arguments, if one optional argument is selected, another optional argument must also be selected. In such cases, text following the format clarifies this fact.

Table 18-2 General Rules of Syntax

Element	Syntax Rule
Entry point names	Entry point names are always shown in uppercase characters.
Argument names	Argument names are always shown in lowercase characters.
Spaces	Use one or more spaces between the entry point name and the first argument, and between each argument.
Braces	Braces surround two or more arguments. You must choose one of the arguments.
Brackets ([ ])	Brackets surround optional arguments. Note that commas can also be optional (see the comma element).
Commas	Between arguments, the comma always follows the space. If the argument is optional, the comma appears inside the brackets or outside the brackets, depending on the position of the argument in the list and on whether surrounding arguments are optional or required.
Null arguments	A null argument is a place-holding argument. It is used for either of the following reasons: **1** To hold a place in the argument list for an argument that has not yet been implemented by DIGITAL but might be in the future. **2** To mark the position of an argument that was used in earlier versions of the routine but is not used in the latest version. (This ensures upward compatibility, because arguments that follow the null argument in the argument list keep their original positions.) A null argument is always given the name **nullarg**. In the argument list constructed on the stack, when a procedure is called, both null arguments and omitted optional arguments are represented by longword argument list entries containing the value *0*. The programming language syntax required to produce argument list entries containing *0* differs from language to language, so see the appropriate language user guide for language-specific syntax.

18.1.2 Returns Heading

If any information is returned by the routine to the caller, the description of that information appears under the returns heading. Programs written in VAX MACRO return information in R0. Returned information is a longword value.

High-level language programmers receive status information in the return (or status) variable they use when they make the call. The run-time environment established for a high-level language program allows the status information in R0 to be moved automatically to the user return variable. Returned information is always a longword value.

18.1.3 Arguments Heading

Detailed information about each argument in the call format appears under the arguments heading. Arguments are described in the order they appear in the call format.

The following format is used to describe each argument:

argument-name

VMS Usage:	argument-VMS-data-type
type:	argument-data-type
access:	argument-access
mechanism:	argument-passing-mechanism

One paragraph of structured text, followed by other paragraphs of text as needed.

18.2 Functional Organization of UIS Routines

UIS routines perform many functions within an application program. In addition to those that create the graphic objects you see on the display screen, there are routines that manage input devices and return information to the program, to name a few.

Figure 18–1 lists each UIS routine by functional category.

Figure 18-1 Functional Categories of UIS Routines

AST-Enabling Routines

UIS$SET_ADDOPT_AST
UIS$SET_BUTTON_AST
UIS$SET_CLOSE_AST
UIS$SET_EXPAND_ICON_AST
UIS$SET_GAIN_KB_AST
UIS$SET_KB_AST
UIS$SET_LOSE_KB_AST
UIS$SET_MOVE_INFO_AST
UIS$SET_POINTER_AST
UIS$SET_RESIZE_AST
UIS$SET_SHRINK_TO_ICON_AST
UIS$SET_TB_AST

Attribute Routines

UIS$SET_ARC_TYPE
UIS$SET_BACKGROUND_INDEX
UIS$SET_CHAR_ROTATION
UIS$SET_CHAR_SIZE
UIS$SET_CHAR_SLANT
UIS$SET_CHAR_SPACING
UIS$SET_CLIP
UIS$SET_FILL_PATTERN
UIS$SET_FONT
UIS$SET_LINE_STYLE
UIS$SET_LINE_WIDTH
UIS$SET_TEXT_FORMATTING
UIS$SET_TEXT_MARGINS
UIS$SET_TEXT_PATH
UIS$SET_TEXT_SLOPE
UIS$SET_WRITING_INDEX
UIS$SET_WRITING_MODE

Color Routines

UIS$CREATE_COLOR_MAP
UIS$CREATE_COLOR_MAP_SEG
UIS$DELETE_COLOR_MAP
UIS$DELETE_COLOR_MAP_SEG
UIS$HLS_TO_RGB
UIS$HSV_TO_RGB
UIS$RESTORE_CMS_COLORS
UIS$RGB_TO_HLS
UIS$RGB_TO_HSV
UIS$SET_COLOR
UIS$SET_COLORS
UIS$SET_INTENSITIES
UIS$SET_INTENSITY

Display List Routines

UIS$BEGIN_SEGMENT
UIS$COPY_OBJECT
UIS$DELETE_OBJECT
UIS$DELETE_PRIVATE
UIS$DISABLE_DISPLAY_LIST
UIS$ENABLE_DISPLAY_LIST
UIS$END_SEGMENT
UIS$ERASE
UIS$EXECUTE
UIS$EXECUTE_DISPLAY
UIS$EXTRACT_HEADER
UIS$EXTRACT_OBJECT
UIS$EXTRACT_PRIVATE
UIS$EXTRACT_REGION
UIS$EXTRACT_TRAILER
UIS$FIND_PRIMITIVE
UIS$FIND_SEGMENT
UIS$INSERT_OBJECT
UIS$MOVE_AREA
UIS$PRIVATE
UIS$SET_INSERTION_POSITION
UIS$TRANSFORM_OBJECT

Graphics Routines

UIS$CIRCLE
UIS$ELLIPSE
UIS$IMAGE
UIS$LINE
UIS$LINE_ARRAY
UIS$PLOT
UIS$PLOT_ARRAY

Inquiry Routines

UIS$GET_ABS_POINTER_POS
UIS$GET_ALIGNED_POSITION
UIS$GET_ARC_TYPE
UIS$GET_BACKGROUND_INDEX
UIS$GET_BUTTONS
UIS$GET_CHAR_ROTATION
UIS$GET_CHAR_SIZE
UIS$GET_CHAR_SLANT
UIS$GET_CHAR_SPACING
UIS$GET_CLIP
UIS$GET_COLOR
UIS$GET_COLORS
UIS$GET_CURRENT_OBJECT

ZK-4674-85

Figure 18-1 Cont'd. on next page

Figure 18-1 (Cont.) Functional Categories of UIS Routines

Inquiry Routines (cont.)

UIS$GET_DISPLAY_SIZE
UIS$GET_FILL_PATTERN
UIS$GET_FONT
UIS$GET_FONT_ATTRIBUTES
UIS$GET_FONT_SIZE
UIS$GET_HW_COLOR_INFO
UIS$GET_INTENSITIES
UIS$GET_INTENSITY
UIS$GET_KB_ATTRIBUTES
UIS$GET_LINE_STYLE
UIS$GET_LINE_WIDTH
UIS$GET_NEXT_OBJECT
UIS$GET_OBJECT_ATTRIBUTES
UIS$GET_PARENT_SEGMENT
UIS$GET_POINTER_POSITION
UIS$GET_POSITION
UIS$GET_PREVIOUS_OBJECT
UIS$GET_ROOT_SEGMENT
UIS$GET_TB_INFO
UIS$GET_TB_POSITION
UIS$GET_TEXT_FORMATTING
UIS$GET_TEXT_MARGINS
UIS$GET_TEXT_PATH
UIS$GET_TEXT_SLOPE
UIS$GET_VCM_ID
UIS$GET_VIEWPORT_ICON
UIS$GET_VIEWPORT_POSITION
UIS$GET_VIEWPORT_SIZE
UIS$GET_VISIBILITY
UIS$GET_WINDOW_ATTRIBUTES
UIS$GET_WINDOW_SIZE
UIS$GET_WRITING_INDEX
UIS$GET_WRITING_MODE
UIS$GET_WS_COLOR
UIS$GET_WS_INTENSITY

Keyboard Routines

UIS$CREATE_KB
UIS$DELETE_KB
UIS$DISABLE_KB
UIS$DISABLE_VIEWPORT_KB
UIS$ENABLE_KB
UIS$ENABLE_VIEWPORT_KB

Keyboard Routines (cont.)

UIS$READ_CHAR
UIS$SET_KB_ATTRIBUTES
UIS$SET_KB_COMPOSE2
UIS$SET_KB_COMPOSE3
UIS$SET_KB_KEYTABLE
UIS$TEST_KB

Pointer Routines

UIS$CREATE_TB
UIS$DELETE_TB
UIS$DISABLE_TB
UIS$ENABLE_TB
UIS$SET_POINTER_PATTERN
UIS$SET_POINTER_POSITION

Sound Routines

UIS$SOUND_BELL
UIS$SOUND_CLICK

Text Routines

UIS$MEASURE_TEXT
UIS$NEW_TEXT_LINE
UIS$SET_ALIGNED_POSITION
UIS$SET_POSITION
UIS$TEXT

Windowing Routines

UIS$CLOSE_WINDOW
UIS$CREATE_DISPLAY
UIS$CREATE_TERMINAL
UIS$CREATE_TRANSFORMATION
UIS$CREATE_WINDOW
UIS$DELETE_DISPLAY
UIS$DELETE_TRANSFORMATION
UIS$DELETE_WINDOW
UIS$EXPAND_ICON
UIS$MOVE_VIEWPORT
UIS$MOVE_WINDOW
UIS$POP_VIEWPORT
UIS$PUSH_VIEWPORT
UIS$RESIZE_WINDOW
UIS$SHRINK_TO_ICON

ZK-4674/1-85

18.3 Routine Arguments Quick Reference

This section is intended as a quick reference to eliminate repetition of common argument descriptions used by many different UIS and UISDC routines. (The arguments described separately here are referenced in the routine descriptions.)

Descriptions of frequently-occurring arguments follow.

18.3.1 vd_id

Argument

vd_id

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Virtual display identifier. The **vd_id** argument is the address of a longword that uniquely identifies a virtual display. The longword value is returned as the virtual display identifier in the variable *vd_id* or R0 (VAX MACRO). The virtual display identifier uniquely identifies the virtual display and is used as a parameter in all output and attribute routines.

18.3.2 wd_id

Argument

wd_id

VMS Usage:	**object_id**
type:	**longword**
access:	**read only**
mechanism:	**by reference**

Display window identifier. The **wd_id** argument is the address of a longword that uniquely identifies a display window. If this argument is specified, it must be a valid **wd_id** associated with the virtual display. The colors returned are the realized colors for the specific device for which the window was created. The longword value is returned as the display window identifier in the variable *wd_id* or R0 (VAX MACRO).

If **wd_id** is not specified, the *set* color values, that is, the actual color values in the specified color map entry, are returned.

18.3.3 obj_id

obj_id

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Object identifier. The **obj_id** argument is the address of a longword that uniquely identifies an object.

18.3.4 seg_id

seg_id

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Segment identifier. The **seg_id** argument is the address of a longword that uniquely identifies the segment. In UIS$BEGIN_SEGMENT, the longword value is returned as the segment identifier in the variable *set_id* or R0 (VAX MACRO).

18.3.5 iatb

iatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Input attribute block number. The **iatb** argument is the address of a longword integer that identifies an attribute block to be modified.

18.3.6 oatb

oatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Output attribute block number. The **oatb** argument is the address of a longword that identifies a newly modified attribute block.

18.3.7 astprm

astprm

VMS Usage:	**user_arg**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

AST parameter. The **astprm** argument is the address of a single argument or data structure such as an array or record to be used by the AST routine. In VAX FORTRAN application programs, code calls to AST routines as follows: %REF(%LOC(astprm)).

18.3.8 kb_id

kb_id

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Virtual keyboard identifier. The **kb_id** argument is the address of a longword that uniquely identifies a virtual keyboard. This represents the longword value returned as the virtual keyboard identifier in the variable *kb_id* or R0 (VAX MACRO).

18.3.9 devnam

devnam

VMS Usage:	**device_name**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

Device name string. The **devnam** argument is the character string descriptor address of the workstation device name. Specify the device name SYS$WORKSTATION in the **devnam** argument.

18.4 UIS Routines and Arguments

Table 18–3 lists each argument and the routines it uses.

Table 18-3 Routine Arguments

Routine	Arguments
UIS$BEGIN_SEGMENT	vd_id
UIS$CIRCLE	vd_id, atb, center_x, center_y, xradius, start_deg, end_deg
UIS$CLOSE_WINDOW	wd_id
UIS$COPY_OBJECT	obj_id, seg_id, matrix, atb
UIS$CREATE_COLOR_MAP	vcm_size, vcm_name, vcm_attributes
UIS$CREATE_COLOR_MAP_SEG	vcm_id, devnam, place_mode, place_data
UIS$CREATE_DISPLAY	x_1, y_1, x_2, y_2, width, height, vcm_id
UIS$CREATE_KB	devnam
UIS$CREATE_TB	devnam
UIS$CREATE_TERMINAL	termtype, title, attributes, devnam, devlen
UIS$CREATE_TRANSFORMATION	vd_id, x_1, y_1, x_2, y_2, vdx_1, vdy_1, vdx_2, vdy_2
UIS$CREATE_WINDOW	vd_id, devnam, title, x_1, y_1, x_2, y_2, width, height, attributes
UIS$DELETE_COLOR_MAP	vcm_id
UIS$DELETE_COLOR_MAP_SEG	cms_id
UIS$DELETE_DISPLAY	vd_id
UIS$DELETE_KB	kb_id
UIS$DELETE_OBJECT	obj_id, seg_id
UIS$DELETE_PRIVATE	obj_id, seg_id
UIS$DELETE_TB	tb_id
UIS$DELETE_TRANSFORMATION	tr_id
UIS$DELETE_WINDOW	wd_id
UIS$DISABLE_DISPLAY_LIST	vd_id, display_flags
UIS$DISABLE_KB	kb_id
UIS$DISABLE_TB	tb_id
UIS$DISABLE_VIEWPORT_KB	wd_id
UIS$ELLIPSE	vd_id, atb, center_x, center_y, xradius, yradius, start_deg, end_deg
UIS$ENABLE_DISPLAY_LIST	vd_id, display_flags
UIS$ENABLE_KB	kb_id, wd_id
UIS$ENABLE_TB	tb_id
UIS$ENABLE_VIEWPORT_KB	kb_id, wd_id
UIS$END_SEGMENT	vd_id
UIS$ERASE	vd_id, x_1, y_1, x_2, y_2
UIS$EXECUTE	vd_id, buflen, bufaddr

Table 18-3 (Cont.) Routine Arguments

Routine	Arguments
UIS$EXECUTE_DISPLAY	buflen, bufaddr
UIS$EXPAND_ICON	wd_id, icon_wd_id, attributes
UIS$EXTRACT_HEADER	vd_id, buflen, bufaddr, retlen
UIS$EXTRACT_OBJECT	obj_id, seg_id, buflen, bufaddr, retlen
UIS$EXTRACT_PRIVATE	obj_id, seg_id, buflen, bufaddr, retlen
UIS$EXTRACT_REGION	vd_id, x_1, y_1, x_2, y_2, buflen, bufaddr, retlen
UIS$EXTRACT_TRAILER	vd_id, buflen, bufaddr, retlen
UIS$FIND_PRIMITIVE	vd_id, x_1, y_1, x_2, y_2, context, extent
UIS$FIND_SEGMENT	vd_id, x_1, y_1, x_2, y_2, context, extent
UIS$GET_ABS_POINTER_POS	devnam, retx, rety
UIS$GET_ALIGNED_POSITION	vd_id, atb, retx, rety
UIS$GET_ARC_TYPE	vd_id, atb
UIS$GET_BACKGROUND_ INDEX	vd_id, atb
UIS$GET_BUTTONS	wd_id, restate
UIS$GET_CHAR_ROTATION	vd_id, atb
UIS$GET_CHAR_SIZE	vd_id, atb, char, width, height
UIS$GET_CHAR_SLANT	vd_id, atb
UIS$GET_CHAR_SPACING	vd_id, atb, dx, dy
UIS$GET_CLIP	vd_id, atb, x_1, y_1, x_2, y_2
UIS$GET_COLOR	vd_id, index, retr, retg, retb, wd_id
UIS$GET_COLORS	vd_id, index, count, retr_vector, retg_vector, retb_vector, wd_id
UIS$GET_CURRENT_OBJECT	vd_id
UIS$GET_DISPLAY_SIZE	devnam, retwidth, retheight, retresolx, retresoly, retpwidth, retpheight
UIS$GET_FILL_PATTERN	vd_id, atb, index
UIS$GET_FONT	vd_id, atb, bufferdesc, length
UIS$GET_FONT_ATTRIBUTES	font_id, ascender, descender, height, maximum_ width, item_list
UIS$GET_FONT_SIZE	fontid, text_string, retwidth, retheight
UIS$GET_HW_COLOR_INFO	devnam, type, indices, colors, maps, rbits, gbits, bbits, ibits, res_indices, regen
UIS$GET_INTENSITIES	vd_id, index, count, reti_vector, wd_id
UIS$GET_INTENSITY	vd_id, index, reti, wd_id
UIS$GET_KB_ATTRIBUTES	kb_id, enable_items, disable_items, click_volume
UIS$GET_LINE_STYLE	vd_id, atb
UIS$GET_LINE_WIDTH	vd_id, atb, mode
UIS$GET_NEXT_OBJECT	obj_id, seg_id, flags

Table 18-3 (Cont.) Routine Arguments

Routine	Arguments
UIS$GET_OBJECT_ ATTRIBUTES	obj_id, seg_id, extent
UIS$GET_PARENT_SEGMENT	obj_id, seg_id
UIS$GET_POINTER_POSITION	vd_id, wd_id, retx, rety
UIS$GET_POSITION	vd_id, retx, rety
UIS$GET_PREVIOUS_OBJECT	obj_id, seg_id, flags
UIS$GET_ROOT_SEGMENT	vd_id
UIS$GET_TB_INFO	devnam, retwidth, retheight, retresolx, retresoly, retpwidth, retpheight
UIS$GET_TB_POSITION	tb_id, retx, rety
UIS$GET_TEXT_FORMATTING	vd_id, atb
UIS$GET_TEXT_MARGINS	vd_id, atb, x, y, margin_length
UIS$GET_TEXT_PATH	vd_id, atb, major, minor
UIS$GET_TEXT_SLOPE	vd_id, atb
UIS$GET_VCM_ID	vd_id
UIS$GET_VIEWPORT_ICON	wd_id, icon_wd_id
UIS$GET_VIEWPORT_ POSITION	wd_id, retx, rety
UIS$GET_VIEWPORT_SIZE	wd_id, retwidth, retheight
UIS$GET_VISIBILITY	vd_id, wd_id, x_1, y_1, x_2, y_2
UIS$GET_WINDOW_ ATTRIBUTES	wd_id
UIS$GET_WINDOW_SIZE	vd_id, wd_id, x_1, y_1, x_2, y_2
UIS$GET_WRITING_INDEX	vd_id, atb
UIS$GET_WRITING_MODE	vd_id, atb
UIS$GET_WS_COLOR	vd_id, color_id, retr, retg, retb, wd_id
UIS$GET_WS_INTENSITY	vd_id, color_id, reti, wd_id
UIS$HLS_TO_RGB	H, L, S, retr, retg, retb
UIS$HSV_TO_RGB	H, S, V, retr, retg, retb
UIS$IMAGE	vd_id, atb, x_1, y_1, x_2, y_2, rasterwidth, rasterheight, bitsperpixel, rasteraddr
UIS$INSERT_OBJECT	obj_id, seg_id
UIS$LINE	vd_id, atb, x, y
UIS$LINE_ARRAY	vd_id, atb, count, x_vector, y_vector
UIS$MEASURE_TEXT	vd_id, atb, text_string, retwidth, retheight, ctllist, ctllen, posarray
UIS$MOVE_AREA	vd_id, x_1, y_1, x_2, y_2, new_x, new_y
UIS$MOVE_VIEWPORT	wd_id, attributes
UIS$MOVE_WINDOW	vd_id, wd_id, x_1, y_1, x_2, y_2
UIS$NEW_TEXT_LINE	vd_id, atb

Table 18-3 (Cont.) Routine Arguments

Routine	Arguments
UIS$PLOT	vd_id, atb, x, y
UIS$PLOT_ARRAY	vd_id, atb, count, x_vector, y_vector
UIS$POP_VIEWPORT	wd_id
UIS$PRESENT	major_version, minor_version
UIS$PRIVATE	obj_id, vd_id, facnum, buffer
UIS$PUSH_VIEWPORT	wd_id
UIS$READ_CHAR	kb_id, flags
UIS$RESIZE_WINDOW	vd_id, wd_id, new_abs_x, new_abs_y, new_width, new_height, new_wc_x_1, new_wc_y_1, new_wc_x_2, new_wc_y_2
UIS$RESTORE_CMS_COLORS	cms_id
UIS$RGB_TO_HLS	R, G, B, reth, retl, rets
UIS$RGB_TO_HSV	R, G, B, reth, rets, retv
UIS$SET_ADDOPT_AST	wd_id, astadr, astprm
UIS$SET_ALIGNED_POSITION	vd_id, atb, x, y
UIS$SET_ARC_TYPE	vd_id, iatb, oatb, arc_type
UIS$SET_BACKGROUND_ INDEX	vd_id, iatb, oatb, index
UIS$SET_BUTTON_AST	vd_id, wd_id, astadr, astprm, keybuf, x_1, y_1, x_2, y_2
UIS$SET_CHAR_ROTATION	vd_id, iatb, oatb, angle
UIS$SET_CHAR_SIZE	vd_id, iatb, oatb, char, width, height
UIS$SET_CHAR_SLANT	vd_id, iatb, oatb, angle
UIS$SET_CHAR_SPACING	vd_id, iatb, oatb, dx, dy
UIS$SET_CLIP	vd_id, iatb, oatb, x_1, y_1, x_2, y_2
UIS$SET_CLOSE_AST	wd_id, astadr, astprm
UIS$SET_COLOR	vd_id, index, R, G, B
UIS$SET_COLORS	vd_id, index, count, r_vector, g_vector, b_vector
UIS$SET_EXPAND_ICON_AST	wd_id, astadr, astprm
UIS$SET_FILL_PATTERN	vd_id, iatb, oatb, index
UIS$SET_FONT	vd_id, iatb, oatb, font_id
UIS$SET_GAIN_KB_AST	kb_id, astadr, astprm
UIS$SET_INSERTION_ POSITION	obj_id, seg_id, vd_id, flags
UIS$SET_INTENSITIES	vd_id, index, count, i_vector
UIS$SET_INTENSITY	vd_id, index, I
UIS$SET_KB_AST	kb_id, astadr, astprm, keybuf
UIS$SET_KB_ATTRIBUTES	kb_id, enable_items, disable_items, click_volume
UIS$SET_KB_COMPOSE2	kb_id, table, tablelen

Table 18-3 (Cont.) Routine Arguments

Routine	Arguments
UIS$SET_KB_COMPOSE3	kb_id, table, tablelen
UIS$SET_KB_KEYTABLE	kb_id, table, tablelen
UIS$SET_LINE_STYLE	vd_id, iatb, oatb, style
UIS$SET_LINE_WIDTH	vd_id, iatb, oatb, width, mode
UIS$SET_LOSE_KB_AST	kb_id, astadr, astprm
UIS$SET_MOVE_INFO_AST	wd_id, astadr, astprm
UIS$SET_POINTER_AST	vd_id, wd_id, astadr, astprm, x_1, y_1, x_2, y_2, exitastadr, exitastprm
UIS$SET_POINTER_PATTERN	vd_id, wd_id, pattern_array, pattern_count, activex, activey, x_1, y_1, x_2, y_2, flags
UIS$SET_POINTER_POSITION	vd_id, wd_id, x, y
UIS$SET_POSITION	vd_id, x, y
UIS$SET_RESIZE_AST	vd_id, wd_id, astadr, astprm, new_abs_x, new_abs_y, new_width, new_height, new_wc_x_1, new_wc_y_1, new_wc_x_2, new_wc_y_2
UIS$SET_SHRINK_TO_ICON_AST	wd_id, astadr, astprm
UIS$SET_TB_AST	tb_id, data_astadr, data_astprm, x_pos, y_pos, data_x_1, data_y_1, data_x_2, data_y_2, button_astadr, button_astprm, button_keybuf
UIS$SET_TEXT_FORMATTING	vd_id, iatb, oatb, mode
UIS$SET_TEXT_MARGINS	vd_id, iatb, oatb, x, y, margin_length
UIS$SET_TEXT_PATH	vd_id, iatb, oatb, major, minor
UIS$SET_TEXT_SLOPE	vd_id, iatb, oatb, angle
UIS$SET_WRITING_INDEX	vd_id, iatb, oatb, index
UIS$SET_WRITING_MODE	vd_id, iatb, oatb, mode
UIS$SHRINK_TO_ICON	wd_id, icon_wd_id, icon_flags, icon_name, attributes
UIS$SOUND_BELL	devnam, bell_volume
UIS$SOUND_CLICK	devnam, click_volume
UIS$TEST_KB	kb_id
UIS$TEXT	vd_id, atb, text_string, x, y, ctllist, ctllen
UIS$TRANSFORM_OBJECT	obj_id, seg_id, matrix, atb

UIS$BEGIN_SEGMENT

Begins a new segment in the virtual display.

FORMAT

seg_id = **UIS$BEGIN_SEGMENT** *vd_id*

RETURNS

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the segment identifier in the variable *seg_id* or R0 (VAX MACRO). The segment identifier uniquely identifies a segment and is used as an argument in other routines.

UIS$BEGIN_SEGMENT signals all errors; no condition values are returned.

ARGUMENT

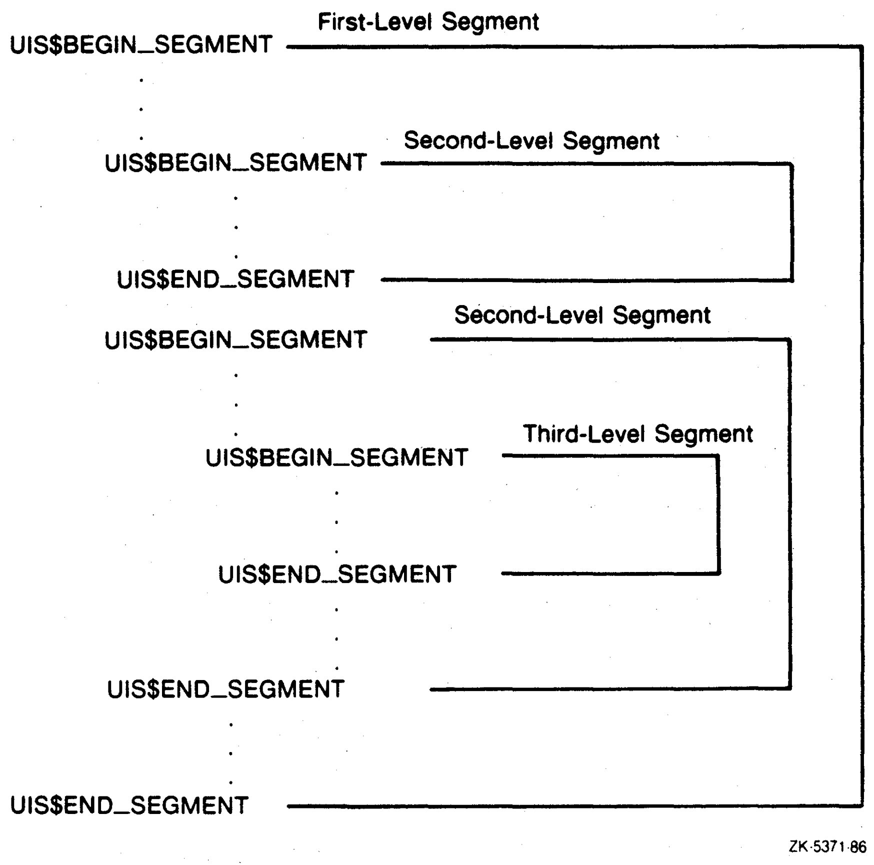
vd_id
See Section 18.3.1 for a description of this argument.

ROUTINE DESCRIPTION

All values of attribute blocks 0 to 255 are propagated to the new segment, but all changes to attribute blocks in this segment are local to this segment only and not to the parent.

You can also nest segments.

illustration



UIS$CIRCLE

Draws an arc along the circumference of a circle.

FORMAT **UIS$CIRCLE** *vd_id, atb, center_x, center_y, xradius [,start_deg ,end_deg]*

RETURNS UIS$CIRCLE signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword integer that specifies an attribute block that controls the appearance of the circle or arc.

center_x
center_y

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Center position x and y world coordinates. The **center_x** and **center_y** arguments are the addresses of f_floating point numbers that define a point in the virtual display that is the center of the arc or circle.

xradius

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Radius of the circle specified as an x world coordinate width. The **xradius** argument is the address of an f_floating point number that defines the distance from the center of the circle to the circumference of the circle.

start_deg
end_deg

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Degrees at which the arc starts and ends. The **start_deg** and **end_deg** arguments are the addresses of f_floating point numbers that define the starting and ending points on the circumference of the circle where the arc or circle will be drawn. Degrees are measured clockwise from the top of the circle. If these arguments are not specified, *0.0* degrees and *360.0* degrees are assumed, respectively.

DESCRIPTION

UIS$CIRCLE draws an arc specified by a center position and a radius for the range of the degrees specified.

The arc can be closed by drawing one or more lines between the endpoints. The arc type associated with the attribute block specifies the way in which the arc is closed. The arc is not closed off by default. See UIS$SET_ARC_TYPE for details.

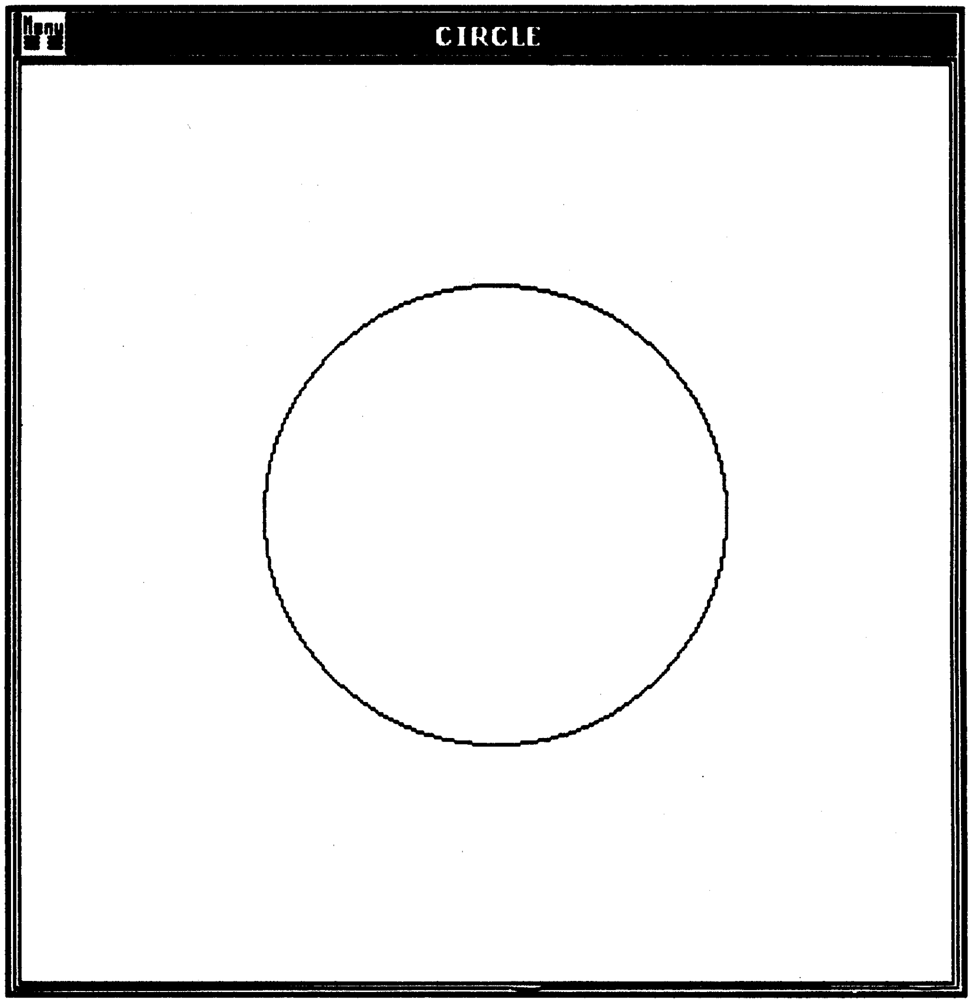
The points are drawn with the current line pattern and width, and filled with the current fill pattern if enabled.

UIS$CIRCLE does not support the following combination of attributes:

- Line width not equal to *1* and line style not equal to $FFFFFFFF_{16}$
- Line width not equal to *1* and complement writing mode

Circles are distorted by differences between the aspect ratios of the display window and display viewport.

screen output



ZK-5390-86

UIS$CLOSE_WINDOW

Calls the system service SYS$EXIT to exit the current image.

FORMAT

UIS$CLOSE_WINDOW *wd_id*

RETURNS

UIS$CLOSE_WINDOW signals all errors; no condition values are returned.

ARGUMENT

wd_id
See Section 18.3.2 for a description of this argument.

DESCRIPTION

UIS$CLOSE_WINDOW is invoked as the default action taken by the Delete menu item in the Window Options Menu. See UIS$SET_CLOSE_AST for information about overriding this routine.

UIS$COPY_OBJECT

Copies the specified object and its private data within the virtual display. Also transforms the coordinates or attributes or both of the specified object. The original object remains unchanged in the virtual display.

FORMAT

copy_id = **UIS$COPY_OBJECT** $\left\{ \begin{array}{l} \textit{obj\_id} \\ \textit{seg\_id} \end{array} \right\}$ *[,matrix]* *[,atb]*

RETURNS

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the copy identifier in the variable *copy_id* or R0 (VAX MACRO). The copy identifier uniquely identifies a newly copied object.

UIS$COPY_OBJECT signals all errors; no condition values are returned.

ARGUMENTS

obj_id
See Section 18.3.3 for a description of this argument.

seg_id
See Section 18.3.4 for a description of this argument.

matrix

VMS Usage:	**vector_longword_signed**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Transformation matrix. The **matrix** argument is the address of a 2 x 3 matrix of longwords containing scaling, translation, and/or rotation data.

Structure of a VAX FORTRAN Two-Dimensional Array

A two-dimensional array declared as ARRAY(2,3) has the following structure.

1,1	1,2	1,3
2,1	2,2	2,3

ZK-5492-86

Different languages allocate memory for array elements in different orders. This description assumes the order used by VAX FORTRAN. If you call UIS$COPY_OBJECT from another language, make sure that the array elements are in the same order.

Memory addresses of array elements range from lowest to highest in the following order: (1,1),(2,1), (1,2),(2,2),(1,3), and (2,3). The following figure shows the order of array elements.

1	3	5
2	4	6

ZK-5493-86

Pairs of array elements govern how displayed objects are scaled, rotated, and translated. UIS computes the transformed coordinates in the following manner.

$$x_1 = A(1,1)*x + A(1,2)*y + A(1,3)$$
$$y_1 = A(2,1)*x + A(2,2)*y + A(2,3)$$

Translation

When translation alone is performed, the following array elements are assigned values. D_x and D_y represent distances between the original coordinates and the new coordinates.

1	0	Dx
0	1	Dy

ZK-5494-86

Scaling

When scaling alone is performed, the following array elements are assigned values.

Sx	0	0
0	Sy	0

ZK-5495-86

Rotation

When rotation alone is performed, the following array elements are assigned values, where "@" is the desired angle of rotation measured clockwise. The values returned from the VAX FORTRAN SIN and COS functions are stored in the appropriate array elements.

cos (@)	sin (@)	0
-sin (@)	cos (@)	0

ZK-5496-86

An unlimited number of transformations can be performed at one time by simply multiplying the matrices together into a single matrix using matrix multiplication.

In order to multiply two matrices together, you must add a row to the bottom of each matrix.

0	0	1

ZK-5461-86

After the multiplication is performed, remove the last row of the result.

atb

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

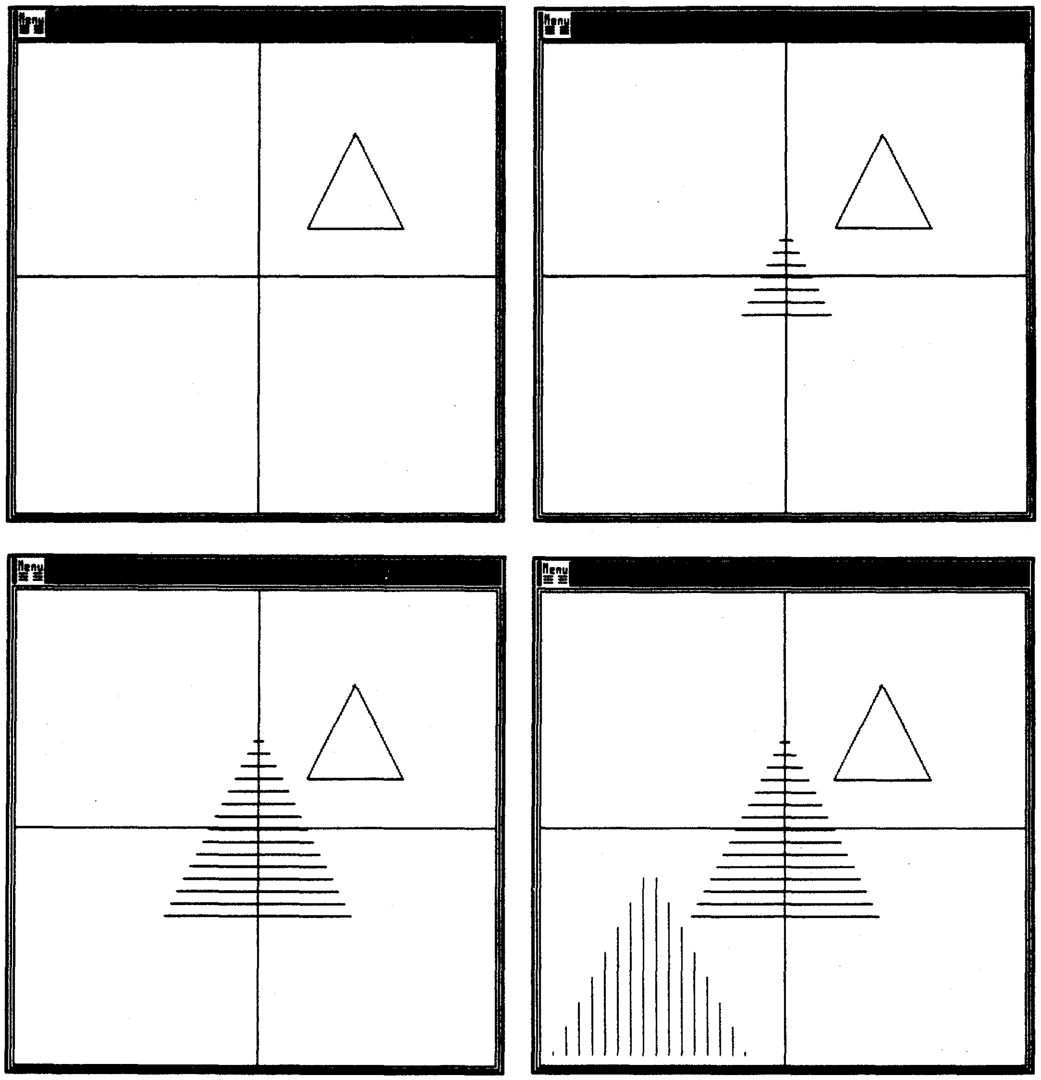
Attribute block number. The **atb** argument is the address of a longword that identifies an attribute block whose attribute settings override current segment attributes.

DESCRIPTION

Either the coordinates can be transformed, or the attributes can be overridden or both.

After a transformation, occluded objects might not appear correctly on the display screen. To correct this, call UIS$EXECUTE to refresh the display screen correctly.

screen output



ZK 5420-86

UIS$CREATE_COLOR_MAP

Creates a virtual color map of the specified size and with the specified attributes.

FORMAT

vcm_id = **UIS$CREATE_COLOR_MAP** *vcm_size [,vcm_name] [,vcm_attributes]*

RETURNS

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the virtual color map identifier in the variable *vcm_id* or R0 (VAX MACRO). The virtual color map identifier uniquely identifies the virtual color map and must be specified in UIS$CREATE_DISPLAY. It is also used as an argument in other color routines.

UIS$CREATE_COLOR_MAP signals all errors; no condition values are returned.

ARGUMENTS

vcm_size

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Size of the virtual color map. The **vcm_size** argument is the address of a longword that defines the number of entries in the virtual color map.

vcm_name

VMS Usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

Name of the virtual color map. The **vcm_name** argument is the address of a string descriptor of the name of the virtual color map. Specify the name of an existing shareable color map. If your application is creating the shareable color map, specify a valid color map name.

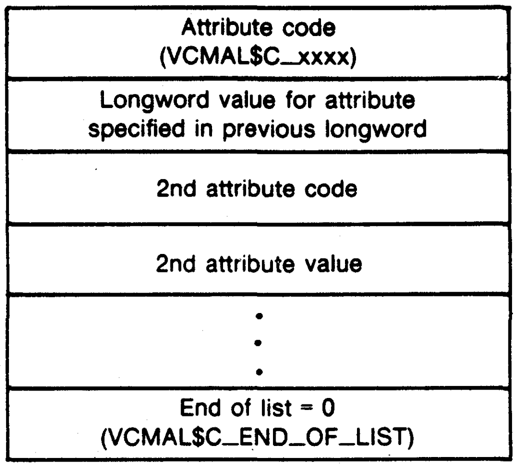
The virtual color map name should not exceed 15 characters.

vcm_attributes

VMS Usage:	**item_list_pair**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by descriptor**

Virtual color map attributes. The **vcm_attributes** argument is the address of data structure of longword pairs that specify virtual color attributes.

The following figure describes the structure of this argument.



ZK-5367-86

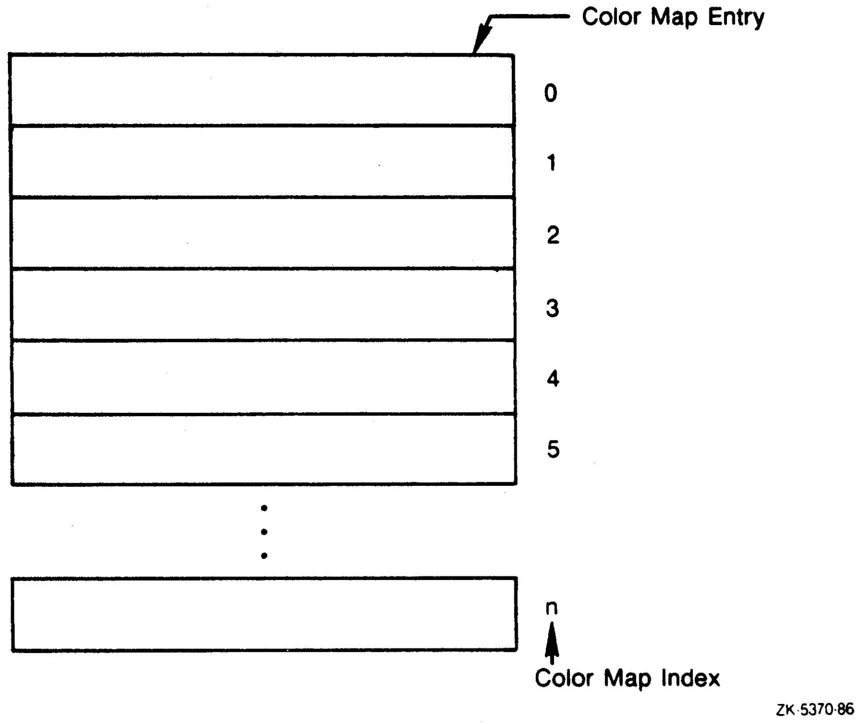
All of the following virtual color map attributes are optional.

Attributes	Function
VCMAL$C_ATTRIBUTES	General attributes
VCMAL$M_RESIDENT	Set for resident virtual color map
VCMAL$M_SHARE	Set for shareable virtual color map
VCMAL$M_SYSTEM[1,2]	Set for system shareable virtual color map
VCMAL$M_NO_BIND	Set to disable automatic hardware color map binding

[1]VCMAL$M_SHARE must also be set.

[2]SYSGBL privilege is required.

illustration



UIS$CREATE_COLOR_MAP_SEG

Allocates one or more hardware color map indices and binds them to a virtual color map.

FORMAT

cms_id = **UIS$CREATE_COLOR_MAP_SEG** *vcm_id*
[,devnam]
[,place_m
[,place_da

RETURNS

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the color map segment identifier in the variable *cms_id* or R0 (VAX MACRO). The color map segment identifier uniquely identifies the color map segment and is used as an argument in other routines.

UIS$CREATE_COLOR_MAP_SEG signals all errors; no condition values are returned.

ARGUMENTS

vcm_id

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Virtual color map identifier. The **vcm_id** argument is the address of a longword that uniquely identifies the virtual color map. See UIS$CREATE_COLOR_MAP for more information about the **vcm_id** argument.

NOTE: **This routine can be used only once for each virtual color map identifier.**

devnam

See Section 18.3.9 for more information about this argument.

place_mode

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Placement mode. The **place_mode** argument is the address of a longword that specifies the placement mode; that is, which hardware color map entries can be allocated. The following table lists valid placement modes.

Symbol	Function
UIS$C_GENERAL	General placement—Allocates any available entries in the hardware color map.
UIS$C_COLOR_EXACT	Exact placement—Allocates map entries starting at the specified entry and aligned on a natural entry boundary. Given the size of the virtual color map, UIS computes a working size that is the smallest power of 2 greater than or equal to the requested size. The natural alignment of a map is a starting index that is a multiple of the working size. For example, a six-entry color map could be placed at indices 0, 8, 16, and so on.
UIS$C_COLOR_BASED	Based placement (default)—Allocates entries such that writing modes using Boolean logic operations on pixel values can correctly display color intersections.

place_data

VMS Usage: **longword_signed**
type: **longword (signed)**
access: **read only**
mechanism: **by reference**

Placement data. The **place_data** argument is the address of a longword that contains the first index to be allocated. The **place_data** argument is used with exact placement mode.

DESCRIPTION

For hardware that supports bit plane write masks, the segment is based at an index that is a power of 2; that write operation is performed with the appropriate mask. The virtual color map entry index specified in the **place_data** argument indicates the binding between the virtual color map and the hardware color map entries allocated by UIS$CREATE_COLOR_MAP_SEG. The default value is 0; that is, the first allocated map entry is bound to virtual color map entry 0, the second allocated map entry is bound to virtual color map entry 1, and so on.

If the appropriate entries cannot be allocated, an error is signaled. In addition to resource depletion failure, a call to UIS$CREATE_COLOR_MAP_SEG can fail because UIS has already issued this call for the application. This occurs if internal processing requires binding to hardware resources and the flag VCMAL$M_NO_BIND is not set when the virtual color map is created. For example, UIS$CREATE_WINDOW allocates and binds hardware color map resources when it creates a display viewport.

Conversely, if VCMAL$M_NO_BIND is set but UIS$CREATE_COLOR_MAP_SEG was not called, calls to some UIS routines such as UIS$SET_COLOR and UIS$SET_INTENSITY might fail.

NOTE: **Use this routine as follows:**

1 **When you create the virtual color map with UIS$CREATE_COLOR_MAP, specify the flag VCMAL$M_NO_BIND.**

2 **Before you call any other UIS routine, invoke UIS$CREATE_COLOR_MAP_SEG.**

3 Initialize the color map with UIS$SET_COLORS. (By definition all colors are black.)

UIS$CREATE_DISPLAY

Creates a virtual display.

FORMAT

vd_id = **UIS$CREATE_DISPLAY** *$x_1$, y_1, x_2, y_2, width, height [,vcm_id]*

RETURNS

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the virtual display identifier in the variable *vd_id* or R0 (VAX MACRO). The virtual display identifier uniquely identifies the virtual display and is used as a parameter in all output and attribute routines.

UIS$CREATE_DISPLAY signals all errors; no condition values are returned.

ARGUMENTS

x_1, y_1
x_2, y_2

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

World coordinates of the virtual display space. The x_1 and y_1 arguments are the addresses of f_floating point numbers that define the lower-left corner of the virtual display space. The x_2 and y_2 arguments are the addresses of f_floating point numbers that define the upper-right corner of the virtual display.

These arguments define mapping and scaling factors and are not the boundaries of the virtual display.

width
height

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Width and height of the display viewport. The **width** and **height** arguments are the addresses of f_floating point numbers that define both the width and height of the display viewport in centimeters.

vcm_id

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Virtual color map identifier. The **vcm_id** argument is the address of a longword that uniquely identifies the virtual color map. See UIS$CREATE_COLOR_MAP for more information about the **vcm_id** argument.

If **vcm_id** is not specified, a two-entry virtual color map is created for the virtual display by default.

DESCRIPTION

To avoid distortion of the resulting graphic image, the aspect ratio of the world coordinate range of the display window must be equal to the aspect ratio of the display viewport. See UIS$CREATE_WINDOW for more information about aspect ratios.

UIS$CREATE_KB

Creates a virtual keyboard on the specified device.

FORMAT

kb_id = **UIS$CREATE_KB** *devnam*

RETURNS

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the virtual keyboard identifier in the variable *kb_id* or R0 (VAX MACRO). The virtual keyboard identifier uniquely identifies the virtual keyboard. The variable *kb_id* is used as an argument in other routines.

UIS$CREATE_KB signals all errors; no condition values are returned.

ARGUMENT

devnam
See Section 18.3.9 for more information about this argument.

DESCRIPTION

UIS$CREATE_KB generates a value for the **kb_id** argument that is referenced in subsequent routines that use **kb_id** as a parameter.

EXAMPLE

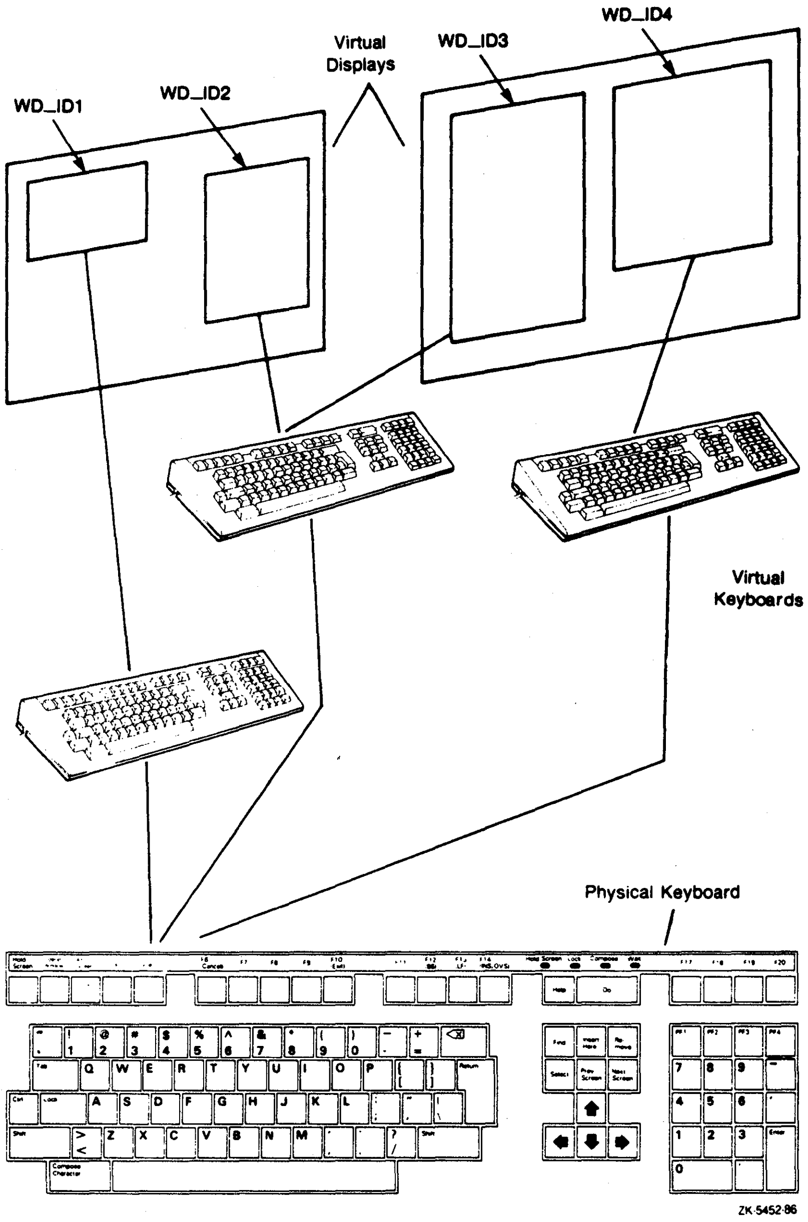
```
    .
    .
    .
VD_ID=UIS$CREATE_DISPLAY(-5.0,-5.0,50.0,45.0,15.0,15.0)
KB_ID=UIS$CREATE_KB('SYS$WORKSTATION')     1
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','VIEWPORT TITLE',
2     10.0,10.0,25.0,25.0)
CALL UIS$ENABLE_VIEWPORT_KB(KB_ID,WD_ID)    2
    .
    .
    .
CALL UIS$DISABLE_VIEWPORT_KB(WD_ID)    3
```

The preceding example creates a virtual keyboard 1 and binds the virtual keyboard to a display window 2. In order to use the virtual keyboard and its characteristics with the desired viewport, you must assign the physical keyboard to the desired virtual keyboard and viewport. Press the F5 or $\boxed{\text{CYCLE}}$ key until the KB icon in the appropriate viewport is highlighted.

The call to UIS$DISABLE_VIEWPORT_KB ❸ explicitly disables the binding between the virtual keyboard and the display window. Also, the ability to assign the physical keyboard to the appropriate virtual keyboard, that is, to *cycle* from viewport to viewport, is disabled.

If UIS$ENABLE_KB is called after UIS$ENABLE_VIEWPORT_KB, the KB icon is highlighted as soon as the program executes.

illustration



UIS$CREATE_TB

Creates a tablet digitizer identifier that allows you to connect your process to the tablet.

FORMAT

tb_id = **UIS$CREATE_TB** *devnam*

RETURNS

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by reference**

Longword value returned as the tablet identifier in the variable *tb_id* or R0 (VAX MACRO). The tablet identifier uniquely identifies the tablet device and can be used in other routines where appropriate.

UIS$CREATE_TB signals all errors; no condition values are returned.

ARGUMENT

devnam

See Section 18.3.9 for more information about this argument.

DESCRIPTION

UIS$CREATE_TB creates a tablet digitizer identifier. When you want to connect to the tablet, you must specify this identifier in a call to UIS$ENABLE_TB.

UIS$CREATE_TERMINAL

Creates a terminal emulation window of the specified type.

FORMAT

UIS$CREATE_TERMINAL *termtype [,title] [,attributes] [,devnam] [,devlen] [,term attributes]*

RETURNS

UIS$CREATE_TERMINAL signals all errors; no condition values are returned.

ARGUMENTS

termtype

VMS Usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

Terminal type. The **termtype** argument is the address of a character string descriptor of the terminal type. Specify either WT for a VT220 emulation window or TK for a TEK4010/4014 emulation window.

title

VMS Usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

Window title. The **title** argument is the address of a descriptor of a character string that is the title of the terminal emulation window.

attributes

VMS Usage:	**item_list_pair**
type:	**longword**
access:	**read only**
mechanism:	**by reference**

Window attributes list. The **attributes** argument is the address of a data structure that contains two or more longwords. The list consists of one or more longword pairs, or *doublets*. The first longword contains an attribute code, while the second longword holds an attribute value (which can be real or integer). The constant WDPL$C_END_OF_LIST terminates the list.

The window attributes list has the same format as defined in the UIS$CREATE_WINDOW service. If your application program is written in FORTRAN, use the RECORD data type to construct the attribute list. Refer to UIS$CREATE_WINDOW for a description of the attribute list.

devnam

VMS Usage:	**device_name**
type:	**character string**
access:	**write only**
mechanism:	**by descriptor**

New terminal emulation device name. The **devnam** argument is the character string descriptor address of a location that receives the new terminal emulation device name string.

devlen

VMS Usage:	**word_signed**
type:	**word (signed)**
access:	**write only**
mechanism:	**by reference**

Length of the terminal emulation device name string. The **devlen** argument is the address of a word that receives the length of the terminal device name character string.

term attributes

VMS Usage:	**item_list_pair**
type:	**longword**
access:	**read only**
mechanism:	**by reference**

Terminal attributes list. The **term attributes** argument is the address of a data structure that contains two or more longwords. The list consists of one or more longword pairs. The first longword contains an attribute code (UIS$C_TERM_COLOR); the second longword holds an integer attribute value for UIS$C_TERM_COLOR. The constant UIS$C_TERM_END_OF_LIST terminates the list. See UIS$CREATE_WINDOW.

DESCRIPTION

UIS$CREATE_TERMINAL creates a pseudodevice in the VMS database and returns the device name string for the device. The window might not appear on the screen until a channel is assigned to the device using the SYS$ASSIGN system service and the first write to the device is performed.

The pseudodevice is created without any initial owner. Once a channel is assigned to the device, it is owned by that process, which is usually the same process that issued the UIS$CREATE_TERMINAL call. After all channels have been deassigned, the pseudodevice will be removed automatically from the system. If a permanent pseudodevice is required, then the application should specify a process that maintains a permanent channel to the device.

UIS$CREATE_TRANSFORMATION

Creates a two-dimensional world coordinate transformation into an existing virtual display's coordinate space. It provides for two-dimensional translation and scaling, but not rotation.

FORMAT

tr_id = **UIS$CREATE_TRANSFORMATION** *vd_id*, x_1, y_1, x_2, y_2 [,vdx_1, vdy_1, vdx_2, vdy_2]

RETURNS

VMS Usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword value returned as the transformation identifier in the variable *tr_id* or R0 (VAX MACRO). The transformation identifier uniquely identifies a transformation coordinate space. See the Description section below for more information about *tr_id*.

UIS$CREATE_TRANSFORMATION signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

x_1, y_1
x_2, y_2
VMS Usage: **floating_point**
type: **f_floating**
access: **read only**
mechanism: **by reference**

World coordinates of the new coordinate space. The x_1 and y_1 arguments and the x_2 and y_2 arguments are the addresses of f_floating point numbers that define the lower-left corner and upper-right corner of the new transformation coordinate space, respectively.

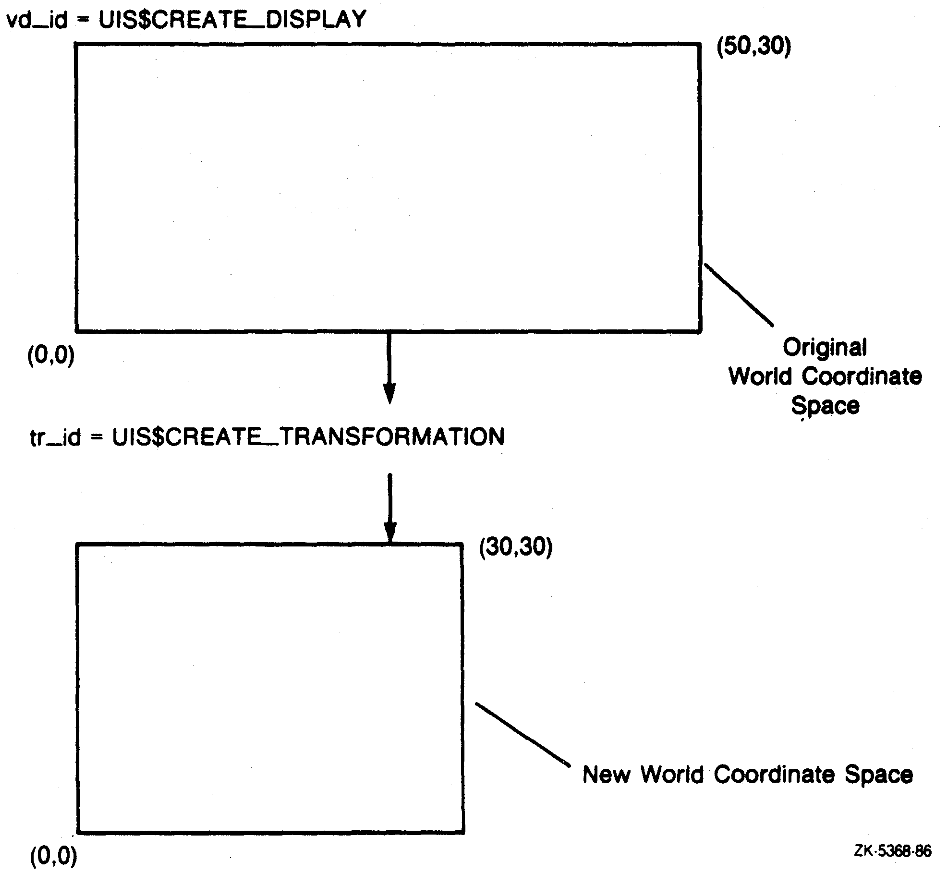
vdx_1, vdy_1
vdx_2, vdy_2
VMS Usage: **floating_point**
type: **f_floating**
access: **read only**
mechanism: **by reference**

World coordinates of the original virtual display space. The **vdx_1** and **vdy_1** arguments are the addresses of f_floating point numbers that define the lower-left corner of the corresponding virtual display space. The **vdx$_2$** and **vdy$_2$** arguments are the addresses of f_floating point numbers that define the upper-right corner of the corresponding virtual display space. If these optional arguments are not specified, the world coordinates specified in UIS$CREATE_DISPLAY are used.

DESCRIPTION

Once the transformation is created, it can be used in any routine that accepts a **vd_id** argument except UIS$DELETE_DISPLAY by substituting the **tr_id** argument instead. When the tr_id value is used, it indicates the same virtual display but that the coordinates are mapped relative to the transformation coordinate space, and not the original virtual display coordinate space. Each routine automatically performs the transformation.

illustration


vd_id = UIS$CREATE_DISPLAY
(50,30)
(0,0)
Original World Coordinate Space
tr_id = UIS$CREATE_TRANSFORMATION
(30,30)
New World Coordinate Space
(0,0)
ZK-5368-86

UIS$CREATE_WINDOW

Creates a display window and an associated display viewport. See UIS$GET_WINDOW_ATTRIBUTES for information about window attributes.

FORMAT

wd_id = **UIS$CREATE_WINDOW** *vd_id, devnam [,title] [,x_1, y_1, x_2, y_2] [,width, height] [,attributes]*

RETURNS

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the display window identifier in the variable *wd_id* or R0 (VAX MACRO). The display window identifier uniquely identifies the display window and is used as an argument in other routines.

UIS$CREATE_WINDOW signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

devnam
See Section 18.3.9 for more information about this argument.

title

VMS Usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

Banner title. The **title** argument is the address of a descriptor of the character string to be inserted into the banner of the display viewport. If the argument **title** is not specified, the display banner is created without a title.

x_1, y_1
x_2, y_2

VMS Usage: **floating_point**
type: **f_floating**
access: **read only**
mechanism: **by reference**

World coordinates of the display window. The x_1, y_1 and x_2, y_2 arguments are addresses of f_floating point numbers that define the lower-left corner and upper-right corner of the display window rectangle. The display window rectangle defines the visible portion of the virtual display. The world coordinate space of the display window rectangle is mapped to the display screen as the display viewport.

If these coordinates are not specified, the entire world coordinate space specified in the UIS$CREATE_DISPLAY routine is used.

width
height

VMS Usage: **floating_point**
type: **f_floating**
access: **read only**
mechanism: **by reference**

Initial dimensions of the display viewport. The **width** and **height** arguments are addresses of f_floating point numbers that define the width and height of the display viewport in centimeters. If the **width** and **height** arguments of the display viewport specified in UIS$CREATE_WINDOW are different from the **width** and **height** arguments specified in the UIS$CREATE_DISPLAY routine, the default values of UIS$CREATE_DISPLAY are overridden and scaling occurs.

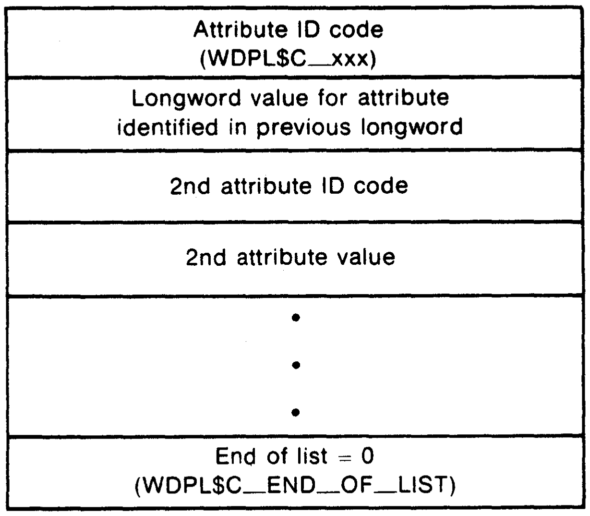
If the world coordinates of the display window **are** specified and the **width** and **height** arguments are **not** specified, the default dimensions of the display viewport are calculated from the ratios of the world coordinate values and the width and height specified in UIS$CREATE_DISPLAY. See the Description section for more information about calculating the default display viewport dimensions.

Display viewports that are too large to fit on the screen are automatically proportionally scaled in size.

attributes

VMS Usage: **item_list_pair**
type: **longword integer (signed) or f_floating**
access: **read only**
mechanism: **by reference**

Display viewport attribute list. The **attributes** argument is the address of a data structure that contains longword pairs, or *doublets*. The first longword stores an attribute ID code and the second longword holds the attribute value (which can be real or integer). The constant WDPL$C_END_OF_LIST terminates this list. FORTRAN application programs should create a record using the RECORD statement to construct this list. It has the following format.



ZK-4581-85

Window attributes are optional and control window placement and attributes.

Attribute	Description
WDPL$C_ABS_POS_X	Exact x placement on the screen. This attribute defines the x origin of the viewport relative to the lower-left corner of the screen. The value is expressed as an f_floating point number of centimeters. Note that the actual point WDPL$C_ABS_POS_X defines is the lower-left corner of the display viewport without the border. Along with WDPL$C_ABS_POS_Y, this provides the ability to place exactly a new viewport at a specific position anywhere on the workstation screen.
WDPL$C_ABS_POS_Y	Exact y placement on the screen. This attribute defines the y origin of the viewport relative to the lower-left corner of the screen. The value is expressed as an f_floating point number of centimeters. Note that the actual point WDPL$C_ABS_POS_Y defines is the lower-left corner of the display viewport without the border. Along with WDPL$C_ABS_POS_X, this attribute provides the ability to place exactly a new viewport at a specific position anywhere on the workstation screen.

Attribute	Description
WDPL$C_PLACEMENT	Display viewport placement flags. The attribute list is a longword bit vector providing viewport placement information. To combine the preference masks (top, bottom, left, and right), set more than one bit in the bit vector. If the screen becomes crowded, the system might override the preference masks. • WDPL$M_TOP—The display viewport is placed near the top of the physical display • WDPL$M_BOTTOM—The display viewport is placed near the bottom of the physical display • WDPL$M_LEFT—The display viewport is placed near the left side of the physical display • WDPL$M_RIGHT—The display viewport is placed near the right side of the physical display • WDPL$M_CENTER—The display viewport is centered over the position specified by WDPL$C_ABS_POS_X and WDPL$C_ABS_POS_Y. • WDPL$M_INVISIBLE—The display viewport is created invisibly, that is, off the screen and, hence, cannot be seen. • Other bits—The remaining bits are reserved to DIGITAL and must be set to zero.
WDPL$C_ATTRIBUTES	Display viewport attributes. This data structure argument causes the display viewport to be created with one or more of the following attributes. These attributes are specified as bits in a longword mask.

Attribute	Description
	• WDPL$M_ALIGNED—The left inner edge of the display viewport is to be aligned on byte boundaries. Applications, such as the VT220 terminal emulator, can use WDPL$M_ALIGNED to take advantage of text drawing performance optimizations when 8-bit characters are written on byte boundaries. • WDPL$M_NOBANNER—The display viewport is created without a banner. If a banner title was specified, it is ignored. • WDPL$M_NOBORDER—The display viewport is created without a border. When you specify WDPL$M_NOBORDER, the attribute WDPL$M_NOBANNER is implied. A viewport created without a border cannot be moved with the user interface. • WDPL$M_NOKB_ICON—The display viewport banner is created without a KB icon. Specify this attribute, if you are sure the application will never require a KB icon or if you wish to add more space in the banner for the title. Otherwise, UIS saves an extra quarter of an inch in the banner for the KB icon. • WDPL$M_NOMENU_ICON—The display viewport banner is created without a menu icon. Therefore, the Window Options Menu cannot be activated. • Other bits—The remaining bits are reserved to DIGITAL and must be zero.
WDPL$C_END_OF_LIST	Terminates attributes list. This must be the last longword in the attribute list. It does not require an associated longword value.

DESCRIPTION

UIS$CREATE_WINDOW defines a portion of the virtual display that lies within the display window and that is mapped to the display screen as the display viewport.

Default Dimensions of the Display Viewport

Whenever the world coordinates of the display window are defined, but the dimensions of the display viewport are not specified, the system calculates the default dimensions of the display viewport using the appropriate arguments from each routine as shown in the following figure. The size of the display viewport is based on the **width** and **height** arguments in UIS$CREATE_DISPLAY in the following manner:

UIS$CREATE_DISPLAY UIS$CREATE_WINDOW

$$\frac{\text{width}}{x_2 - x_1} = \frac{\text{new\_width}}{x_2 - x_1}$$

$$\frac{\text{height}}{y_2 - y_1} = \frac{\text{new\_height}}{y_2 - y_1}$$

ZK-5462-86

The variables new_width and new_height represent unknown quantities, the default dimensions of the display viewport. All other variables are the parameters used in the respective routine calls.

For example, the viewport that is created in the following example is 4 centimeters wide and 2 centimeters high.

```
vd_id=UIS$CREATE_DISPLAY(0.0,0.0,1.0,1.0,8.0,4.0)
wd_id=UIS$CREATE_WINDOW(vd_id,'SYS$WORKSTATION','TEST WINDOW',
                         0.0,0.0,0.5,0.5)
```

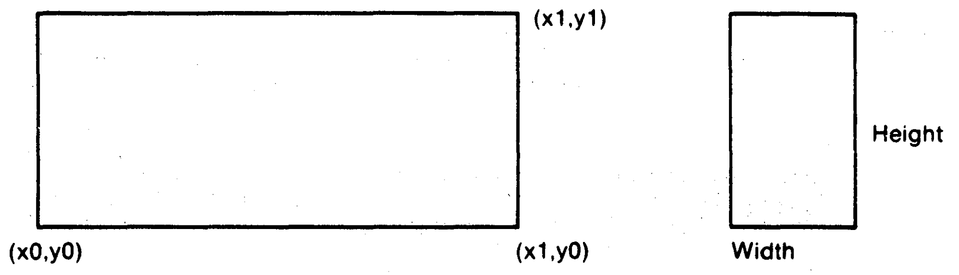
Otherwise, these values can be overridden with the optional **width** and **height** arguments in UIS$CREATE_WINDOW.

Display Viewport Creation

Display viewports are always created completely on or off the display screen.

Distortion of Graphic Objects

To avoid distortion of graphic objects, the aspect ratios of the display window and the display viewport must be equal.



ZK-4582-85

In the preceding illustration, the aspect ratio of the display window on the left does not appear to be equal to the aspect ratio of the viewport on the right.

You can compare aspect ratios using the following equation.

$$\frac{|y1 - y0|}{|x1 - x0|} = \frac{\text{viewport height}}{\text{viewport width}}$$

ZK-4579-85

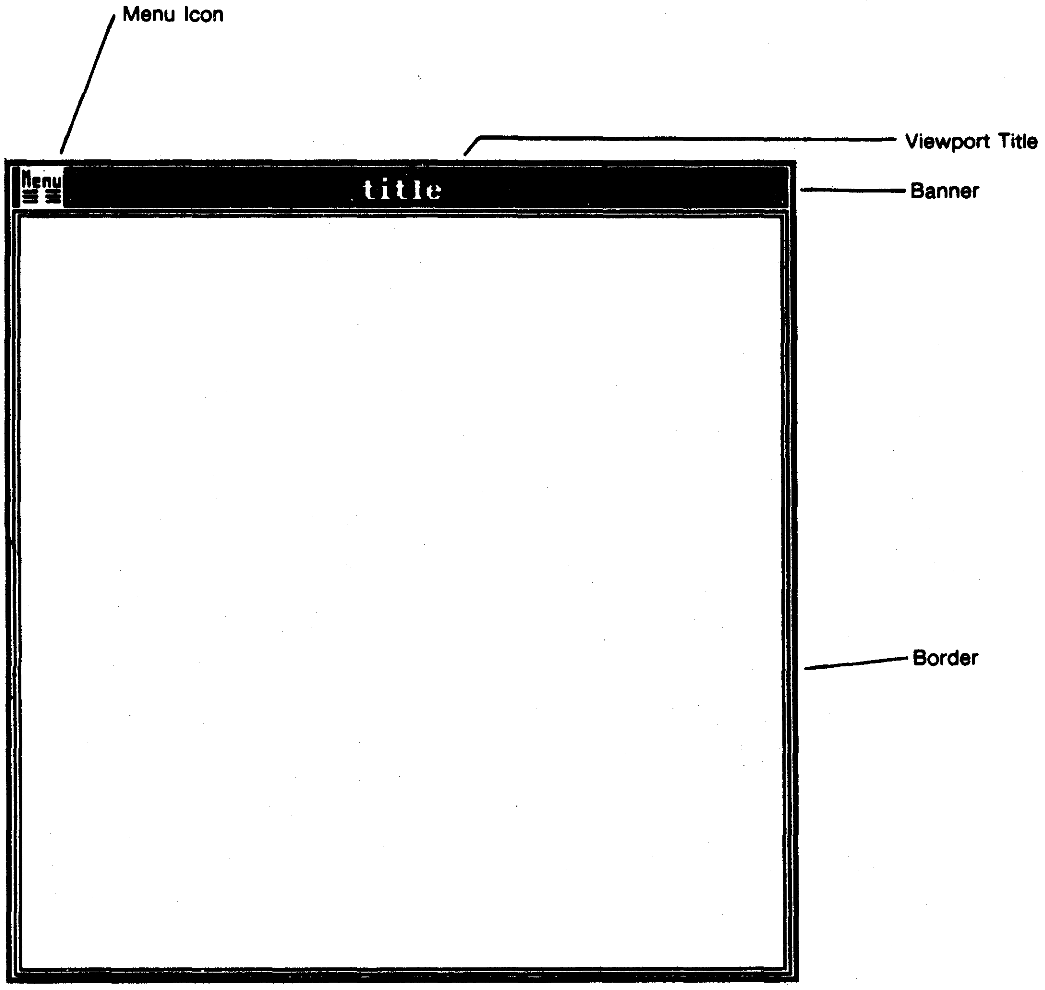
The aspect ratio of the display viewport is the absolute value of the height divided by the absolute value of the width.

EXAMPLE

```
PROGRAM EXAMPLE_A
   .
   .
   .
STRUCTURE/STRUCT/       1
   INTEGER*4 CODE_1
   REAL*4  ATTRIB_1
   INTEGER*4 CODE_2
   REAL*4  ATTRIB_2
   INTEGER*4 CODE_3
   INTEGER*4 ATTRIB_3
   INTEGER*4 END
END STRUCTURE

RECORD/STRUCT/WINDOW    2

WINDOW.CODE_1=WDPL$C_ABS_POS_X
WINDOW.ATTRIB_1=10.5
WINDOW.CODE_2=WDPL$C_ABS_POS_Y
WINDOW.ATTRIB_2=13.25
WINDOW.CODE_3=WDPL$C_ATTRIBUTES
WINDOW.ATTRIB_3=WDPL$M_NOKB_ICON .OR. WDPL$M_NOMENU_ICON
WINDOW.END=WDPL$C_END_OF_LIST
   .
   .
   .
VD_ID=UIS$CREATE_DISPLAY(-10.0,-10.0,35.5,35.5,16.0,16.0)  3

WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','LOOK',2.0,2.0,28.0,28.0
2       20.0,20.0,WINDOW)    4
   .
   .
   .
```

This example describes how to construct the data structure argument used in UIS$CREATE_WINDOW to enable viewport placement and characteristics 1 2. In addition, the example illustrates the minimum number of calls used to create a display window 3 4.

screen output



ZK-5278-86

UIS$DELETE_COLOR_MAP

Deletes a virtual color map.

FORMAT

UIS$DELETE_COLOR_MAP *vcm_id*

RETURNS

UIS$DELETE_COLOR_MAP signals all errors; no condition values are returned.

ARGUMENT

vcm_id

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Virtual color map identifier. The **vcm_id** argument is the address of a longword that uniquely identifies the virtual color map. See UIS$CREATE_COLOR_MAP for more information about the **vcm_id** argument.

DESCRIPTION

An attempt to delete an active virtual color map, that is, a virtual color map associated with one or more virtual displays, signals an error.

Use UIS$DELETE_DISPLAY first to delete all virtual displays that reference the virtual color map.

UIS$DELETE_COLOR_MAP_SEG

Deletes the specified color map segment.

FORMAT

UIS$DELETE_COLOR_MAP_SEG *cms_id*

RETURNS

UIS$DELETE_COLOR_MAP_SEG signals all errors; no condition values are returned.

ARGUMENT

cms_id

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Color map segment identifier. The **cms_id** argument is the address of a longword that uniquely identifies the color map segment to be deleted. See UIS$CREATE_COLOR_MAP_SEG for more information about the **cms_id** argument.

DESCRIPTION

Color map segment deletion has no effect on the colors being mapped by the hardware color map. The deletion of color map segments marks the corresponding entries as available for allocation.

An attempt to delete an active color map segment, that is, a color map segment referenced by a virtual color map, signals an error.

Use UIS$DELETE_COLOR_MAP first to delete the virtual color map.

UIS$DELETE_DISPLAY

Deletes the virtual display, all associated windows, and viewports.

FORMAT **UIS$DELETE_DISPLAY** *vd_id*

RETURNS UIS$DELETE_DISPLAY signals all errors; no condition values are returned.

ARGUMENT ***vd_id***
See Section 18.3.1 for a description of this argument.

DESCRIPTION You cannot substitute the **tr_id** argument for the virtual display identifier in this routine.

UIS$DELETE_KB

Deletes a virtual keyboard. If the specified virtual keyboard is bound to a window or to the physical keyboard, those bindings are terminated.

FORMAT

UIS$DELETE_KB *kb_id*

RETURNS

UIS$DELETE_KB signals all errors; no condition values are returned.

ARGUMENT

kb_id
See Section 18.3.8 for more information about the **kb_id** argument.

DESCRIPTION

You can use UIS$DELETE_KB to delete a virtual keyboard at any time within a program.

UIS$DELETE_OBJECT

Deletes the specified object from the virtual display.

FORMAT

UIS$DELETE_OBJECT $\left\{ \begin{array}{l} \textit{obj\_id} \\ \textit{seg\_id} \end{array} \right\}$

RETURNS

UIS$DELETE_OBJECT signals all errors; no condition values are returned.

ARGUMENTS

obj_id
See Section 18.3.3 for a description of this argument.

seg_id
See Section 18.3.4 for a description of this argument.

DESCRIPTION

The screen is updated immediately to reflect the new state of the virtual display. If it is impossible to modify only the changed portions, the entire display can be replotted. Occluded objects are always refreshed.

UIS$DELETE_PRIVATE

Deletes the private data associated with the object.

FORMAT

UIS$DELETE_PRIVATE $\left\{ \begin{array}{l} \textit{obj\_id} \\ \textit{seg\_id} \end{array} \right\}$

RETURNS

UIS$DELETE_PRIVATE signals all errors; no condition values are returned.

ARGUMENTS

obj_id
See Section 18.3.3 for a description of this argument.

seg_id
See Section 18.3.4 for a description of this argument.

DESCRIPTION

If more than one private data item exists, all private data items are deleted.

UIS$DELETE_TB

Deletes the tablet digitizer identifier and disconnects the application from the tablet.

FORMAT

UIS$DELETE_TB *tb_id*

RETURNS

UIS$DELETE_TB signals all errors; no condition values are returned.

ARGUMENT

tb_id

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Tablet identifier. The **tb_id** argument is the address of a longword that uniquely identifies the tablet device. See UIS$CREATE_TB for more information about the **tb_id** argument.

DESCRIPTION

UIS$DELETE_TB deletes a tablet digitizing identifier. When your process has completed digitizing, you should call this routine to delete the identifier.

UIS$DELETE_TRANSFORMATION

Deletes a world coordinate transformation of a virtual display. The corresponding virtual display is not affected.

FORMAT

UIS$DELETE_TRANSFORMATION *tr_id*

RETURNS

UIS$DELETE_TRANSFORMATION signals all errors; no condition values are returned.

ARGUMENT

tr_id

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Transformation identifier. The **tr_id** argument is the address of a longword that uniquely identifies the transformation to be deleted. See UIS$CREATE_TRANSFORMATION for more information about the **tr_id** argument.

UIS$DELETE_WINDOW

Deletes an existing display window and viewport.

FORMAT **UIS$DELETE_WINDOW** *wd_id*

RETURNS UIS$DELETE_WINDOW signals all errors; no condition values are returned.

ARGUMENT ***wd_id***
See Section 18.3.2 for a description of this argument.

DESCRIPTION UIS$DELETE_WINDOW deletes the display window specified by the **wd_id** argument. The associated viewport is removed from the screen. The virtual display associated with this display window is neither modified nor destroyed during the execution of this service.

UIS$DISABLE_DISPLAY_LIST

Disables specified display list functions.

FORMAT

UIS$DISABLE_DISPLAY_LIST *vd_id [,display_flags]*

RETURNS

UIS$DISABLE_DISPLAY_LIST signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

display_flags

VMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Display list flags. The **display_flags** argument is the address of a longword mask that controls display screen and display list updates.

The following table describes the flags and masks.

Flag	Description
UIS$M_DL_ENHANCE_LIST	Controls making additions to the display list. When disabled, no new display list entries are made. This flag is set by default when a virtual display is created.
UIS$M_DL_MODIFY_LIST	Controls display list modifications. When disabled, no display list editing is allowed. This flag is set by default when a virtual display is created.
UIS$M_DL_UPDATE_WINDOW	Controls drawing. When disabled, no drawing or update occurs. This flag is set by default when a virtual display is created.

The following table lists UIS routines that check the flags.

Flag	UIS Routine
UIS$M_DL_MODIFY_LIST[1]	UIS$COPY_OBJECT
	UIS$DELETE_OBJECT
	UIS$ERASE
	UIS$INSERT_OBJECT
	UIS$MOVE_AREA
	UIS$TRANSFORM_OBJECT
UIS$M_DL_ENHANCE_LIST[1]	UIS$CIRCLE
	UIS$ELLIPSE
	UIS$EXECUTE
	UIS$EXECUTE_DISPLAY
	UIS$IMAGE
	UIS$LINE
	UIS$LINE_ARRAY
	UIS$PLOT
	UIS$PLOT_ARRAY
	UIS$TEXT

[1]All routines listed under UIS$M_DL_ENHANCE_LIST and UIS$M_DL_MODIFY_LIST will also check the state of UIS$M_DL_UPDATE_WINDOW before doing any screen updates.

If a bit is set in the mask, the corresponding function is disabled. If the bit is 0, the corresponding function is not changed. See UIS$ENABLE_DISPLAY_LIST for information on how to enable functions.

If **display_flags** is not specified, UIS$M_DL_ENHANCE_LIST is disabled.

DESCRIPTION

UIS$DISABLE_DISLAY_LIST is useful in applications such as animation. In such a case, display list additions are neither necessary nor desired because of the additional overhead.

EXAMPLE

At some point in your application you might want to perform several modifications to the display list without seeing the screen change.

```
    .
    .
    .
display_flags= UIS$M_DL_UPDATE_WINDOW
    .
    .
    .
CALL UIS$DISABLE_DISPLAY_LIST(VD_ID, DISPLAY_FLAGS)
    .
    .
    .
Insert your modifications here
    .
    .
    .
CALL UIS$ENABLE_DISPLAY_LIST(VD_ID, DISPLAY_FLAGS)
CALL UIS$EXECUTE(VD_ID)     ! Erases and redraws the virtual display
```

UIS$DISABLE_KB

Disconnects the physical keyboard from the specified virtual keyboard. See the example in UIS$CREATE_KB for more information.

FORMAT **UIS$DISABLE_KB** *kb_id*

RETURNS UIS$DISABLE_KB signals all errors; no condition values are returned.

ARGUMENT ***kb_id***
See Section 18.3.8 for more information about the **kb_id** argument.

UIS$DISABLE_TB

Disconnects the digitizing tablet.

FORMAT

UIS$DISABLE_TB *tb_id*

RETURNS

UIS$DISABLE_TB signals all errors; no condition values are returned.

ARGUMENT

tb_id

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Tablet identifier. The **tb_id** argument is the address of longword that uniquely identifies the tablet device. See UIS$CREATE_TB for more information about the **tb_id** argument.

DESCRIPTION

UIS$DISABLE_TB disconnects your process from the tablet. This routine reenables the system pointer and frees the tablet for use by another process.

UIS$DISABLE_VIEWPORT_KB

Prevents the user from assigning the physical keyboard to a viewport. See the example in UIS$CREATE_KB for more information.

FORMAT

UIS$DISABLE_VIEWPORT_KB *wd_id*

RETURNS

UIS$DISABLE_VIEWPORT_KB signals all errors; no condition values are returned.

ARGUMENT

wd_id
See Section 18.3.2 for a description of this argument.

DESCRIPTION

UIS$DISABLE_VIEWPORT_KB removes the display window from the assignment list. You can no longer use the CYCLE key to make the viewport active. Use UIS$ENABLE_VIEWPORT_KB or UIS$ENABLE_KB to place the display window on the assignment list.

UIS$ELLIPSE

Draws an arc along the circumference of an ellipse.

FORMAT

UIS$ELLIPSE *vd_id, atb, center_x, center_y, xradius, yradius [,start_deg ,end_deg]*

RETURNS

UIS$ELLIPSE signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword integer that identifies the attribute block that will modify the ellipse. If you specify *0* in the **atb** argument, the default settings of attribute block 0 are used.

center_x
center_y

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Center position x and y world coordinates. The **center_x** and **center_y** arguments are the addresses of f_floating point numbers that define a point in the virtual display that is the center of the ellipse or arc.

xradius
yradius

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Radii of the ellipses specified as x and y world coordinate widths. The **xradius** argument is the address of an f_floating point number that defines the distance from the center of the ellipse to the circumference of the ellipse or arc. The **yradius** argument is the address of an f_floating point number that defines the distance from the center of the ellipse to the circumference of the ellipse or arc.

start_deg
end_deg

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Degree at which the arc starts and ends. The **start_deg** and **end_deg** arguments are the addresses of f_floating numbers that define the starting point and ending point in degrees on the circumference of the ellipse where the arc or ellipse will be drawn. Degrees are measured clockwise from the top of the ellipse. If you do not specify **start_deg**, *0.0* degrees is assumed; if you do not specify **end_deg**, *360.0* degrees is assumed. If you specify neither argument, a complete ellipse is drawn.

DESCRIPTION

UIS$ELLIPSE uses center position coordinates and *x* and *y* radii to construct an ellipse. Along the circumference of this ellipse, UIS$ELLIPSE draws an arc for a specified range of degrees.

The arc is closed by drawing one or more lines between the endpoints. The type of arc associated with the attribute block specifies the way in which the arc is closed. See the UIS$SET_ARC_TYPE routine.

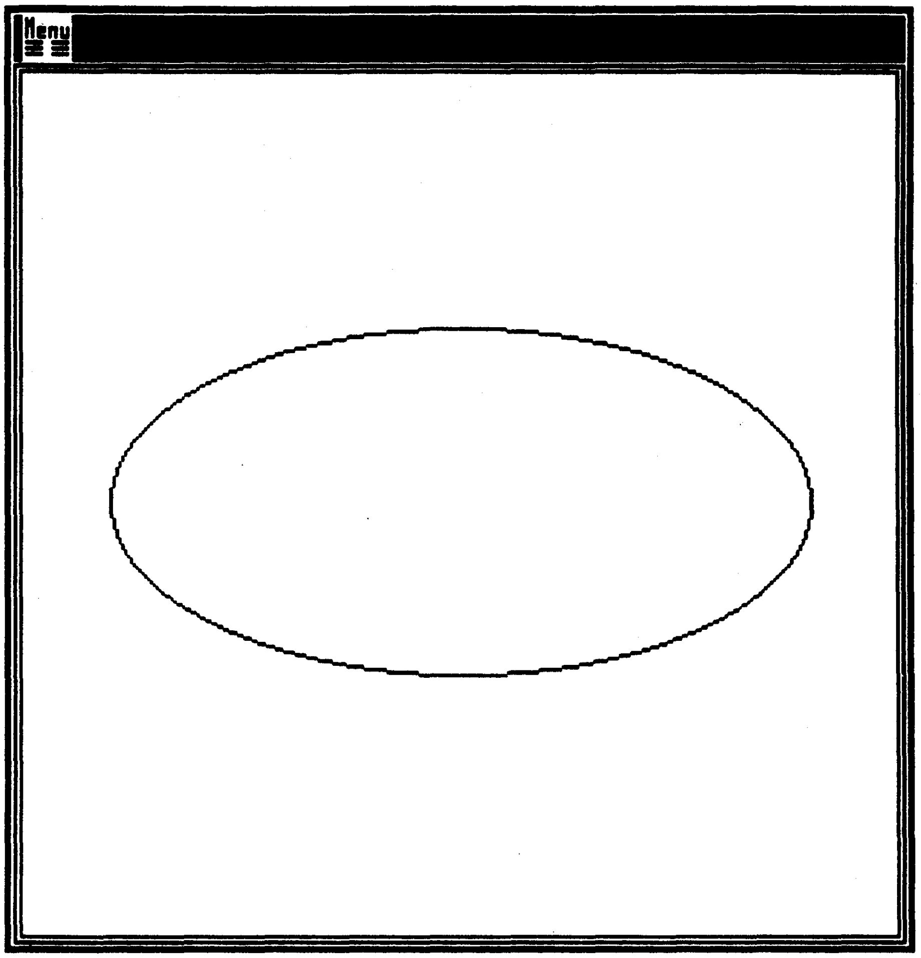
The points are drawn with the current line pattern and width, and filled with the current fill pattern, if enabled.

UIS$ELLIPSE does not support the following combination of attributes:

- Line width not equal to *1* and line style not equal to $FFFFFFFF_{16}$
- Line width not equal to *1* and complement writing mode

Ellipses are distorted by differences between the aspect ratios of the virtual display and display window.

screen output



ZK-5418-86

UIS$ENABLE_DISPLAY_LIST

Reenables automatic additions to the display list.

FORMAT

UIS$ENABLE_DISPLAY_LIST *vd_id [,display_flags]*

RETURNS

UIS$ENABLE_DISPLAY_LIST signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

display_flags

VMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Display list flags. The **display_flags** argument is the address of a longword mask that controls display screen and display list updates.

The following table describes the flags and masks.

Flag	Description
UIS$M_DL_ENHANCE_LIST	Controls making additions to the display list. When disabled, no new display list entries are made. This flag is set by default when a virtual display is created.
UIS$M_DL_MODIFY_LIST	Controls display list modifications. When disabled, no display list editing is allowed. This flag is set by default when a virtual display is created.
UIS$M_DL_UPDATE_WINDOW	Controls drawing. When disabled, no drawing or update occurs. This flag is set by default when a virtual display is created.

The following table lists UIS routines that check the flags.

Flag	UIS Routine
UIS$M_DL_MODIFY_LIST[1]	UIS$COPY_OBJECT UIS$DELETE_OBJECT UIS$ERASE UIS$INSERT_OBJECT UIS$MOVE_AREA UIS$TRANSFORM_OBJECT
UIS$M_DL_ENHANCE_LIST[1]	UIS$CIRCLE UIS$ELLIPSE UIS$EXECUTE UIS$EXECUTE_DISPLAY UIS$IMAGE UIS$LINE UIS$LINE_ARRAY UIS$PLOT UIS$PLOT_ARRAY UIS$TEXT

[1]All routines listed under UIS$M_DL_ENHANCE_LIST and UIS$M_DL_MODIFY_LIST will also check the state of UIS$M_DL_UPDATE_WINDOW before doing any screen updates.

If a bit is set in the mask, the corresponding function is disabled. If the bit is 0, the corresponding function is not changed.

If **display_flags** is not specified, UIS$M_DL_ENHANCE_LIST is disabled.

EXAMPLE

At some point in your application you might wish to perform several modifications to the display list without seeing the screen change.

```
     .
     .
     .
display_flags= UIS$M_DL_UPDATE_WINDOW
     .
     .
     .
CALL UIS$DISABLE_DISPLAY_LIST(VD_ID, DISPLAY_FLAGS)
     .
     .
     .
Insert your modifications here
     .
     .
     .
CALL UIS$ENABLE_DISPLAY_LIST(VD_ID, DISPLAY_FLAGS)
CALL UIS$EXECUTE(VD_ID)     ! Erases and redraws the virtual display
```

UIS$ENABLE_KB

Connects the physical keyboard to the specified virtual keyboard. See the example in UIS$CREATE_KB for more information.

FORMAT

UIS$ENABLE_KB *kb_id [,wd_id]*

RETURNS

UIS$ENABLE_KB signals all errors; no condition values are returned.

ARGUMENTS

kb_id
See Section 18.3.8 for more information about the **kb_id** argument.

wd_id
See Section 18.3.2 for more information about the **wd_id** argument.

DESCRIPTION

Because you should be able to control the keyboard, it is recommended that you use the UIS$ENABLE_KB as little as possible. However, there are times when you might want to use it:

- When you start up a new application—In this case, you might want the workstation keyboard to be implicitly connected to a new application.
- When the physical keyboard is already connected to the application (as determined by the UIS$TEST_KB routine)—In this case, the application might have to facilitate movement of the keyboard between its windows.

Note that these are not restrictions imposed by the workstation software.

UIS$ENABLE_TB

Assigns the tablet to the calling process.

FORMAT

UIS$ENABLE_TB *tb_id*

RETURNS

UIS$ENABLE_TB signals all errors; no condition values are returned.

ARGUMENT

tb_id

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Tablet identifier. The **tb_id** argument is the address of a longword that uniquely identifies a tablet device. See UIS$CREATE_TB for more information about the **tb_id** argument.

DESCRIPTION

Only one application at a time can own the tablet. When a process connects to the tablet, the system hardware cursor is turned off and the connected process receives all input from the tablet device. The process owns the tablet until it calls UIS$DISABLE_TB to disconnect itself from the tablet.

The process must use a software cursor to track the pointer in a display window.

UIS$ENABLE_VIEWPORT_KB

Allows the user to assign a virtual keyboard to the physical keyboard and signals binding through the KB icon in the viewport banner. See the example in UIS$CREATE_KB for more information.

FORMAT

UIS$ENABLE_VIEWPORT_KB *kb_id, wd_id*

RETURNS

UIS$ENABLE_VIEWPORT_KB signals all errors; no condition values are returned.

ARGUMENTS

kb_id
See Section 18.3.8 for more information about the **kb_id** argument.

wd_id
See Section 18.3.2 for a description of this argument.

DESCRIPTION

UIS$ENABLE_VIEWPORT_KB makes the display window as a KB handle.

The viewport contains a nonhighlighted KB icon.

UIS$END_SEGMENT

Ends a current segment in a virtual display.

FORMAT **UIS$END_SEGMENT** *vd_id*

RETURNS UIS$END_SEGMENT signals all errors; no condition values are returned.

ARGUMENT ***vd_id***
See Section 18.3.1 for a description of this argument.

DESCRIPTION Context is returned to the parent segment. All values of attribute blocks 0 to 255 are restored to the current values of the parent's attribute blocks.

UIS$ERASE

Erases the specified rectangle in the virtual display and removes all entities that lie **completely** within the rectangle from the display list.

FORMAT

UIS$ERASE *vd_id [,$x_1$, y_1, x_2, y_2]*

RETURNS

UIS$ERASE signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

x_1, y_1
x_2, y_2

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

World coordinate pairs. The x_1 and y_1 arguments are the addresses of f_floating point numbers that define the lower-left corner of the rectangle in the virtual display. The x_2 and y_2 arguments are the addresses of f_floating point numbers that define the upper-right corner of the rectangle in the virtual display. If no rectangle is specified, the entire virtual display is erased.

DESCRIPTION

UIS$ERASE removes all graphics entities that lie completely within the rectangle from the display list as if they had never been written. Objects that do not lie completely within the specified rectangle are not erased. Empty segments are not deleted.

Areas within the display window affected by this routine are filled with color specified by entry 0 in the color map of the virtual display.

UIS$EXECUTE

Executes a binary encoding stream in a specified virtual display.

FORMAT

UIS$EXECUTE *vd_id [,buflen] [,bufaddr]*

RETURNS

UIS$EXECUTE signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

buflen

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Length of the binary encoding stream. The **buflen** argument is the address of longword that contains the length of the binary encoding stream.

bufaddr

VMS Usage:	**vector_longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**reference**

Binary encoding stream. The **bufaddr** argument is the address of an array of longwords that compose the binary encoding stream.

DESCRIPTION

If the buffer is omitted, all display windows are erased and refreshed.

Note the effects of the display list flags.

UIS$EXECUTE_DISPLAY

Creates a virtual display from a display list.

FORMAT

vd_id = **UIS$EXECUTE_DISPLAY** *buflen, bufaddr*

RETURNS

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Longword value returned as the virtual display identifier in the variable *vd_id* or R0 (VAX MACRO).

UIS$EXECUTE_DISPLAY signals all errors; no condition values are returned.

ARGUMENTS

buflen

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Buffer length. The **buflen** argument is the address of a longword that defines the length of the buffer.

bufaddr

VMS Usage:	**vector_byte_unsigned**
type:	**byte integer (unsigned)**
access:	**read only**
mechanism:	**by reference**

Buffer address. The **bufaddr** argument is the address of an array of integer bytes that contains the binary encoded stream.

The binary encoded stream is executed in the virtual display.

UIS$EXPAND_ICON

Replaces an icon with its associated viewport.

FORMAT **UIS$EXPAND_ICON** *wd_id [,icon_wd_id] [,attributes]*

RETURNS UIS$EXPAND_ICON signals all errors; no condition values are returned.

ARGUMENTS

wd_id
See Section 18.3.2 for a description of this argument.

icon_wd_id

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

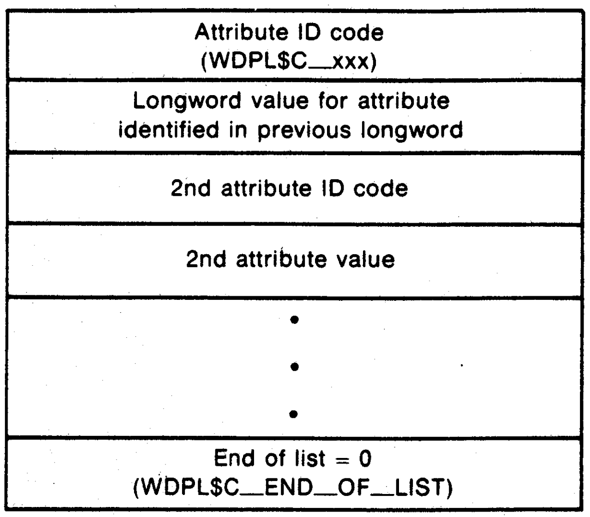
Icon window identifier. The **icon_wd_id** argument is the address of a longword that uniquely identifies the icon window.

If the **icon_wd_id** argument is specified, it must match the value of the **icon_wd_id** argument specified in UIS$SHRINK_TO_ICON.

attributes

VMS Usage:	**item_list_pair**
type:	**longword integer (signed) or f_floating**
access:	**read only**
mechanism:	**by reference**

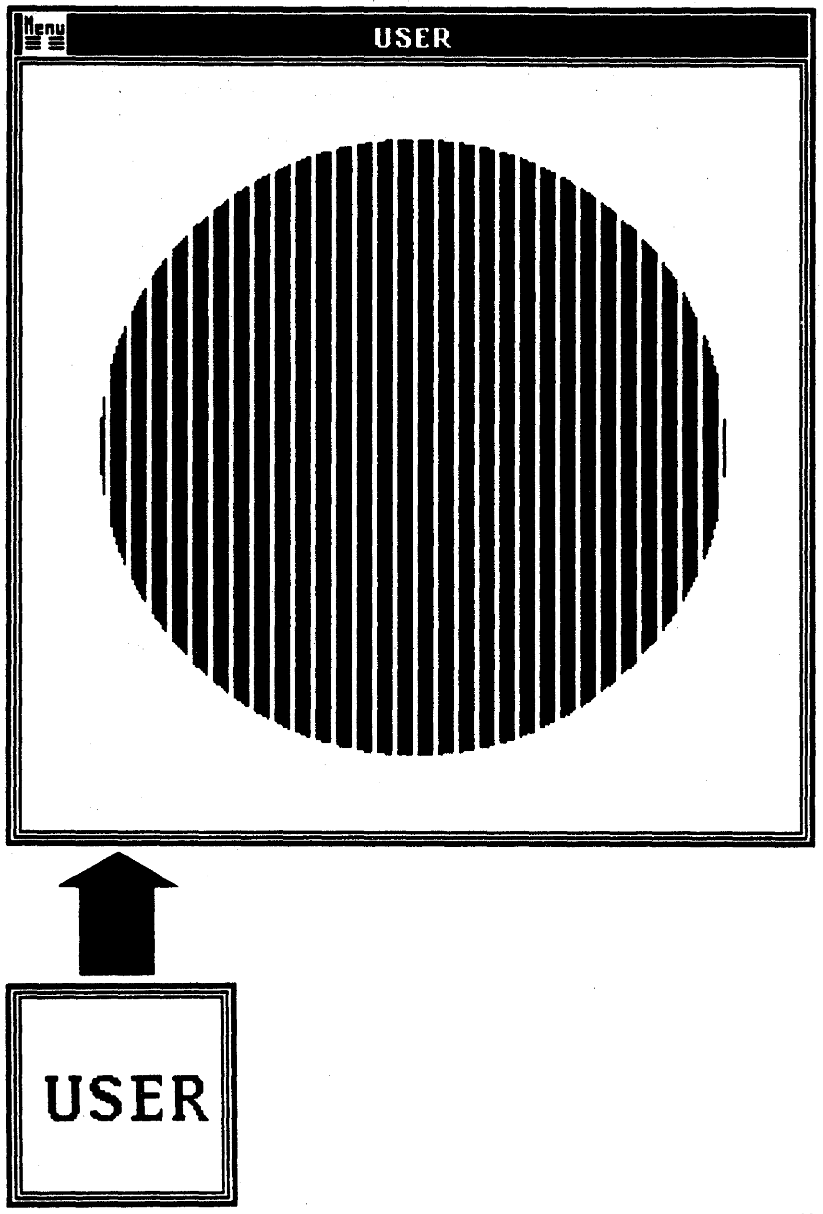
Viewport attributes list. The **attributes** argument is the address of a data structure such as an array or record. The **attributes** can be used to specify exact placement of the display viewport.



ZK-4581-85

See the **attributes** argument in UIS$CREATE_WINDOW for more information.

screen output



ZK-5447-86

UIS$EXTRACT_HEADER

Returns the header information needed to create a UIS metafile.

FORMAT

UIS$EXTRACT_HEADER *vd_id, [buflen, bufaddr] [,retlen]*

RETURNS

UIS$EXTRACT_HEADER signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

buflen

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Buffer length. The **buflen** argument is the address of a longword that defines the length of the buffer.

bufaddr

VMS Usage:	**vector_byte_unsigned**
type:	**byte integer (unsigned)**
access:	**read only**
mechanism:	**by reference**

Buffer address. The **bufaddr** argument is the address of an array of bytes that receives the binary encoding stream.

retlen

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Return length. The **retlen** argument is the address of a longword that receives the length of the buffer.

DESCRIPTION

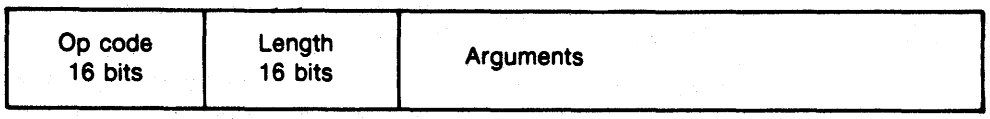
Header information must be at the beginning of all UIS metafiles.

Allocating Space for the Buffer

If you want to know how much space to allocate for the buffer, specify **obj_id** and **retlen** only.

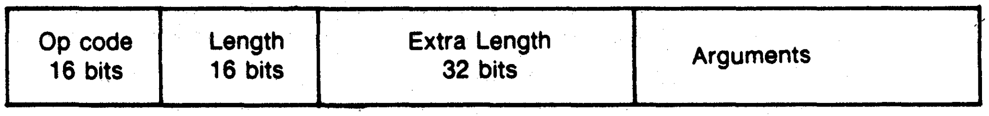
Format of Header Information

The format of header binary instructions is as follows:



ZK-5472-86

If the length field exceeds 32,767 bytes, an extended format is used. The length field should be set to UIS$C_LENGTH_DIFF and the extra length field should be set to the total number of bytes in the binary instruction.



ZK-5473-86

UIS$EXTRACT_OBJECT

Returns the binary encoding stream for the desired object (segment or primitive).

FORMAT

UIS$EXTRACT_OBJECT $\left\{ \begin{array}{l} \textit{obj\_id} \\ \textit{seg\_id} \end{array} \right\}$ *[,buflen ,bufaddr] [,retlen]*

RETURNS

UIS$EXTRACT_OBJECT signals all errors; no condition values are returned.

ARGUMENTS

obj_id
See Section 18.3.3 for a description of this argument.

seg_id
See Section 18.3.4 for a description of this argument.

buflen

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Length of buffer. The **buflen** argument is the address of a longword that specifies the length of the buffer that receives the binary encoding stream.

bufaddr

VMS Usage:	**vector_byte_unsigned**
type:	**byte (unsigned)**
access:	**read only**
mechanism:	**by reference**

Name of an array. The **bufaddr** argument is the address of an array of bytes that receives the binary encoding stream.

retlen

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Length of the binary encoding stream. The **retlen** argument is the address of a longword that receives the length of the binary encoding stream.

DESCRIPTION

If you want to know how much space to allocate for the buffer, specify **obj_id** and **retlen** only.

If the extracted object lies within a segment, a binary instruction denoting the beginning of the segment precedes all binary instructions associated with the extracted object. A binary instruction denoting the end of the segment follows the binary instructions associated with the extracted object.

UIS$EXTRACT_PRIVATE

Returns the binary data associated with the specified object.

FORMAT

UIS$EXTRACT_PRIVATE $\left\{ \begin{array}{l} \textit{obj\_id} \\ \textit{seg\_id} \end{array} \right\}$ *[,buflen ,bufaddr] [,retlen]*

RETURNS

UIS$EXTRACT_PRIVATE signals all errors; no condition values are returned.

ARGUMENTS

obj_id
See Section 18.3.3 for a description of this argument.

seg_id
See Section 18.3.4 for a description of this argument.

buflen

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Length of the buffer. The **buflen** argument is the address of a longword that contains the length of the buffer that receives the binary encoding stream.

bufaddr

VMS Usage:	**vector_byte_unsigned**
type:	**byte (unsigned)**
access:	**read only**
mechanism:	**by reference**

Buffer address. The **bufaddr** argument is the address of an array of bytes that receives the binary encoding stream.

retlen

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by reference**

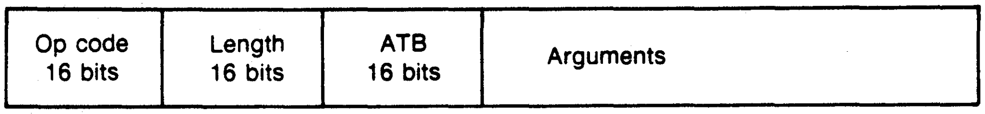
Length of the binary encoding stream. The **retlen** is the address of longword that receives the length of the binary encoding stream.

DESCRIPTION

If more than one private data item is associated with the specified object, all private data items are returned. The following figure describes the format of the data. If you want to know how much space to allocate for the returned encoding, specify the **obj_id** and **retlen** arguments only.

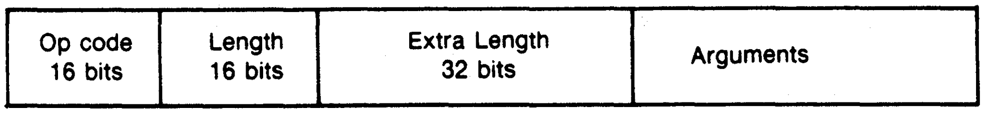
Format of a Private Data Binary Instruction

The format of binary encoding returned is as follows:



ZK-5475-86

If the length field exceeds 32,767 bytes, an extended format is used. The length field should be set to UIS$C_LENGTH_DIFF and the extra length field should be set to the total number of bytes in the binary instruction.



ZK-5473-86

Attribute modification instructions precede the binary instruction of the extracted object. The binary instructions of any private data associated with the extracted object follow the binary instruction of the extracted object.

UIS$EXTRACT_REGION

Locates all output primitives and portions of output primitives that lie entirely within the specified rectangle, and returns the binary encoding stream for the selected display.

FORMAT

UIS$EXTRACT_REGION *vd_id [,$x_1$,$y_1$, x_2,y_2] [,buflen ,bufaddr] [,retlen]*

RETURNS

UIS$EXTRACT_REGION signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

x_1, y_1
x_2, y_2

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

World coordinates of the specified rectangle. The x_1,y_1 and x_2,y_2 arguments are the addresses of f_floating point numbers that define the lower-left and upper-right corners of the specified rectangle.

If you specify a region within the virtual display, UIS$EXTRACT_REGION returns the entire display list except for the following:

- Objects that do not lie completely within the specified region
- Segments that do not contain any objects that fall completely within the specified region

If these arguments are not specified, the coordinates of the entire virtual display are used.

buflen

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Length of a buffer. The **buflen** is the address of a longword that contains the length of the buffer that receives the binary encoding stream.

bufaddr

VMS Usage:	**vector_byte_unsigned**
type:	**byte_unsigned**
access:	**read only**
mechanism:	**by reference**

Buffer address. The **bufaddr** argument is the address of an array of bytes that receives the binary encoding stream.

retlen

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Length of the binary encoding stream. The **retlen** argument is the address of a longword that receives the length of the binary encoding stream.

DESCRIPTION

If you want to know how much space to allocate for the returned encoding, do not specify the **buflen** and **bufaddr** arguments.

Format of Binary Instructions

The format of binary instructions is as follows:

Op code 16 bits	Length 16 bits	Arguments

ZK-5472-86

If the length field exceeds 32,767 bytes, an extended format is used. The length field should be set to UIS$C_LENGTH_DIFF and the extra length field should be set to the total number of bytes in the binary instruction.

Op code 16 bits	Length 16 bits	Extra Length 32 bits	Arguments

ZK-5473-86

UIS$EXTRACT_TRAILER

Returns trailer information needed to create a UIS metafile.

FORMAT

UIS$EXTRACT_TRAILER *vd_id [,buflen, bufaddr] [,retlen]*

RETURNS

UIS$EXTRACT_TRAILER signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

buflen

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Buffer length. The **buflen** argument is the address of a longword that defines the length of the buffer.

bufaddr

VMS Usage:	**vector_byte_unsigned**
type:	**byte integer (unsigned)**
access:	**read only**
mechanism:	**by reference**

Buffer address. The **bufaddr** argument is the address of an array of bytes that receive the binary encoded stream.

retlen

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Return length. The **retlen** argument is the address of a longword that defines the returned length of the buffer.

DESCRIPTION

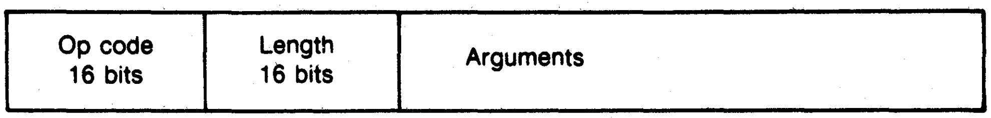
Trailer information must appear at the end of all UIS metafiles.

Allocating Space for the Buffer

If you want to know how much space to allocate for the buffer, specify **obj_id** and **retlen** only.

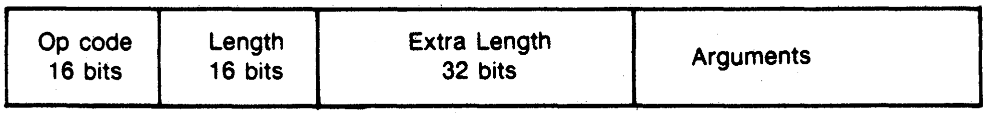
Format of Trailer Information

The format of trailer binary instructions is as follows:



ZK-5472-86

If the length field exceeds 32,767 bytes, an extended format is used. The length field should be set to UIS$C_LENGTH_DIFF and the extra length field should be set to the total number of bytes in the binary instruction.



ZK-5473-86

UIS$FIND_PRIMITIVE

Locates the next output primitive that intersects the specified rectangle.

FORMAT

obj_id = **UIS$FIND_PRIMITIVE** *vd_id, x_1,y_1, x_2,y_2 [,context] [,extent]*

RETURNS

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the object identifier in the variable *obj_id* or R0 (VAX MACRO). The object identifier uniquely identifies the object and is used as an argument in other routines.

UIS$FIND_PRIMITIVE signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

x_1, y_1
x_2, y_2

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

World coordinates of the selection rectangle. The x_1,y_1 and x_2,y_2 are the addresses of f_floating points numbers that define the lower-left and upper-right corners of the rectangle.

context

VMS Usage:	**context**
type:	**longword (signed)**
access:	**modify**
mechanism:	**by reference**

Context value. The **context** argument is the address of a longword that stores the state of the search and should not be modified if repetitive searches are desired. If this argument is omitted, only the first match can be found in the display list.

You must initialize the **context** argument to 0 before starting a search operation.

extent

VMS Usage:	**vector_longword_unsigned**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by reference**

Address of the extent rectangle array. The **extent** argument is an array of four longwords that receives the world coordinate values of the lower-left and upper-right corner of the extent rectangle.

DESCRIPTION

When you try to locate the specified object closest to the specified location, the size of the rectangle controls the object or primitive matching granularity. Normally, when you search for the primitive nearest a position, the rectangle would surround the position, and have a small width and height (perhaps equivalent to 1 to 10 pixels), depending on the desired granularity.

Once the primitive is located, it returns an object identifier which can be used later to reference the primitive, for example, UIS$EXTRACT_OBJECT or UIS$DELETE_OBJECT.

Each time UIS$FIND_PRIMITIVE is called, it continues the search operation from where it left off, using the context longword to keep track of the current state.

Generally, in order to find all matches, UIS$FIND_PRIMITIVE is called repeatedly with the same context longword until it returns a value of 0.

UIS$FIND_SEGMENT

Locates the next segment that contains any objects or primitives that intersect with the specified rectangle.

FORMAT

seg_id = **UIS$FIND_SEGMENT** *vd_id, x_1, y_1, x_2, y_2 [,context] [,extent]*

RETURNS

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the segment identifier in the variable *seg_id* or R0 (VAX MACRO). The segment identifier uniquely identifies the segment and is used as an argument in other routines.

UIS$FIND_SEGMENT signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

x_1, y_1
x_2, y_2

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

World coordinates of the selection rectangle. The x_1,y_1 and x_2,y_2 arguments are the addresses of f_floating point numbers that define the lower-left and upper-right corners of the rectangle.

context

VMS Usage:	**context**
type:	**longword (signed)**
access:	**modify**
mechanism:	**by reference**

Context value. The **context** argument is the address of a longword that stores the state of the search and should not be modified if repetitive searches are desired. If this argument is omitted, only the first match can be found in the display list.

You must initialize the **context** argument to 0 before starting a search operation.

extent

VMS Usage:	**vector_longword_unsigned**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by reference**

Address of the extent rectangle array. The **extent** argument is the address of an array of four longwords that receives the world coordinate pairs that define the lower-left and upper-right corners of the extent rectangle containing the segment.

DESCRIPTION

The size of the rectangle controls the matching granularity when trying to locate the primitive closest to a specific position. Normally, when searching for the primitive nearest a position, the rectangle would surround the position, and have a small width and height (perhaps equivalent to 1 to 10 pixels), depending on the desired granularity.

Once the object is located, UIS$FIND_SEGMENT returns the object identifier for the segment containing that object.

Each time this routine is called, it continues the search operation from where it left off, using the context longword to keep track of the search state.

Generally, in order to find all matches, UIS$FIND_SEGMENT is called repeatedly with the same context longword until it returns a value of 0.

UIS$GET_ABS_POINTER_POS

Returns the current pointer position relative to the lower-left corner of the workstation screen.

FORMAT

UIS$GET_ABS_POINTER_POS *devnam, retx, rety*

RETURNS

UIS$GET_ABS_POINTER_POS signals all errors; no condition values are returned.

ARGUMENTS

devnam

See Section 18.3.9 for more information about this argument.

retx
rety

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Absolute device coordinate pair. The **retx** and **rety** arguments are the addresses of f_floating point longwords that receive the *x* and *y* coordinate positions of the pointer in centimeters relative to the lower-left corner of the display screen.

UIS$GET_ALIGNED_POSITION

Returns the current position for text output which is the upper-left corner of the character cell.

FORMAT

UIS$GET_ALIGNED_POSITION *vd_id, atb, retx, rety*

RETURNS

UIS$GET_ALIGNED_POSITION signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block. The **atb** argument is the address of a longword integer that identifies an attribute block that contains the font to use in calculating the aligned position.

retx
rety

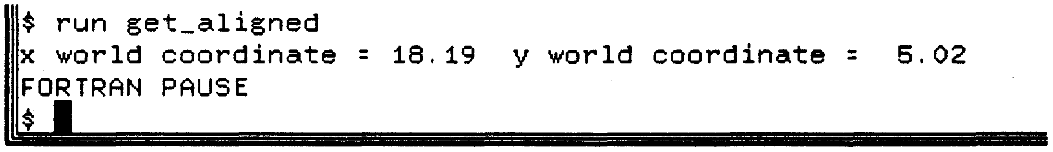
VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

World coordinate pair. The **retx** and **rety** arguments are the addresses of f_floating point longwords that receive the current position as x and y world coordinate positions.

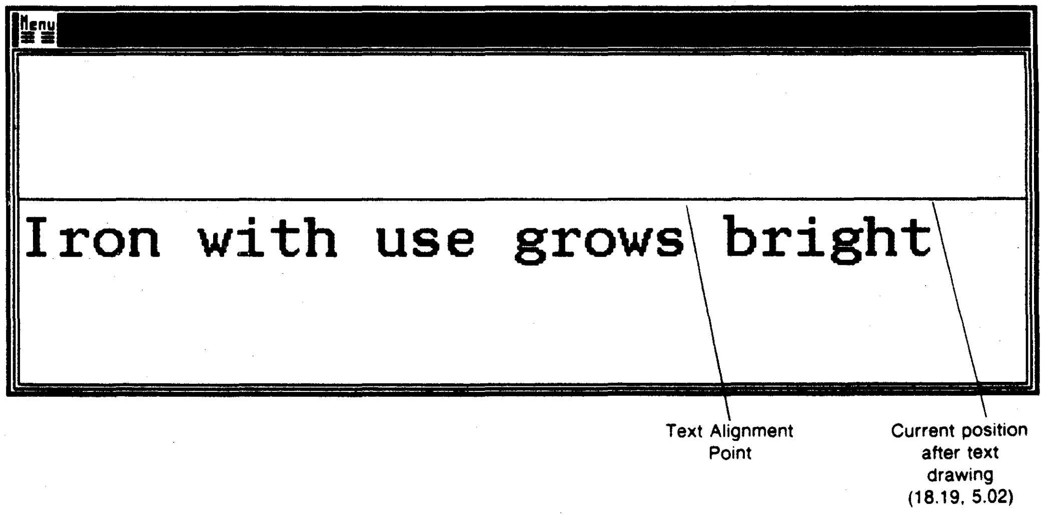
DESCRIPTION

UIS$GET_ALIGNED_POSITION differs from UIS$GET_POSITION in that the current position refers to the upper-left corner of the character cell of the next character to be output. This is useful for applications that require the position of the upper-left corner, but do not know enough about the font baseline to determine the proper alignment point. The position is converted into the proper alignment point using the font specified in the given attribute block. See UIS$SET_ALIGNED_POSITION.

screen output



```
$ run get_aligned
x world coordinate = 18.19  y world coordinate =  5.02
FORTRAN PAUSE
$
```


Menu
Iron with use grows bright
Text Alignment Point
Current position after text drawing (18.19, 5.02)
ZK 5293-86

UIS$GET_ARC_TYPE

Returns the current arc type attribute code. See UIS$SET_ARC_TYPE for more information about arc types.

FORMAT

arc_type = **UIS$GET_ARC_TYPE** *vd_id, atb*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the current arc type code in the variable *arc_type*. The arc type code is an integer value representing one of the following UIS constants: UISC_ARC_OPEN, UISC_ARC_PIE, and UIS$C_ARC_CHORD. See UIS$SET_ARC_TYPE for a description of the constants.

UIS$GET_ARC_TYPE signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block identifier. The **atb** argument is the address of a longword integer that identifies the attribute block from which the arc type is obtained.

DESCRIPTION

Refer to 6.6 for more information about UIS symbols and symbol definition files.

UIS$GET_BACKGROUND_INDEX

Returns the background color index for text and graphics output.

FORMAT

index = **UIS$GET_BACKGROUND_INDEX** *vd_id, atb*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the color map index in the variable *index* or R0 (VAX MACRO).

UIS$GET_BACKGROUND_INDEX signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword integer that identifies the attribute block from which the background color index is obtained.

UIS$GET_BUTTONS

Returns the current state of the pointer buttons.

FORMAT

status = **UIS$GET_BUTTONS** *wd_id, retstate*

RETURNS

VMS Usage:	**Boolean**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Boolean value is returned in the variable *status* or R0 (VAX MACRO). A value of *1* is returned, if the pointer is within the visible portion of the viewport. If the pointer is outside the visible portion of the viewport, a value of *0* is returned.

UIS$GET_BUTTONS signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for a description of this argument.

retstate

VMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by reference**

State of the pointer buttons. The **retstate** argument is the address of a longword that receives the current state of the pointer buttons. The state of pointer buttons is returned in a longword whose bits indicate the state of each pointer button, for example, *1* is up and *0* is down. The symbolic definitions for these bits are UIS$M_POINTER_BUTTON_1, and UIS$M_POINTER_BUTTON_2, UIS$M_POINTER_BUTTON_3, and UIS$M_POINTER_BUTTON_4.

DESCRIPTION

When you use this function always test the returned status value, because the pointer could be outside the window when the function is called.

UIS$GET_CHAR_ROTATION

Returns the angle of character rotation in degrees.

FORMAT

angle = **UIS$GET_CHAR_ROTATION** *vd_id, atb*

RETURNS

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by value**

Longword value returned as the angle of character rotation in degrees in the variable *angle* or R0 (VAX MACRO). The baseline vector and the actual path of text drawing form the angle of character rotation. The character rotates on its baseline point.

UIS$GET_CHAR_ROTATION signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a number that identifies an attribute block containing the character rotation attribute used to calculate character rotation.

UIS$GET_CHAR_SIZE

Returns both a value indicating whether or not character scaling is enabled and the character size used.

FORMAT

Boolean = **UIS$GET_CHAR_SIZE** *vd_id, atb ,[char],[width][,height*

RETURNS

VMS Usage:	**Boolean**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Boolean value returned to indicate the status of character scaling in a status variable or R0 (VAX MACRO).

UIS$GET_CHAR_SIZE signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword that identifies an attribute block that contains the character size attribute setting.

char

VMS Usage:	**char_string**
type:	**character_string**
access:	**write only**
mechanism:	**by descriptor**

Single character. The **char** argument is the address of a character string descriptor of a single char. The **char** is specified only for proportionally spaced fonts. It is used as a reference point against which other characters are scaled.

width
height

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Character width and height. The **width** argument is the address of an f_floating point longword that receives the character width in world coordinates. The **height** argument is the address of an f_floating point longword that receives the character height in world coordinates.

UIS$GET_CHAR_SLANT

Returns the angle of character slant in degrees.

FORMAT

angle = **UIS$GET_CHAR_SLANT** *vd_id, atb*

RETURNS

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by value**

Longword value returned as the angle of character slant in degrees in the variable *angle* or R0 (VAX MACRO). The character cell up vector and the baseline vector form the angle of character slant.

UIS$GET_CHAR_SLANT signals all errors; no condition values are returned.

ARGUMENTS

vd_id

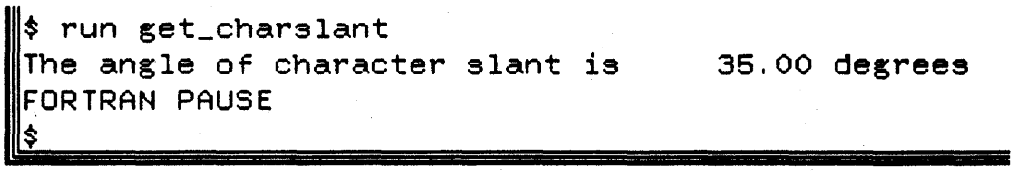
See Section 18.3.1 for a description of this argument.

atb

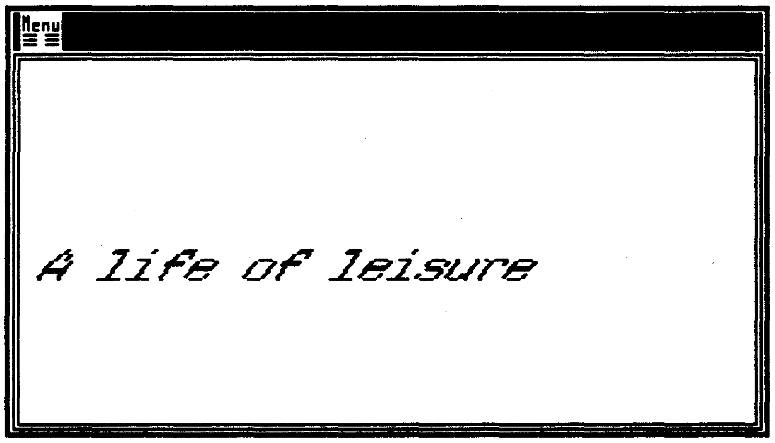
VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a number that identifies an attribute block containing the character slant attribute setting to be returned.

screen output



```
$ run get_charslant
The angle of character slant is     35.00 degrees
FORTRAN PAUSE
$
```



UIS$GET_CHAR_SPACING

Returns the character spacing factors.

FORMAT **UIS$GET_CHAR_SPACING** *vd_id, atb, dx, dy*

RETURNS UIS$GET_CHAR_SPACING signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

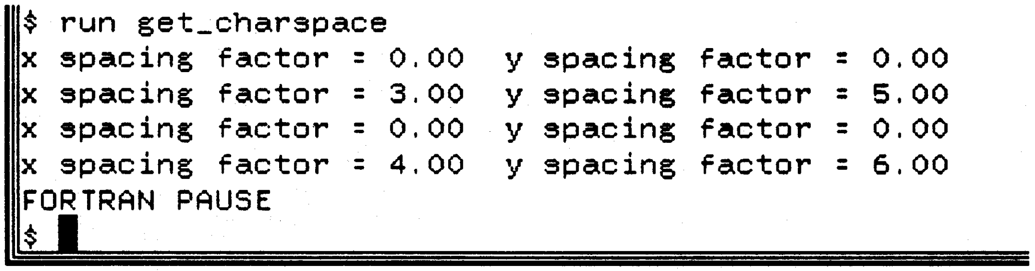
Attribute block number. The **atb** argument is the address of a longword integer that identifies the attribute block from which the character spacing factors are obtained.

dx
dy

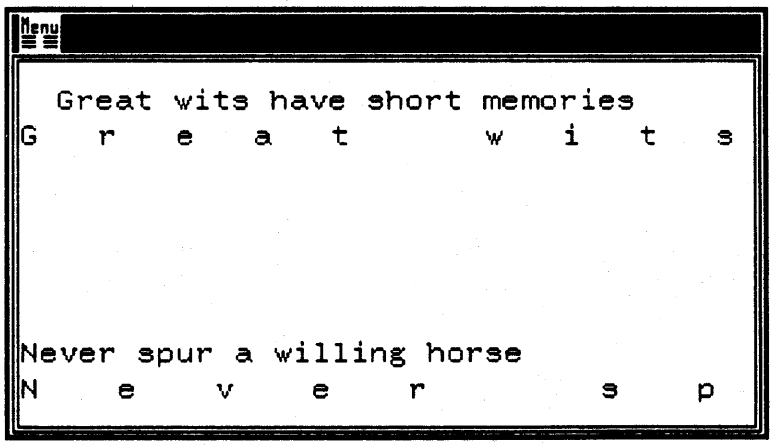
VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Additional *x* and *y* spacing factor. The **dx** argument is the address of an f_floating point longword that receives the x spacing factor. The *x* spacing factor represents the relative width of the character cell. If *0* is returned, no additional spacing factor was specified. The **dy** argument is the address of an f_floating point longword that receives the y spacing factor. The *y* spacing factor represents the relative height of the character cell. If *0* is returned, no additional spacing factor was specified.

screen output



```
$ run get_charspace
x spacing factor = 0.00  y spacing factor = 0.00
x spacing factor = 3.00  y spacing factor = 5.00
x spacing factor = 0.00  y spacing factor = 0.00
x spacing factor = 4.00  y spacing factor = 6.00
FORTRAN PAUSE
$
```



```
  Great wits have short memories
G   r   e   a   t       w   i   t   s




Never spur a willing horse
N    e    v    e    r         s    p
```

ZK-5291-86

UIS$GET_CLIP

Returns the clipping mode.

FORMAT

status = **UIS$GET_CLIP** *vd_id, atb [,x_1, y_1, x_2, y_2]*

RETURNS

VMS Usage:	**Boolean**
type:	**longword**
access:	**write only**
mechanism:	**by value**

Boolean value returned as the clipping mode in a status variable or R0 (VAX MACRO). If clipping is enabled, a Boolean TRUE is returned. If clipping is disabled, a Boolean FALSE is returned.

UIS$GET_CLIP signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

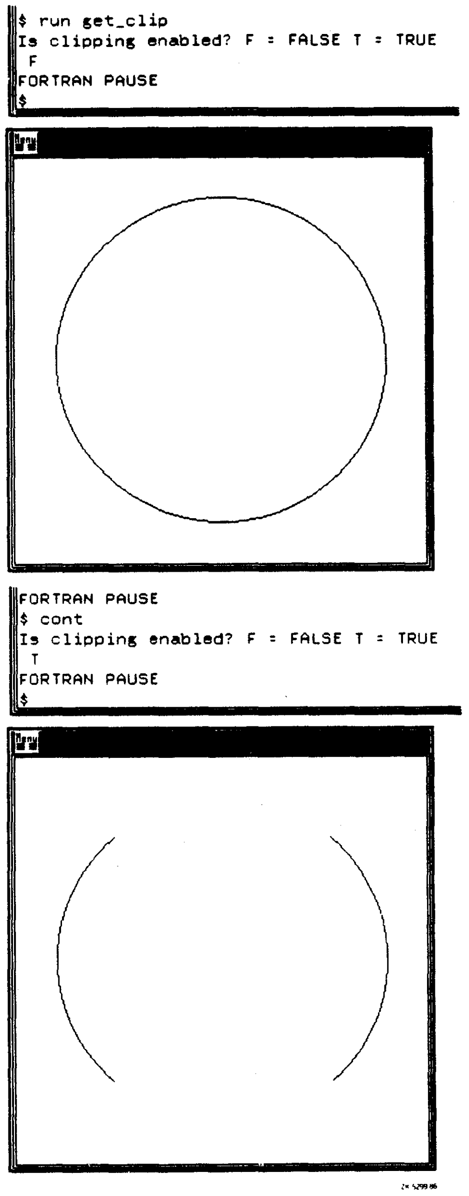
Attribute block number. The **atb** argument is the address of a longword integer that identifies the attribute block from which the clipping rectangle and mode are obtained.

x_1, y_1
x_2, y_2

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

World coordinate pair. The x_1 and y_1 arguments are addresses of f_floating point longwords that receive the coordinates of the lower-left corner of the world coordinate clipping rectangle. The x_2 and y_2 arguments are the addresses of f_floating point longwords that receive the coordinates of the upper-right corner of the world coordinate clipping rectangle.

screen output



```
$ run get_clip
Is clipping enabled? F = FALSE T = TRUE
 F
FORTRAN PAUSE
$
```

```
FORTRAN PAUSE
$ cont
Is clipping enabled? F = FALSE T = TRUE
 T
FORTRAN PAUSE
$
```

UIS$GET_COLOR

Returns a single red green blue (RGB) color value associated with an entry in a virtual color map.

FORMAT **UIS$GET_COLOR** *vd_id, index, retr, retg, retb [,wd_id]*

RETURNS UIS$GET_COLOR signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

index

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Virtual color map index. The **index** argument is the address of a longword that specifies the index of the virtual color map entry to be returned.

retr

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Red value. The **retr** argument is the address of an f_floating point longword that receives the red value. The red value is in the range of *0.0* to *1.0,* inclusive.

retg

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Green value. The **retg** argument is the address of an f_floating point longword that receives the green value. The green value is in the range of *0.0* to *1.0,* inclusive.

retb

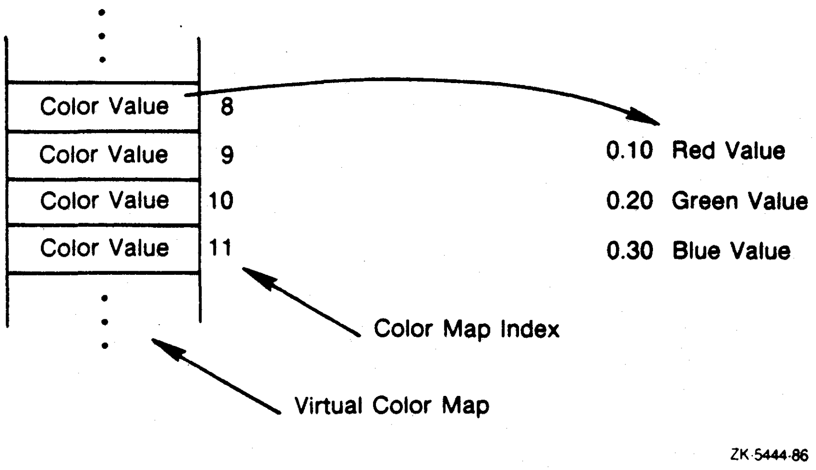
VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Blue value. The **retb** argument is the address of an f_floating point longword that receives the blue value. The blue value is in the range of *0.0* to *1.0*, inclusive.

wd_id

See Section 18.3.2 for a description of this argument.

illustration



UIS$GET_COLORS

Returns red, green, and blue (RGB) color values associated with one or more entries in the virtual color map.

FORMAT

UIS$GET_COLORS *vd_id, index, count, retr_vector, retg_vector, retb_vector [,wd_id]*

RETURNS

UIS$GET_COLORS signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

index

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Starting color map index. The **index** argument is the address of a longword that specifies the index of the first color map entry to be returned.

If the specified index exceeds the maximum index for the virtual color map, an error is signaled.

count

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Number of virtual color map indices. The **count** argument is the address of a longword that defines the total number of color map entries in the virtual color map to be returned including the starting index.

If the total number of indices exceeds the maximum number of indices in the virtual color, an error is signaled.

retr_vector

VMS Usage:	**vector_longword_signed**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Red values. The **retr_vector** argument is the address of an array of f_floating point longwords that receives the red color values. Each red value is in the range of *0.0* to *1.0*, inclusive.

retg_vector

VMS Usage: **vector_longword_signed**
type: **f_floating**
access: **write only**
mechanism: **by reference**

Green values. The **retg_vector** argument is the address of an array of f_floating point longwords that receives the green color values. Each green value is in the range of *0.0* to *1.0*, inclusive.

retb_vector

VMS Usage: **vector_longword_signed**
type: **f_floating**
access: **write only**
mechanism: **by reference**

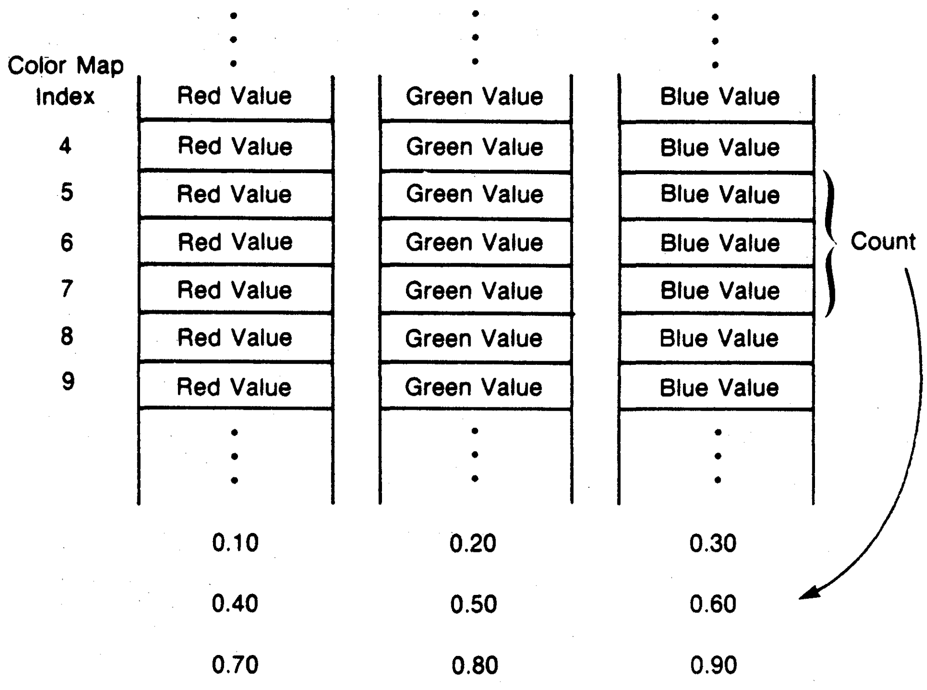
Blue values. The **retb_vector** argument is the address of an array of f_floating point longwords that receives the blue color values. Each blue value is in the range of *0.0* to *1.0*, inclusive.

wd_id

See Section 18.3.2 for a description of this argument.

If the **wd_id** argument is not specified, the red, green, and blue color values returned are the *set* color values originally established by UIS$SET_COLOR or UIS$SET_COLORS.

illustration



ZK 5365-86

UIS$GET_CURRENT_OBJECT

Returns the identifier of the last object drawn in the virtual display and added to the display list.

FORMAT

current_id = **UIS$GET_CURRENT_OBJECT** *vd_id*

RETURNS

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the identifier of the current object in the variable *current_id* or R0 (VAX MACRO).

UIS$GET_CURRENT_OBJECT signals all errors; no condition values are returned.

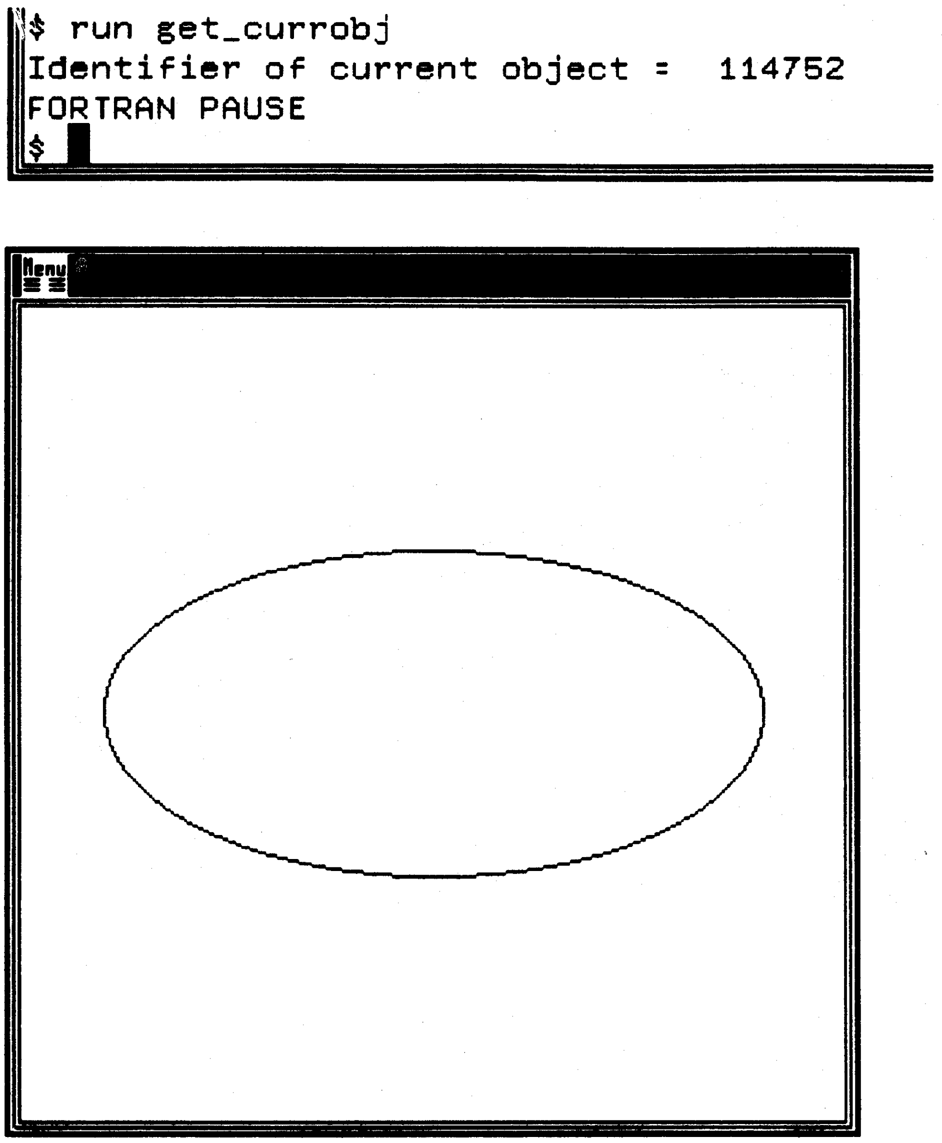
ARGUMENT

vd_id
See Section 18.3.1 for a description of this argument.

DESCRIPTION

If there are no objects in the display list, the root segment identifier is returned. If UIS$GET_CURRENT_OBJECT is called after a call to UIS$SET_INSERTION_POSITION, the returned identifier is based on the current insertion position in the segment.

screen output



```
$ run get_currobj
Identifier of current object =  114752
FORTRAN PAUSE
$
```

ZK-5397-86

UIS$GET_DISPLAY_SIZE

Obtains the dimensions of the workstation display screen.

FORMAT

UIS$GET_DISPLAY_SIZE *devnam, retwidth, retheight [,retresolx, retresoly] [,retpwidth, retpheight]*

RETURNS

UIS$GET_DISPLAY_SIZE signals all errors; no condition values are returned.

ARGUMENTS

devnam
See Section 18.3.9 for more information about this argument.

retwidth
retheight

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

VAXstation display screen size. The **retwidth** and **retheight** arguments are the addresses of f_floating point longwords that receive the physical display screen width and height in centimeters.

retresolx
retresoly

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

VAXstation display screen resolution. The **retresolx** and **retresoly** arguments are the addresses of f_floating point longwords that receive the *x* and *y* resolution in pixels per centimeters.

retpwidth
retpheight

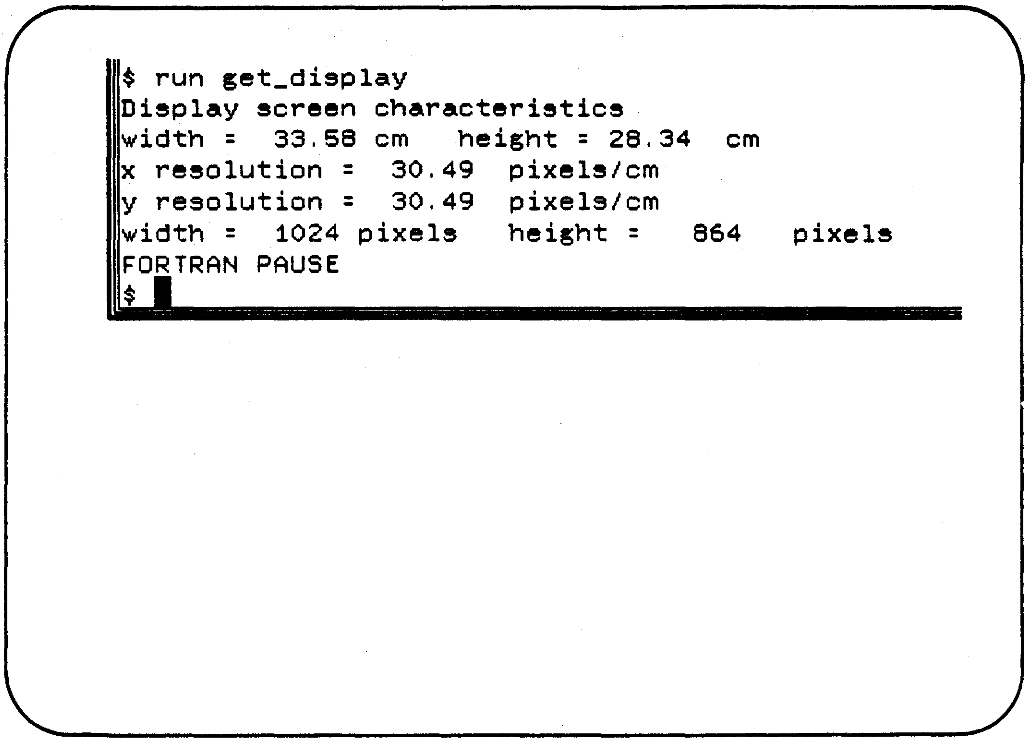
VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

VAXstation screen size in pixels. The **retpwidth** and **retpheight** arguments are the addresses of integer longwords that receive the width and height of the screen in pixels.

DESCRIPTION

Use height and width dimensions to determine the size of a virtual display or viewport. Use resolution values when it is important for the application to determine the exact physical size (or world coordinate dimensions) that map to a single pixel.

screen output



```
$ run get_display
Display screen characteristics
width =   33.58 cm   height = 28.34  cm
x resolution =  30.49  pixels/cm
y resolution =  30.49  pixels/cm
width =  1024 pixels   height =   864   pixels
FORTRAN PAUSE
$ █
```

ZK-5449-86

UIS$GET_FILL_PATTERN

Returns the index of the fill pattern.

FORMAT

status = **UIS$GET_FILL_PATTERN** *vd_id, atb [,index]*

RETURNS

VMS Usage:	**Boolean**
type:	**longword**
access:	**write only**
mechanism:	**by value**

Boolean value returned as the filling mode in a status variable or R0 (VAX MACRO). The Boolean TRUE is returned if filling is enabled, otherwise the Boolean value is FALSE.

UIS$GET_FILL_PATTERN signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

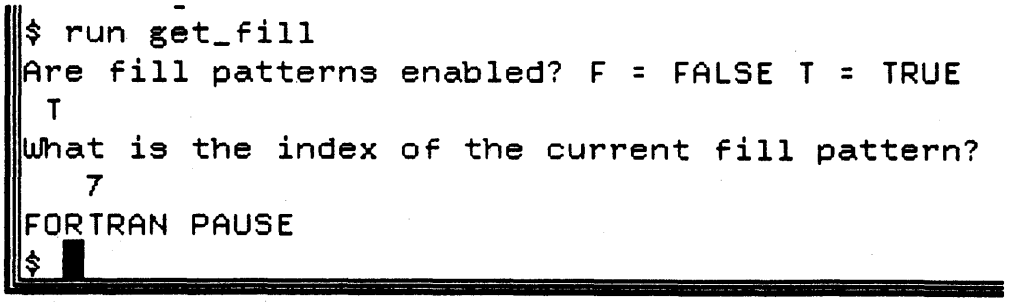
Attribute block number. The **atb** argument is the address of a longword integer that identifies the attribute block from which the fill pattern index is obtained.

index

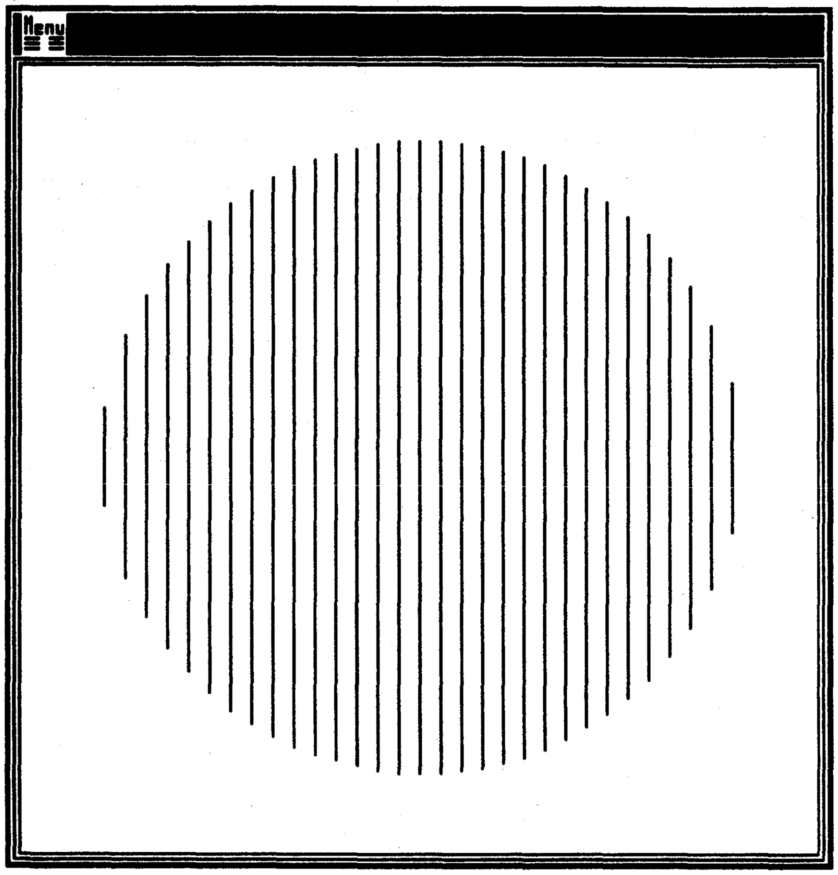
VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Index of the fill pattern. The **index** argument is the address of a longword that receives the value of the fill pattern symbol index. This is the index of a glyph in a fill pattern font.

screen output



```
$ run get_fill
Are fill patterns enabled? F = FALSE T = TRUE
 T
What is the index of the current fill pattern?
    7
FORTRAN PAUSE
$
```



ZK-5391-86

UIS$GET_FONT

Returns the name of font file.

FORMAT

UIS$GET_FONT *vd_id, atb, bufferdesc [,length]*

RETURNS

UIS$GET_FONT signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword integer that identifies the attribute block from which the font file name is obtained.

bufferdesc

VMS Usage:	**char_string**
type:	**character string**
access:	**write only**
mechanism:	**by descriptor**

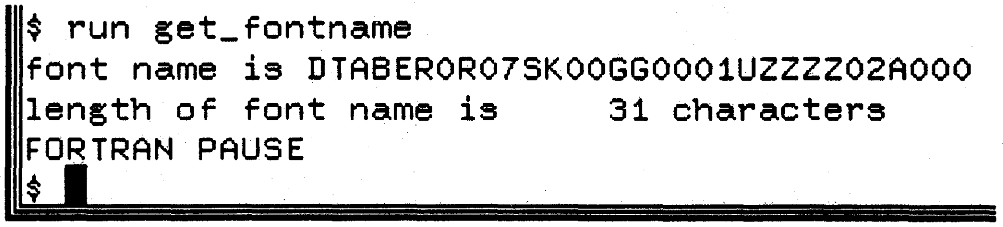
Font file name string. The **bufferdesc** argument is the address of a character string descriptor of a location that receives the font file name character string.

length

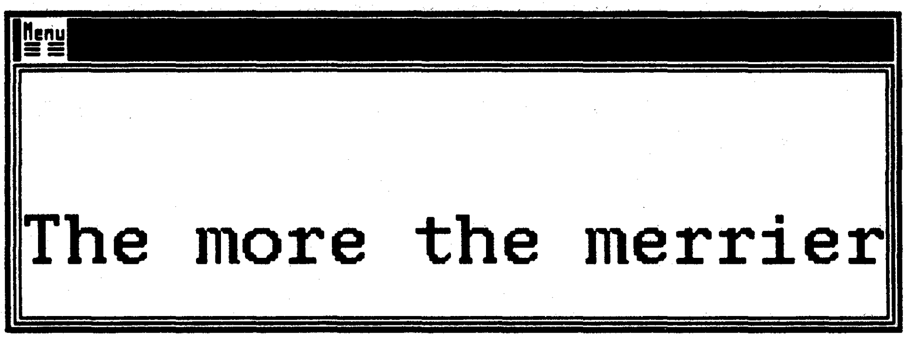
VMS Usage:	**word_signed**
type:	**word (signed)**
access:	**write only**
mechanism:	**by reference**

Length of the font file character string. The **length** argument is the address of a word that receives the length of font file name character string.

screen output



```
$ run get_fontname
font name is DTABER0R07SK00GG0001UZZZZ02A000
length of font name is     31 characters
FORTRAN PAUSE
$
```



The more the merrier

ZK-5392-86

UIS$GET_FONT_ATTRIBUTES

Returns information about the ascender, descender, height, width, and font parameters.

FORMAT

UIS$GET_FONT_ATTRIBUTES *font_id, ascender, descender, height [,maximum_width] [item_list]*

RETURNS

UIS$GET_FONT_ATTRIBUTES signals all errors; no condition values are returned.

ARGUMENTS

font_id

VMS Usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

Font file name. The **font_id** argument is the address of a string descriptor of the font file name only. UIS searches the directory SYS$FONT for the correct file type.

ascender
descender

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Character ascender and descender. The **ascender** argument is the address of a longword that receives the distance between the font baseline and the top of the character cell in pixels. The **descender** argument is the address of a longword that receives the distance between the font baseline and the bottom of the character cell in pixels.

height

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by reference**

Height of the character cell. The **height** argument is the address of a longword that receives the height of the character cell in pixels.

maximum_width

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

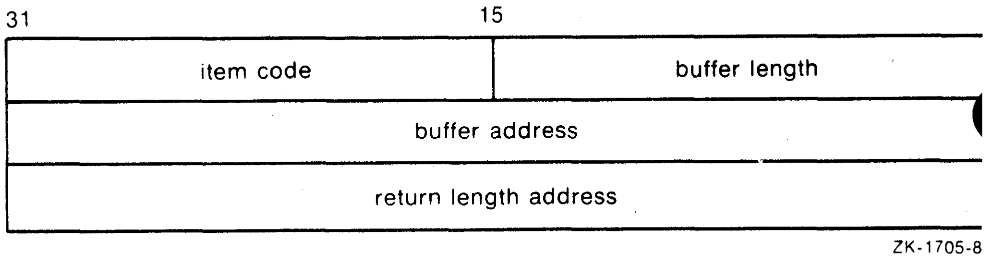
Maximum width of a character cell. The **maximum_width** argument is the address of a longword that receives the maximum width of a character cell in the font in pixels.

item_list

VMS Usage: **item_list_3**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Item list specifying additional font information to be returned. The **item_list** argument is the address of a list of item descriptors, each of which describes an item of information. A longword value of 0 terminates the list of item descriptors.

The structure of the item list is described in the following figure.



The following table lists valid item codes.

Item Code	Information Returned
Character Information	
UIS$C_FNT_FIRST_CHAR	First character in the font
UIS$C_FNT_LAST_CHAR	Last defined character in the font
UIS$C_FNT_GUTPERPIX_X	x resolution of the font in gutenbergs per pixel
UIS$C_FNT_GUTPERPIX_Y	y resolution of the font in gutenbergs per pixel
UIS$C_FNT_AVERAGE_GUT[1]	Average width of a character in the font
UIS$C_FNT_WIDTH	Width in pixels of all glyphs in the font, if the font is monospaced. A zero is returned, if the font is proportionally spaced.
Font Flags[2]	

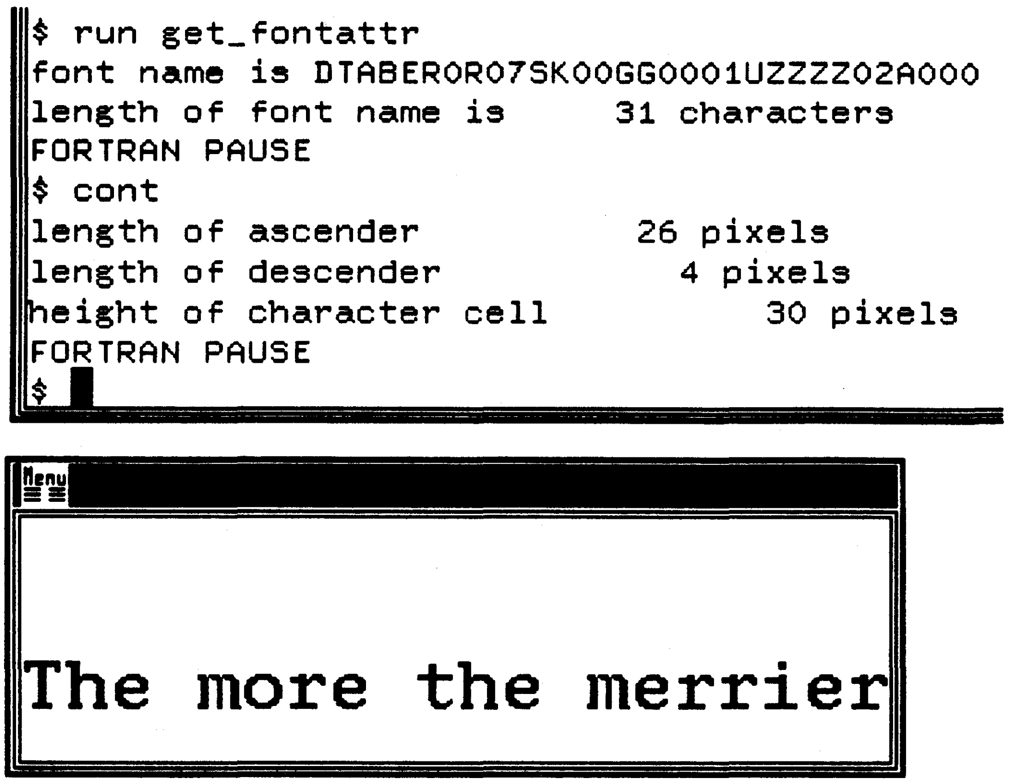
[1]The font designer assigns this number. Although, the graphics subsystem copies the number, no interpretation is applied to it. UIS does not use the number.

[2]The value 1 is returned, if TRUE, and 0, if FALSE.

Item Code	Information Returned
Font Flags[2]	
UIS$C_FNT_FIXED	True, if the font is monospaced False, if the font is proportionally spaced.
UIS$C_FNT_CELLEQRAST	True, if the cell width of all glyphs in the font equals the width the glyph's raster.
UIS$C_FNT_VA_FONT	True, if this is a VA font.
Font Name	
UIS$C_FNT_FONT_ID	Font identifier string

[2]The value 1 is returned, if TRUE, and 0, if FALSE.

screen output



```
$ run get_fontattr
font name is DTABER0R07SK00GG0001UZZZZ02A000
length of font name is      31 characters
FORTRAN PAUSE
$ cont
length of ascender           26 pixels
length of descender            4 pixels
height of character cell           30 pixels
FORTRAN PAUSE
$
```

```
The more the merrier
```

ZK-5282-86

UIS$GET_FONT_SIZE

Obtains the size of a character or string of characters in the specified font in physical dimensions.

FORMAT

UIS$GET_FONT_SIZE *fontid, text_string, retwidth, retheight*

RETURNS

UIS$GET_FONT_SIZE signals all errors; no condition values are returned.

ARGUMENTS

fontid

VMS Usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

Font identifier. The **fontid** argument is the address of a character string descriptor of a font file name. Specify only the font file name. UIS searches the directory SYS$FONT for the correct file type.

text_string

VMS Usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

Text string. The **text_string** argument is the address of a descriptor of a character or character string.

retwidth
retheight

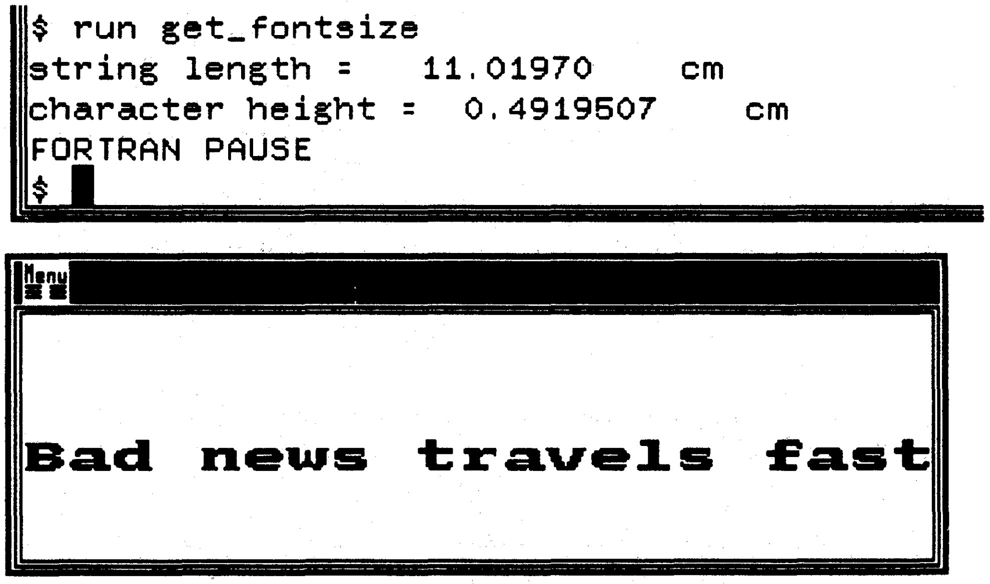
VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

String width and height. The **retwidth** and **retheight** arguments are the addresses of f_floating point longwords that receive the width and height of the character or character string in centimeters.

DESCRIPTION

UIS$GET_FONT_SIZE can be used to determine the proper size of a display viewport based on the size of the characters in a given font.

screen output



```
$ run get_fontsize
string length =   11.01970    cm
character height =  0.4919507    cm
FORTRAN PAUSE
$
```

ZK-5283-86

UIS$GET_HW_COLOR_INFO

Returns information about the hardware color map.

FORMAT

UIS$GET_HW_COLOR_INFO *devnam [,type] [,indices] [,colors] [,maps] [,rbits] [,gbits] [,bbits] [,ibits] [,res_indices] [,regen]*

RETURNS

UIS$GET_HW_COLOR_INFO signals all errors; no condition values are returned.

ARGUMENTS

devnam

See Section 18.3.9 for more information about this argument.

type

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Device type. The **type** argument is the address of a longword that receives the device type. The following table lists device type values.

Device Type	Value	Possible Colors
Monochrome	UIS$C_DEV_MONO	Black and white
Intensity	UIS$C_DEV_INTENSITY	Up to 2^{24} gray tones
Color	UIS$C_DEV_COLOR	Up to 2^{24} chromatic colors

indices

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Number of entries or simultaneous colors. The **indices** argument is the address of longword that receives the number of entries or simultaneous colors in the hardware color map.

colors

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Number of possible colors. The **colors** argument is the address of a longword that receives the number of possible colors represented in the color map. For example monochrome equals 2.

maps

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Number of hardware color maps. The **maps** argument is the address of a longword that receives the number of hardware color maps.

rbits

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Number of binary bits of precision for red. The **rbits** argument is the address of a longword that receives the number of binary bits of precision for the color red.

gbits

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Number of binary bits of precision for green. The **gbits** argument is the address of a longword that receives the number of binary bits of precision for the color green.

bbits

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Number of binary bits of precision for blue. The **bbits** argument is the address of a longword that receives the number of binary bits of precision for the color blue.

ibits

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Number of binary bits of precision for intensity. The **ibits** argument is the address of a longword that receives the number of binary bits of precision for intensity.

res_indices

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Number entries in the hardware color map reserved for special use. The **res_indices** argument is the address of a longword that receives the number entries in the hardware color map reserved for special use.

regen

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Color regeneration characteristics. The **regen** argument is the address of a longword that receives the color regeneration characteristics. The **regen** argument indicates whether the color and intensity changes affect previously drawn display objects that specified the same color index in the hardware look up table. The following symbols are valid values: UIS$C_DEV_RETRO or UIS$C_DEV_NONRETRO.

The following table summarizes regeneration characteristics of direct and mapped color systems.

System	Regeneration Characteristics
Direct color	Usually sequential
Mapped color	Usually retroactive

UIS$GET_INTENSITIES

Returns intensity values associated with one or more entries in the virtual color map.

FORMAT

UIS$GET_INTENSITIES *vd_id, index, count, reti_vector [,wd_id]*

RETURNS

UIS$GET_INTENSITIES signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

index

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Starting color map index. The **index** argument is the address of a longword that specifies the index of the first color map entry to be returned. If the specified index exceeds the maximum index of the virtual color map, an error is signaled.

count

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Number of indices. The **count** argument is the address of a longword that specifies the total number of color map entries to be returned including the starting index. If the specified count exceeds the maximum number of virtual color map entries, an error is signaled.

reti_vector

VMS Usage:	**vector_longword_signed**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

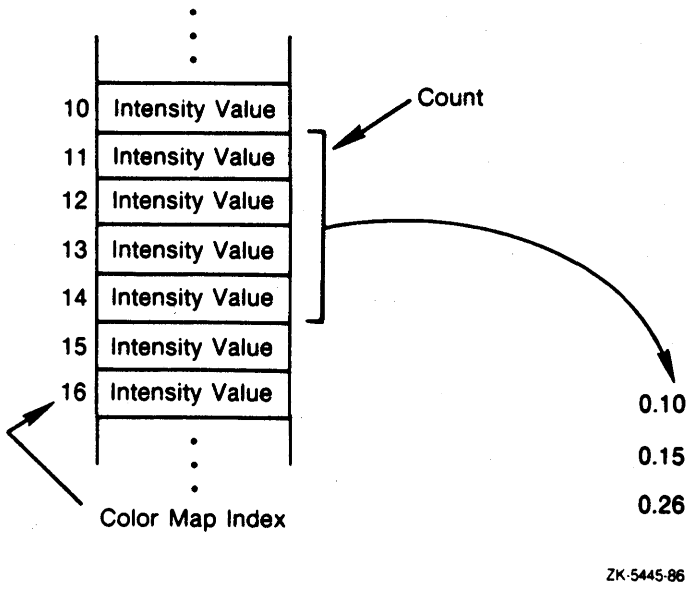
Intensity values. The **reti_vector** argument is the address of an array of f_floating point longwords that receives the intensity values. Each intensity value is in the range of *0.0* to *1.0*, inclusively.

wd_id

See Section 18.3.2 for a description of this argument.

If the **wd_id** argument is not specified, the intensity values returned are *set* color values originally established by a call to UIS$SET_INTENSITY or UIS$SET_INTENSITIES.

illustration



UIS$GET_INTENSITY

Returns the intensity value associated with a single entry in the color map.

FORMAT

UIS$GET_INTENSITY *vd_id, index, reti, [,wd_id]*

RETURNS

UIS$GET_INTENSITY signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

index

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Color map index. The **index** argument is the address of a longword integer that identifies the index of an entry in the color map associated with the virtual display. If the specified index exceeds the maximum number of indices in the virtual color map, an error is signaled.

reti

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

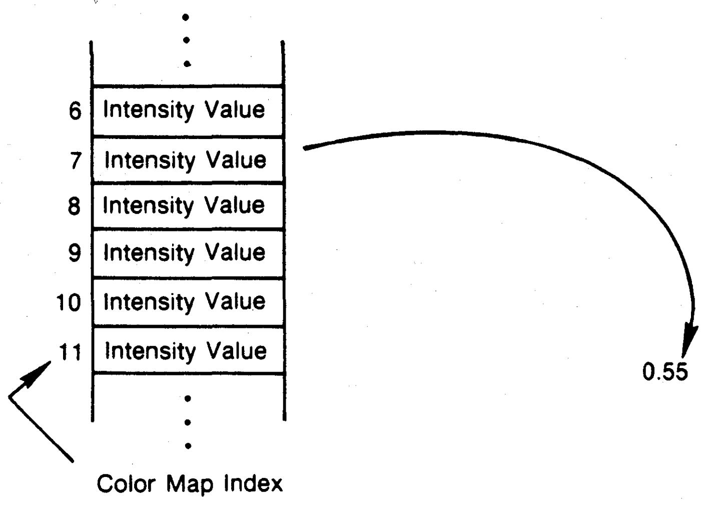
Intensity value. The **reti** argument is the address of an f_floating point longword that receives the intensity value. The intensity value is in the range of *0.0* to *1.0*, inclusive.

wd_id

See Section 18.3.2 for a description of this argument.

If the **wd_id** argument is not specified, the returned intensity values are *set* to the intensity originally established by a call to UIS$SET_INTENSITY or UIS$SET_INTENSITIES.

illustration



ZK-5446-86

UIS$GET_KB_ATTRIBUTES

Returns the virtual keyboard characteristics.

FORMAT

UIS$GET_KB_ATTRIBUTES *kb_id [,enable_items] [,disable_items][,click_vol*

RETURNS

UIS$GET_KB_ATTRIBUTES signals all errors; no condition values are returned.

ARGUMENTS

kb_id

See Section 18.3.8 for more information about the **kb_id** argument.

enable_items

VMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by reference**

Enabled keyboard characteristics. The **enable_items** argument is the address of a longword mask that receives the bit mask of the enabled keyboard characteristics.

disable_items

VMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by reference**

Disabled keyboard characteristics. The **disable_items** argument is the address of a longword mask that receives the bit mask of the disabled keyboard characteristics.

click_volume

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Key click volume level. The **click_volume** argument is the address of a longword that receives the key click volume level. The key click volume is in the range of *1* to *8*, inclusively, where *1* is quiet and *8* is loud.

DESCRIPTION

The enable and disable item lists are longword masks containing bits designating the characteristics to be enabled or disabled. The valid bits in the keyboard characteristics enable and disable masks are:

Symbol	Description[1]
UIS$M_KB_AUTORPT	Enable/disable keyboard autorepeat
UIS$M_KB_KEYCLICK	Enable/disable keyboard keyclick
UIS$M_KB_UDF6	Enable/disable up button transitions for $\boxed{\text{F6}}$ to $\boxed{\text{F10}}$ keys
UIS$M_KB_UDF11	Enable/disable up button transitions for $\boxed{\text{F11}}$ to $\boxed{\text{F14}}$ keys
UIS$M_KB_UDF17	Enable/disable up button transitions for $\boxed{\text{F17}}$ to $\boxed{\text{F20}}$ keys
UIS$M_KB_HELPDO	Enable/disable up button transitions for $\boxed{\text{HELP}}$ and $\boxed{\text{DO}}$ keys
UIS$M_KB_UDE1	Enable/disable up button transitions for $\boxed{\text{E1}}$ to $\boxed{\text{E6}}$ keys
UIS$M_KB_ARROW	Enable/disable up button transitions for arrow keys
UIS$M_KB_KEYPAD	Enable/disable up button transitions for numeric keypad keys

[1]By default down button transitions are enabled.

UIS$GET_LINE_STYLE

Returns the line style patterns.

FORMAT

style = **UIS$GET_LINE_STYLE** *vd_id, atb*

RETURNS

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the line style bit vector in the variable *style* or R0 (VAX MACRO).

UIS$GET_LINE_STYLE signals all errors; no condition values are returned.

ARGUMENTS

vd_id

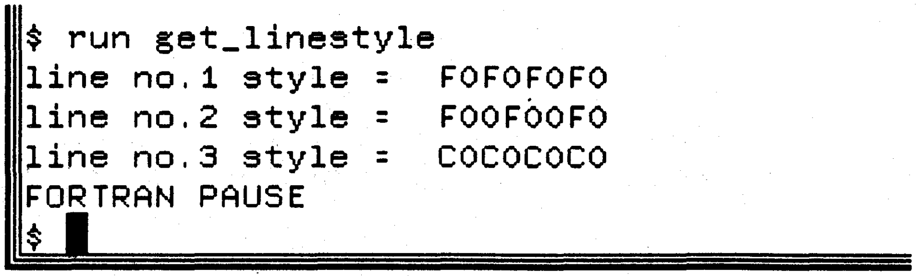
See Section 18.3.1 for a description of this argument.

atb

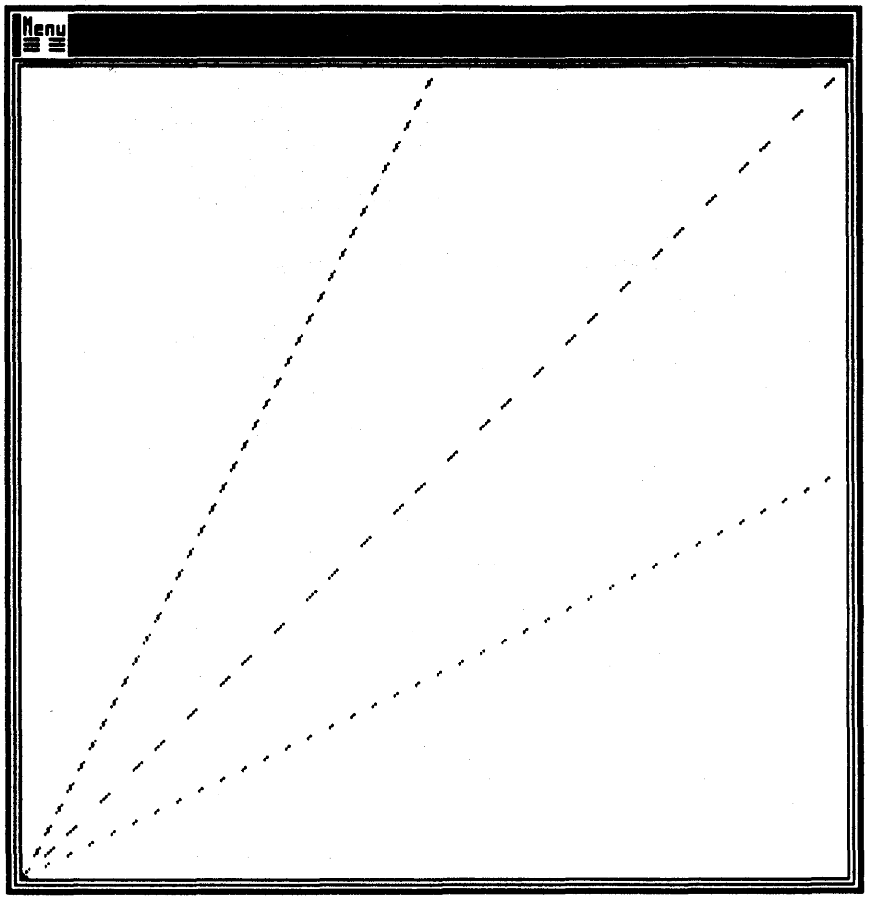
VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword integer that identifies an attribute block from which the line style pattern or bit vector is obtained.

screen output



```
$ run get_linestyle
line no.1 style =  FOFOFOFO
line no.2 style =  FOOFOOFO
line no.3 style =  COCOCOCO
FORTRAN PAUSE
$
```



ZK-5396-86

UIS$GET_LINE_WIDTH

Returns the line width.

FORMAT

width = **UIS$GET_LINE_WIDTH** *vd_id, atb [,mode]*

RETURNS

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by value**

F_floating point value returned as the line width in the variable *width* or R0 (VAX MACRO).

UIS$GET_LINE_WIDTH signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

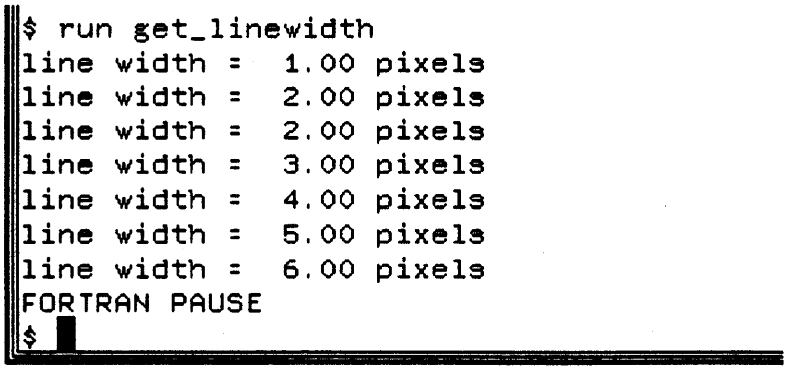
Attribute block number. The **atb** argument is the address of a longword integer that identifies the attribute block from which the line width is obtained.

mode

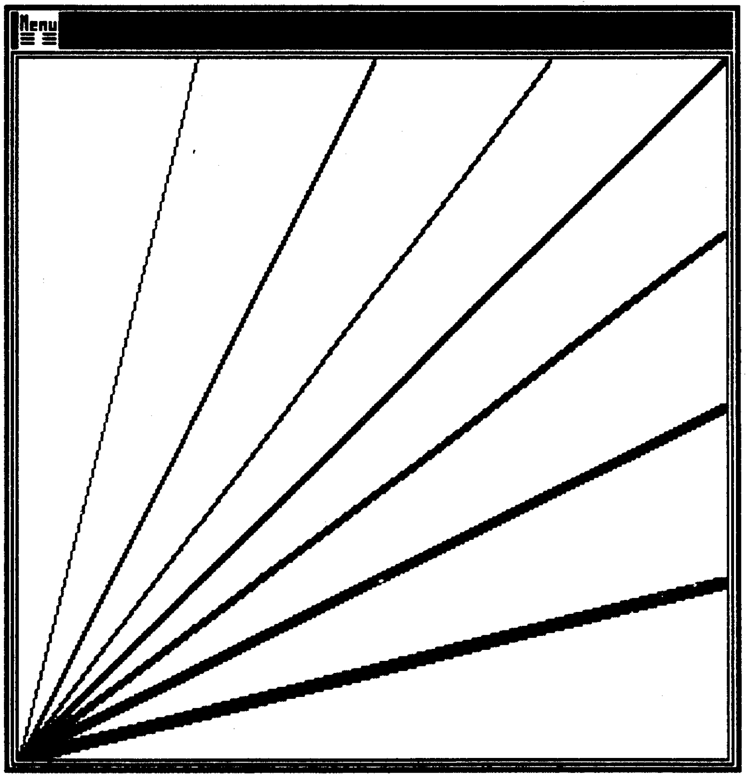
VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Line width mode. The optional **mode** argument is the address of a longword that receives the line width specification mode (UIS$C_WIDTH_WORLD or UIS$C_WIDTH_PIXELS). If UIS$C_WIDTH_WORLD is returned, the line width is interpreted as world coordinates. If UIS$C_WIDTH_PIXELS is returned, the line width is interpreted as pixels.

screen output



```
$ run get_linewidth
line width =   1.00 pixels
line width =   2.00 pixels
line width =   2.00 pixels
line width =   3.00 pixels
line width =   4.00 pixels
line width =   5.00 pixels
line width =   6.00 pixels
FORTRAN PAUSE
$ █
```



ZK-5395-86

UIS$GET_NEXT_OBJECT

Returns the identifier of the next object in the display list.

FORMAT

next_id = **UIS$GET_NEXT_OBJECT** $\left\{ \begin{array}{l} \textit{obj\_id} \\ \textit{seg\_id} \end{array} \right\}$ *[,flags]*

RETURNS

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by reference**

Longword value returned as the next object identifier in the variable *next_id* or R0 (VAX MACRO). The next object identifier uniquely identifies the next specified object in the display list and is used as an argument in other routines.

UIS$GET_NEXT_OBJECT signals all errors; no condition values are returned.

ARGUMENTS

obj_id

See Section 18.3.3 for a description of this argument.

seg_id

See Section 18.3.4 for a description of this argument.

flags

VMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Flags. The **flags** argument is the address of a longword that controls how the display list is searched. If the **flags** argument is set using UIS$M_DL_SAME_SEGMENT, the next object in the segment containing the object specified is returned.

If the **flags** argument is omitted, the next object in the display list, regardless of the segment in which it is contained, is returned.

DESCRIPTION

If a zero is returned, the next object was not found.

UIS$GET_OBJECT_ATTRIBUTES

Returns the type and extent of the specified object.

FORMAT

type = **UIS$GET_OBJECT_ATTRIBUTES** $\left\{ \begin{array}{c} \textit{obj\_id} \\ \textit{seg\_id} \end{array} \right\}$ *[,extent]*

RETURNS

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the object type in the variable *type* or R0 (VAX MACRO). An object type identifies a graphic object such as images, points lines, or ellipses, a display list structure such as a segment, or the occurrence of an event such as movement to a new text line. Possible valid objects are listed in the following table.

Symbol	Description
UIS$C_OBJECT_SEGMENT	Segment
UIS$C_OBJECT_PLOT	Point, line, connected lines, or polygon
UIS$C_OBJECT_TEXT	Characters
UIS$C_OBJECT_ELLIPSE	Elliptical or circular arcs, circles and ellipses
UIS$C_OBJECT_IMAGE	Raster image
UIS$C_OBJECT_LINE	Unconnected lines
UIS$C_OBJECT_NEW_TEXT_ LINE	New text line

UIS$GET_OBJECT_ATTRIBUTES signals all errors; no condition values are returned.

ARGUMENTS

obj_id

See Section 18.3.3 for a description of this argument.

seg_id

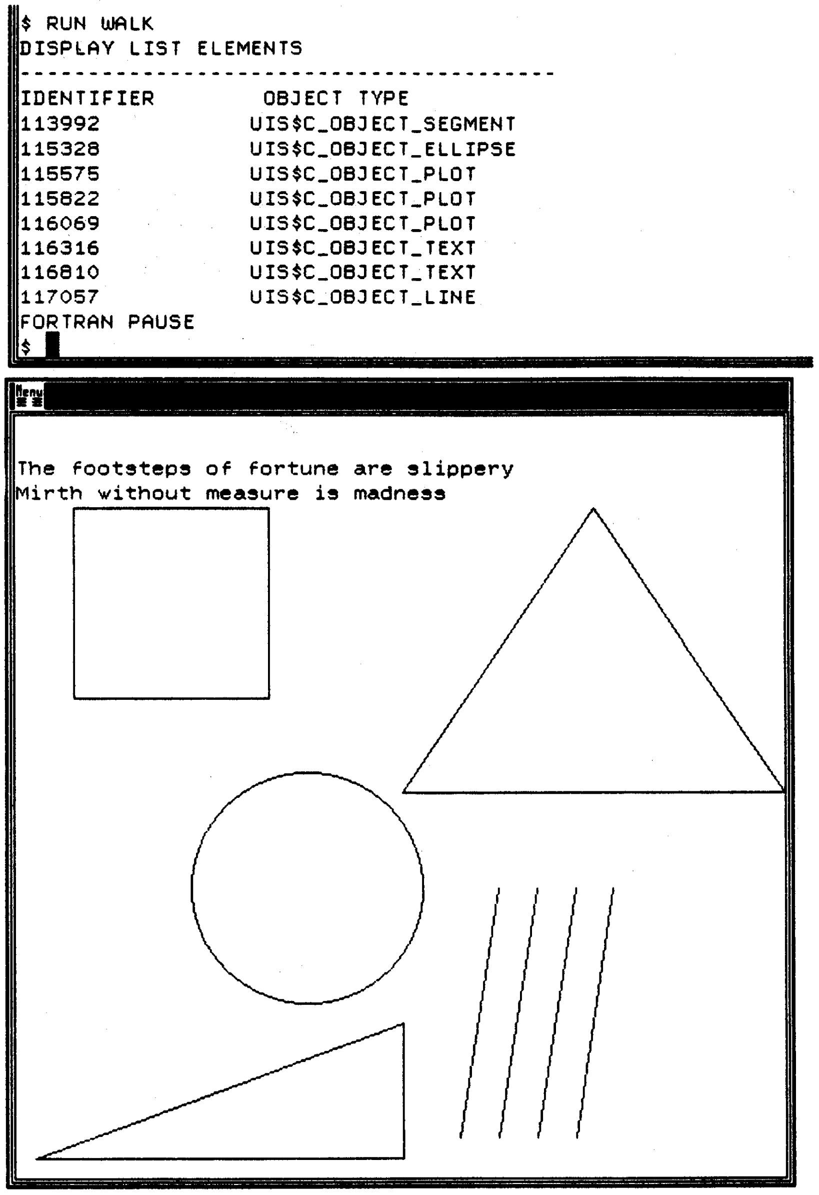
See Section 18.3.4 for a description of this argument.

extent

VMS Usage:	**vector_longword_signed**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

World coordinates of the extent rectangle. The **extent** argument is the address of an array of four longwords that receives the values of the world coordinates of the lower-left corner and the upper-right corner of the extent rectangle containing the object.

screen output



```
$ RUN WALK
DISPLAY LIST ELEMENTS
--------------------------------------
IDENTIFIER         OBJECT TYPE
113992            UIS$C_OBJECT_SEGMENT
115328            UIS$C_OBJECT_ELLIPSE
115575            UIS$C_OBJECT_PLOT
115822            UIS$C_OBJECT_PLOT
116069            UIS$C_OBJECT_PLOT
116316            UIS$C_OBJECT_TEXT
116810            UIS$C_OBJECT_TEXT
117057            UIS$C_OBJECT_LINE
FORTRAN PAUSE
$ █
```

```
The footsteps of fortune are slippery
Mirth without measure is madness
```

ZK 5300-86

UIS$GET_PARENT_SEGMENT

Returns the parent segment identifier of the specified object.

FORMAT

parent_id = **UIS$GET_PARENT_SEGMENT** $\left\{ \begin{array}{l} \textit{obj\_id} \\ \textit{seg\_id} \end{array} \right\}$

RETURNS

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the parent segment identifier in the variable *parent_id* or R0 (VAX MACRO). The parent segment identifier uniquely identifies a parent segment and is used as an argument in other routines.

UIS$GET_PARENT_SEGMENT signals all errors; no condition values are returned.

ARGUMENTS

obj_id
See Section 18.3.3 for a description of this argument.

seg_id
See Section 18.3.4 for a description of this argument.

DESCRIPTION

If the specified object is the outermost segment or root segment, its own object identifier is returned.

UIS$GET_POINTER_POSITION

Returns the current pointer position in world coordinates.

FORMAT

status = **UIS$GET_POINTER_POSITION** *vd_id, wd_id, retx, rety*

RETURNS

VMS Usage:	**Boolean**
type:	**longword**
access:	**write only**
mechanism:	**by value**

Boolean value returned as the current position of the pointer in a status variable. UIS$GET_POINTER_POSITION returns the Boolean true value *1* if the pointer is within the visible portion of the viewport, *0* is returned if the pointer is outside the visible portion of the viewport. In the latter case, the x and y values are returned as 0,0.

UIS$GET_POINTER_POSITION signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

wd_id
See Section 18.3.2 for a description of this argument.

retx
rety

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

World coordinate pair. The **retx** and **rety** arguments are the addresses of f_floating point longwords that receive the pointer x and y world coordinates.

DESCRIPTION

Note that when you use this routine you should always test the returned status value, since if the pointer is outside the window when the service is called, the x,y values would be meaningless.

UIS$GET_POSITION

Returns the current baseline position for text output.

FORMAT

UIS$GET_POSITION *vd_id, retx, rety*

RETURNS

UIS$GET_POSITION signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

retx
rety

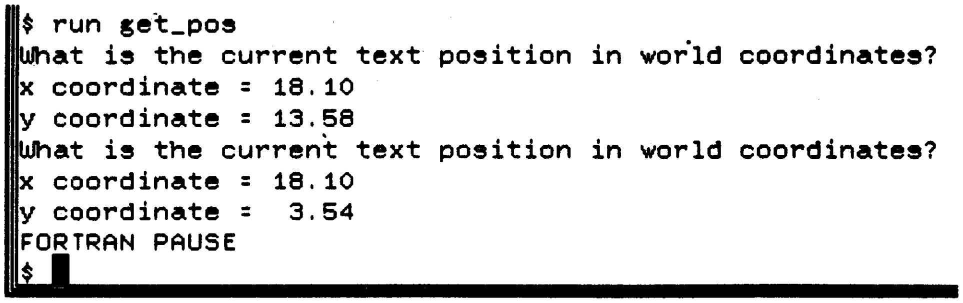
VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

World coordinate pair. The **retx** and **rety** arguments are addresses of f_floating point longwords that receive the x and y world coordinate positions.

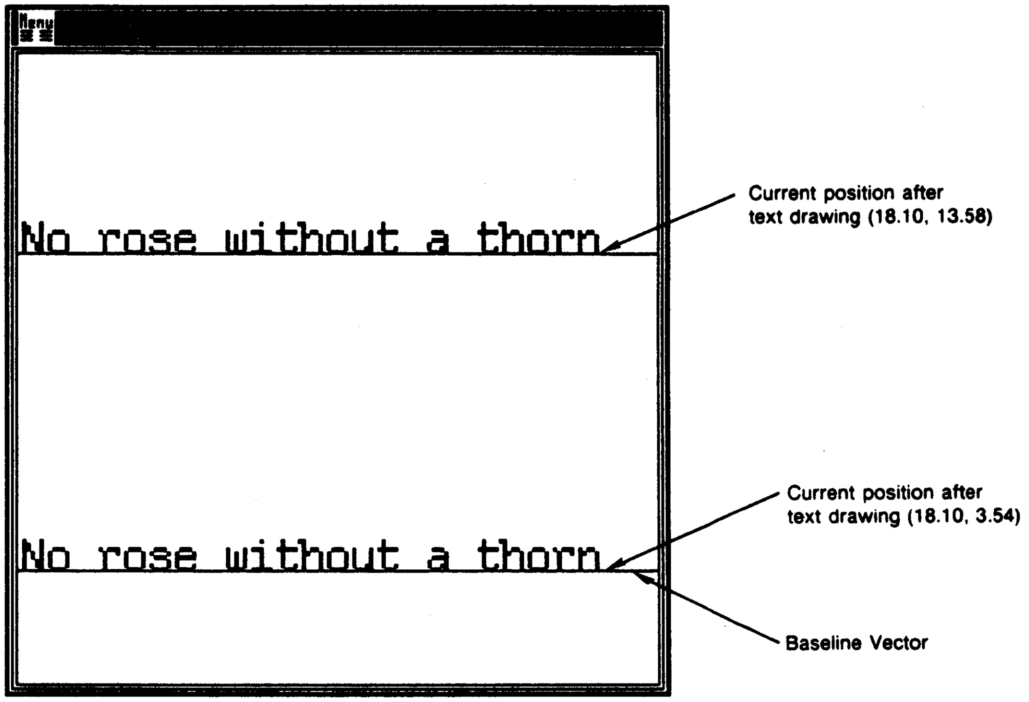
DESCRIPTION

UIS$TEXT and UIS$NEW_TEXT_LINE recognize the concept of current position. The position refers to the alignment point on the baseline of the next character to be output.

screen output



```
$ run get_pos
What is the current text position in world coordinates?
x coordinate =  18.10
y coordinate =  13.58
What is the current text position in world coordinates?
x coordinate =  18.10
y coordinate =   3.54
FORTRAN PAUSE
$
```


Menu
No rose without a thorn
Current position after text drawing (18.10, 13.58)
No rose without a thorn
Current position after text drawing (18.10, 3.54)
Baseline Vector

ZK-5413-86

UIS$GET_PREVIOUS_OBJECT

Returns the identifier of the previous object in the display list.

FORMAT

prev_id = **UIS$GET_PREVIOUS_OBJECT** $\left\{ \begin{array}{l} \textit{obj\_id} \\ \textit{seg\_id} \end{array} \right\}$ *[,flags]*

RETURNS

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the previous object identifier in the variable *prev_id* or R0 (VAX MACRO). The previous object identifier uniquely identifies the previous object in the display list and is used as an argument in other routines.

UIS$GET_PREVIOUS_OBJECT signals all errors; no condition values are returned.

ARGUMENTS

obj_id
See Section 18.3.3 for a description of this argument.

seg_id
See Section 18.3.4 for a description of this argument.

flags

VMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Flags. The **flags** argument is the address of a longword that controls how the display list is searched. If the **flags** argument is specified using UIS$M_DL_SAME_SEGMENT, the previous object in the segment containing the object specified is returned.

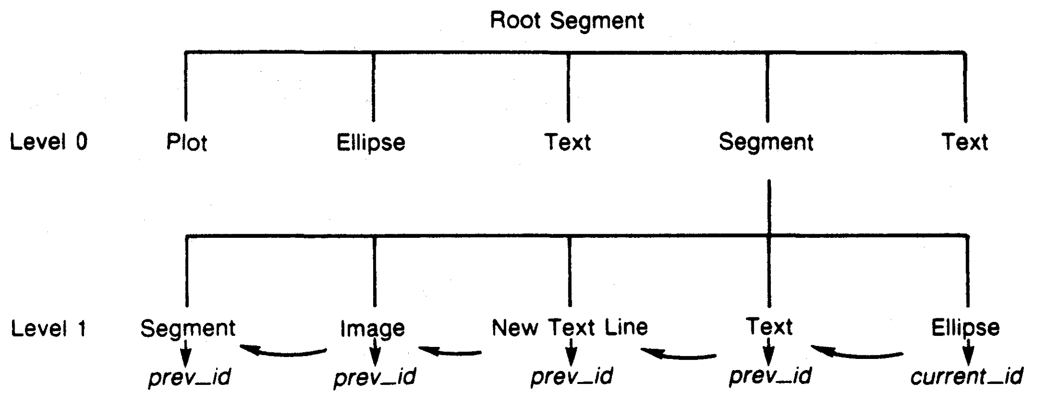
If the **flags** argument is omitted, the previous object in the display list, regardless of the segment in which it is contained, is returned.

DESCRIPTION

If no previous object is found, a zero is returned.

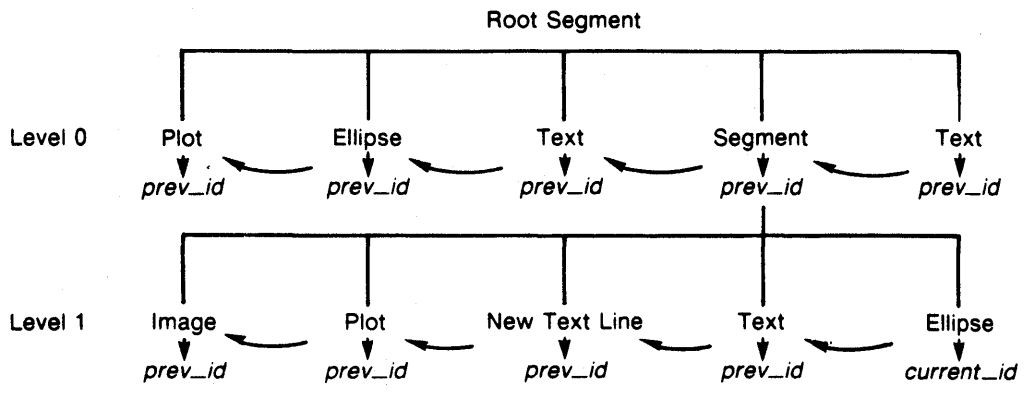
illustration

The following figure illustrates how UIS$GET_PREVIOUS_OBJECT returns the object identifier of each previous object within the same segment.



ZK 5363-86

The following figure illustrates how UIS$GET_PREVIOUS_OBJECT returns the object identifier of all objects in the display list.



ZK 5364 86

UIS$GET_ROOT_SEGMENT

Returns the root segment of the specified virtual display.

FORMAT

root_id = **UIS$GET_ROOT_SEGMENT** *vd_id*

RETURNS

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the root segment identifier in the variable *root_id* or R0 (VAX MACRO). The root segment identifier uniquely identifies the root segment.

UIS$GET_ROOT_SEGMENT signals all errors; no condition values are returned.

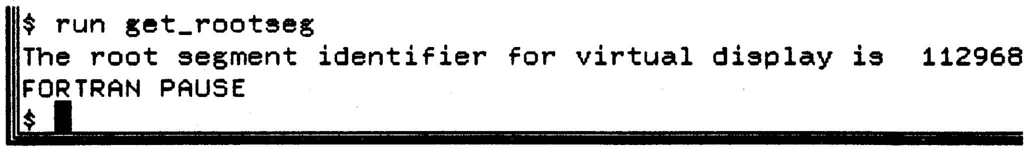
ARGUMENT

vd_id
See Section 18.3.1 for a description of this argument.

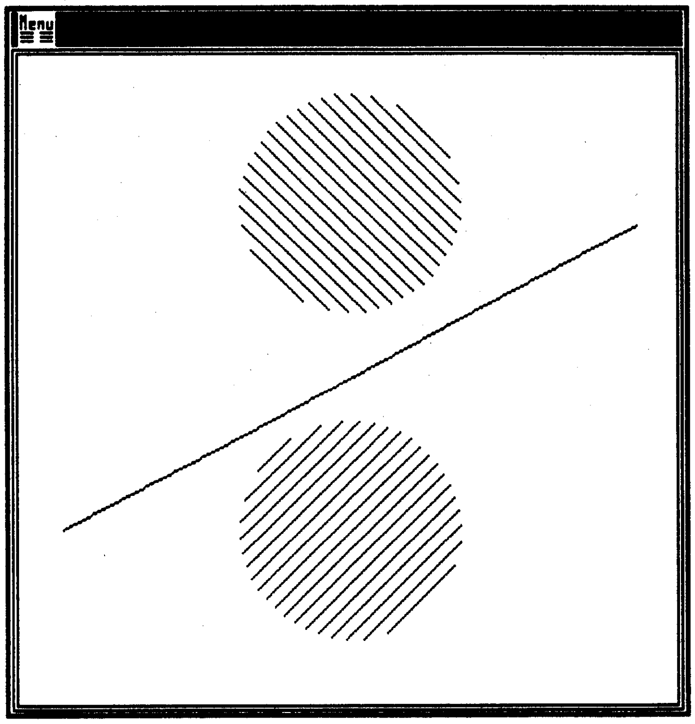
DESCRIPTION

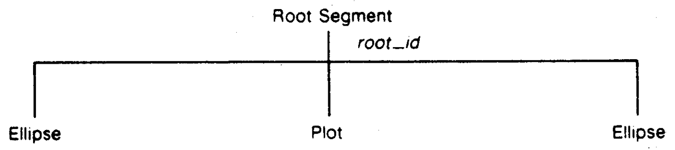
Use UIS$GET_ROOT_SEGMENT with UIS$EXTRACT_OBJECT to extract an entire display list.

screen output



```
$ run get_rootseg
The root segment identifier for virtual display is  112968
FORTRAN PAUSE
$ █
```





ZK-5366-86

UIS$GET_TB_INFO

Returns the characteristics of the tablet device.

FORMAT

status = **UIS$GET_TB_INFO** *devnam, retwidth, retheight, retresolx, retresoly [,retpwidth, retpheight]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Longword value returned in a status variable. If the value *1* is returned, the pointing device is a tablet. If the value *0* is returned, the pointing device is a mouse and the returned information will be zeros. A tablet is required for digitizing.

UIS$GET_TB_INFO signals all errors; no condition values are returned.

ARGUMENTS

devnam

See Section 18.3.9 for more information about this argument.

retwidth
retheight

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Tablet width and height. The **retwidth** argument is the address of an f_floating point longword that receives the width of the tablet in centimeters. The **retheight** argument is the address of an f_floating point longword that receives the height of the tablet in centimeters.

retresolx
retresoly

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Tablet *x* and *y* resolution. The **retresolx** argument is the address of an f_floating longword that receives the *x* resolution of the tablet in centimeters per pixel. The **retresoly** argument is the address of an f_floating point longword that receives the *y* resolution of the tablet in centimeters per pixel.

retpwidth
retpheight

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Tablet width and height. The **retpwidth** argument is the address of a longword that receives the width of the tablet in pixels. The **retpheight** argument is the address of a longword that receives the height of the tablet in pixels.

DESCRIPTION

Call UIS$GET_TB_INFO before you establish digitizing. UIS$GET_TB_INFO returns a value indicating whether the device is a mouse or tablet. A tablet is required for digitizing.

Note that if you unplug the tablet and replace it with a mouse while running an application, you might invalidate the results of this call.

UIS$GET_TB_POSITION

Polls for the position of the pointing device on the tablet.

FORMAT

UIS$GET_TB_POSITION *tb_id ,retx ,rety*

RETURNS

UIS$GET_TB_POSITION signals all errors; no condition values are returned.

ARGUMENTS

tb_id

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Tablet identifier. The **tb_id** argument is the address of a longword that uniquely identifies the tablet. See UIS$CREATE_TB for more information about the **tb_id** argument.

retx
rety

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Digitizer position. The **retx, rety** arguments are the addresses of f_floating numbers that define the current digitizer position.

DESCRIPTION

The digitizer position is not available if the pointing device is a mouse.

If the pointer is not on the tablet, UIS$GET_TB_POSITION returns the last reported pointer.

UIS$GET_TEXT_FORMATTING

Returns a mask describing the enabled text formatting modes.

FORMAT

formatting = **UIS$GET_TEXT_FORMATTING** *vd_id, atb*

RETURNS

VMS Usage: **mask_longword**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword mask returned as the current formatting mode in the variable *formatting* or R0 (VAX MACRO). The following table lists the formatting modes.

Formatting Mode	Function
UIS$C_TEXT_FORMAT_LEFT	Left justified, ragged right (default)
UIS$C_TEXT_FORMAT_RIGHT	Right justified, left ragged
UIS$C_TEXT_FORMAT_CENTER	Centered line between left and right margin
UIS$C_TEXT_FORMAT_JUSTIFY	Justified lines, space filled to right margin
UIS$C_TEXT_FORMAT_NOJUSTIFY	No text justification

UIS$GET_TEXT_FORMATTING signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

atb

VMS Usage: **longword_signed**
type: **longword (signed)**
access: **read only**
mechanism: **by reference**

Attribute block number. The **atb** argument is the address of a longword that identifies an attribute block containing the text formatting attribute setting to be returned.

UIS$GET_TEXT_MARGINS

Returns the text margins for a line of text.

FORMAT

UIS$GET_TEXT_MARGINS *vd_id ,atb ,x ,y [,margin_length]*

RETURNS

UIS$GET_TEXT_MARGINS signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword that identifies an attribute block containing the modified text margins attribute.

x
y

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

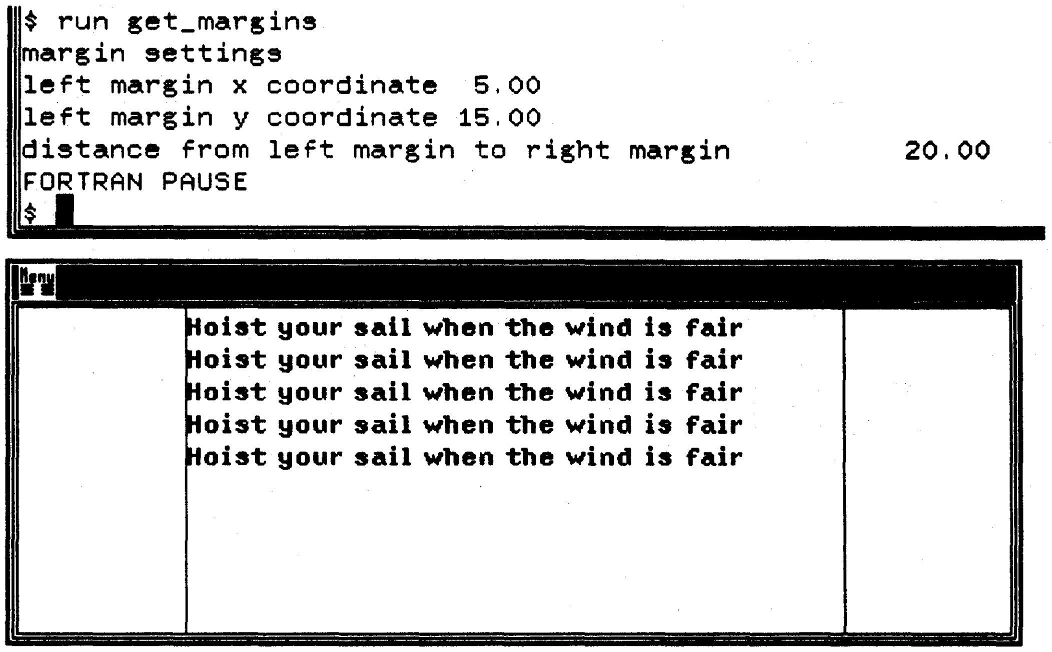
Starting margin position. The **x,y** arguments are the addresses of f_floating longwords that receive the starting margin relative to the direction of text drawing.

margin_length

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Ending margin position. The **margin_length** is the address of an f_floating longword that receives the distance to the end margin. The margin is measured along the actual path of text drawing in the direction of the major text path.

screen output



```
$ run get_margins
margin settings
left margin x coordinate  5.00
left margin y coordinate 15.00
distance from left margin to right margin          20.00
FORTRAN PAUSE
$
```

```
Hoist your sail when the wind is fair
Hoist your sail when the wind is fair
Hoist your sail when the wind is fair
Hoist your sail when the wind is fair
Hoist your sail when the wind is fair
```

ZK 5281-86

UIS$GET_TEXT_PATH

Returns text path types. See UIS$SET_TEXT_PATH for information about valid text path types.

FORMAT

UIS$GET_TEXT_PATH *vd_id, atb [,major] [,minor]*

RETURNS

UIS$GET_TEXT_PATH signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a number that identifies an attribute block containing the text path attribute setting to be returned.

major

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Major text path type. The **major** argument is the address of a code that identifies a major text path type. The major text path of text drawing is the direction of text drawing along a line.

minor

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Minor text path type. The **minor** argument is the address of a code that identifies a minor text path type. The minor path of text drawing is the direction used for new text line creation.

DESCRIPTION

The following table contains symbols for valid character drawing directions.

Path	Direction
UIS$C_TEXT_PATH_RIGHT	Left to right (default major text path)
UIS$C_TEXT_PATH_LEFT	Right to left
UIS$C_TEXT_PATH_UP	Bottom to top
UIS$C_TEXT_PATH_DOWN	Top to bottom (default minor text path)

UIS$GET_TEXT_SLOPE

Returns the angle of the actual path of text drawing relative to the major path in degrees.

FORMAT

angle = **UIS$GET_TEXT_SLOPE** *vd_id, atb*

RETURNS

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by value**

Longword value returned as the angle of the actual path of text drawing relative to the major path in degrees in the variable *angle* or R0 (VAX MACRO). Degrees are measured counterclockwise.

UIS$GET_TEXT_SLOPE signals all errors; no condition values are returned.

ARGUMENTS

vd_id

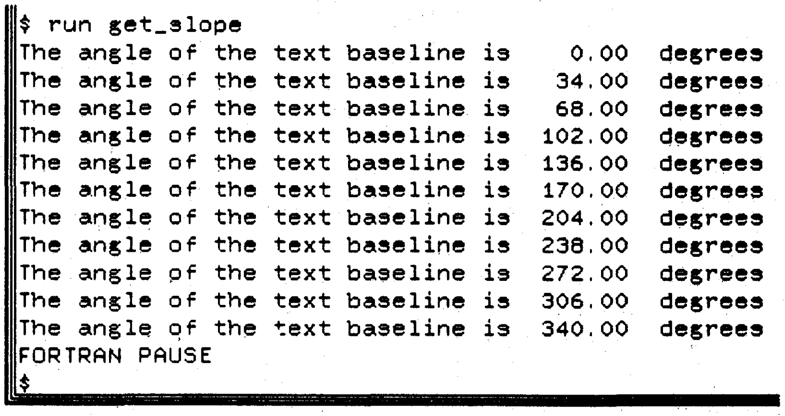
See Section 18.3.1 for a description of this argument.

atb

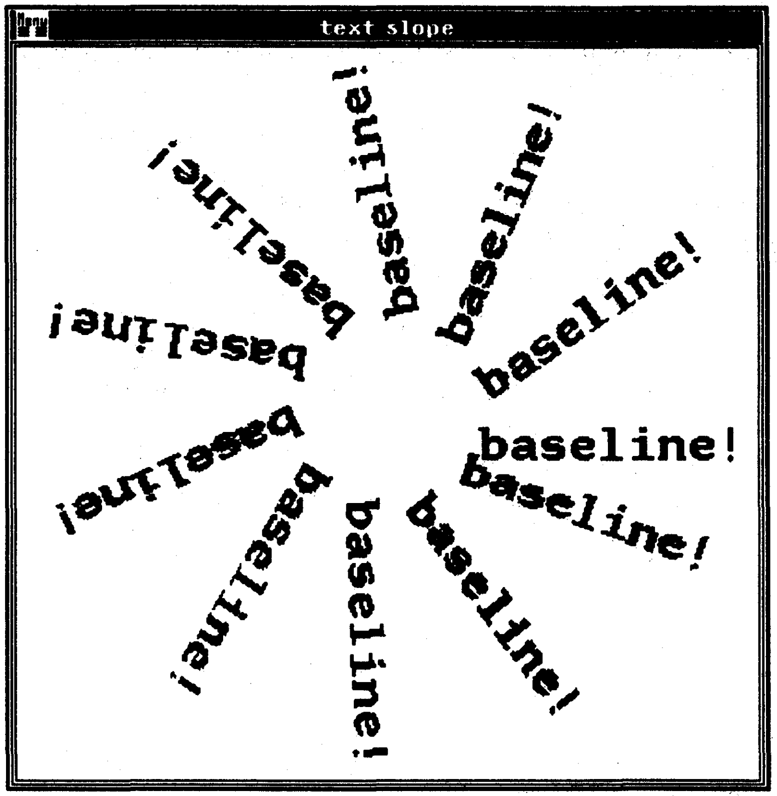
VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword that identifies an attribute block from which the text slope attribute setting is to be returned.

screen output



```
$ run get_slope
The angle of the text baseline is    0.00  degrees
The angle of the text baseline is   34.00  degrees
The angle of the text baseline is   68.00  degrees
The angle of the text baseline is  102.00  degrees
The angle of the text baseline is  136.00  degrees
The angle of the text baseline is  170.00  degrees
The angle of the text baseline is  204.00  degrees
The angle of the text baseline is  238.00  degrees
The angle of the text baseline is  272.00  degrees
The angle of the text baseline is  306.00  degrees
The angle of the text baseline is  340.00  degrees
FORTRAN PAUSE
$
```



ZK 5294 86

UIS$GET_VCM_ID

Returns the virtual color map identifier used by the specified virtual display.

FORMAT

vcm_id = **UIS$GET_VCM_ID** *vd_id*

RETURNS

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the virtual color map identifier in the variable *vcm_id* or R0 (VAX MACRO). The virtual color map identifier uniquely identifies a virtual color map for a specified virtual display. See UIS$CREATE_COLOR_MAP for more information about the **vcm_id** argument.

UIS$GET_VCM_ID signals all errors; no condition values are returned.

ARGUMENT

vd_id

See Section 18.3.1 for a description of this argument.

UIS$GET_VIEWPORT_ICON

Returns Boolean value indicating whether an icon has replaced a viewport.

FORMAT

Boolean = **UIS$GET_VIEWPORT_ICON** *wd_id [,icon_wd_id]*

RETURNS

VMS Usage:	**Boolean**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Boolean value returned in a status variable or R0 (VAX MACRO) indicating whether an icon has replaced a viewport. A *1* denotes a TRUE condition; *0* denotes a FALSE condition.

UIS$GET_VIEWPORT_ICON signals all errors; no condition values are returned.

ARGUMENTS

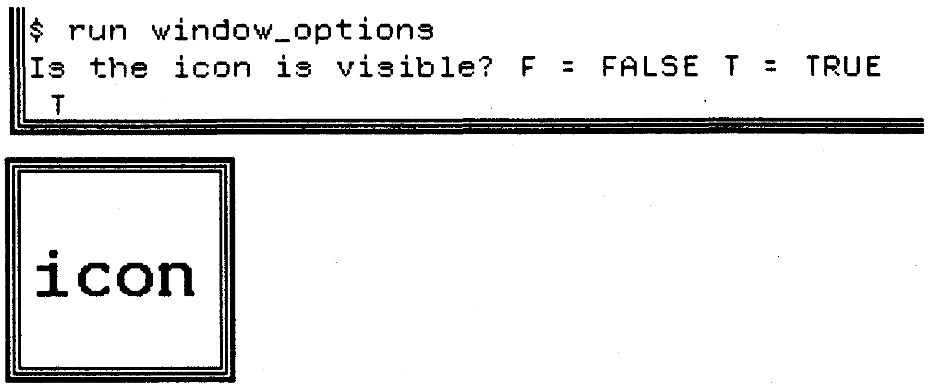
wd_id
See Section 18.3.2 for a description of this argument.

icon_wd_id

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by reference**

Icon identifier. The **icon_wd_id** argument is the address of a longword that uniquely identifies the icon.

screen output



```
$ run window_options
Is the icon is visible? F = FALSE T = TRUE
 T
```

ZK-5270-86

UIS$GET_VIEWPORT_POSITION

Returns the position of the lower-left corner of the display viewport relative to the lower-left corner of the screen.

FORMAT

UIS$GET_VIEWPORT_POSITION *wd_id, retx, rety*

RETURNS

UIS$GET_VIEWPORT_POSITION signals all errors; no condition values are returned.

ARGUMENTS

wd_id
See Section 18.3.2 for a description of this argument.

retx
rety

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Absolute device coordinate pair. The **retx** and **rety** arguments are the addresses of f_floating point longwords that receive the *x* and *y* coordinates of the display viewport origin in centimeters.

These coordinates refer to the inside of the viewport and do not include the border.

DESCRIPTION

UIS$GET_VIEWPORT_POSITION is useful in the exact placement of windows.

screen output

See UIS$GET_VIEWPORT_SIZE.

UIS$GET_VIEWPORT_SIZE

Returns the size of the display viewport associated with the specified display window.

FORMAT

UIS$GET_VIEWPORT_SIZE *wd_id, retwidth, retheight*

RETURNS

UIS$GET_VIEWPORT_SIZE signals all errors; no condition values are returned.

ARGUMENTS

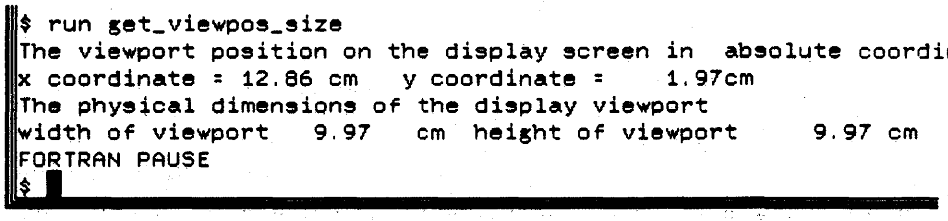
wd_id
See Section 18.3.2 for a description of this argument.

retwidth
retheight

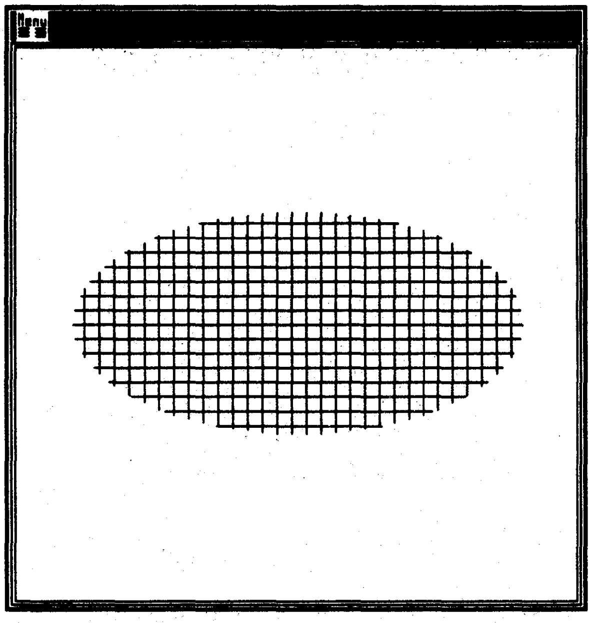
VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Display viewport width and height. The **retwidth** and **retheight** arguments are the addresses of f_floating point longwords that receive the display viewport width and height in centimeters.

screen output



```
$ run get_viewpos_size
The viewport position on the display screen in  absolute coordinates
x coordinate = 12.86 cm   y coordinate =     1.97cm
The physical dimensions of the display viewport
width of viewport   9.97   cm  height of viewport     9.97 cm
FORTRAN PAUSE
$
```



ZK 5251 86

UIS$GET_VISIBILITY

Returns a Boolean value that indicates whether or not the specified rectangle in the display window is visible.

FORMAT

status = **UIS$GET_VISIBILITY** *vd_id, wd_id [,x_1, y_1 [,x_2, y_2]]*

RETURNS

VMS Usage:	**Boolean**
type:	**longword**
access:	**write only**
mechanism:	**by value**

Boolean value returned in a status variable or R0 (VAX MACRO). The returned value, the visibility status, is a Boolean TRUE only if the entire area is visible, and a Boolean FALSE if even a portion of the area is occluded or clipped.

UIS$GET_VISIBILITY signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

wd_id
See Section 18.3.2 for a description of this argument.

x_1, y_1
x_2, y_2

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

World coordinates of a rectangle in the display window. The x_1 and y_1 arguments are addresses of f_floating point numbers that define the lower-left corner of a rectangle in the display window. The x_2 and y_2 arguments are addresses of f_floating point numbers that define the upper-right corner of a rectangle in the display window.

If the coordinates of the rectangle are not specified, the dimensions of the entire display window are used by default.

If only one point is specified, only that point is checked.

UIS$GET_WINDOW_ATTRIBUTES

Returns the value of the mask WDPL$C_ATTRIBUTES used in the creation of the specified window. See UIS$CREATE_WINDOW for more information about window and viewport attributes.

FORMAT

attributes = **UIS$GET_WINDOW_ATTRIBUTES** *wd_id*

RETURNS

VMS Usage:	**mask_longword**
type:	**longword**
access:	**write only**
mechanism:	**by value**

Longword mask representing one or more attributes of the specified display window and returned in the variable *attributes* or R0 (VAX MACRO). See UIS$CREATE_WINDOW for more information.

UIS$GET_WINDOW_ATTRIBUTES signals all errors; no condition values are returned.

ARGUMENT

wd_id
See Section 18.3.2 for a description of this argument.

UIS$GET_WINDOW_SIZE

Returns the dimensions of the display window.

FORMAT **UIS$GET_WINDOW_SIZE** *vd_id, wd_id,* x_1, y_1, x_2, y_2

RETURNS UIS$GET_WINDOW_SIZE signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

wd_id
See Section 18.3.2 for a description of this argument.

x_1, y_1
x_2, y_2

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

World coordinate pairs. The x_1,y_1 and the x_2,y_2 arguments are the addresses of f_floating longwords that receive the locations of the lower-left and upper-right corners of the display window in world coordinates.

UIS$GET_WRITING_INDEX

Returns the writing color index for text and graphics output.

FORMAT

index = **UIS$GET_WRITING_INDEX** *vd_id, atb*

RETURNS

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the color map index in the variable *index* or R0 (VAX MACRO).

UIS$GET_WRITING_INDEX signals all errors; no condition values are returned.

ARGUMENTS

vd_id

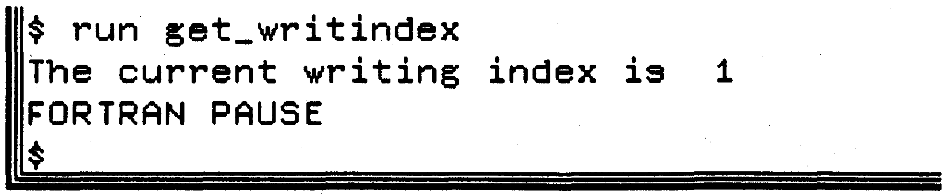
See Section 18.3.1 for a description of this argument.

atb

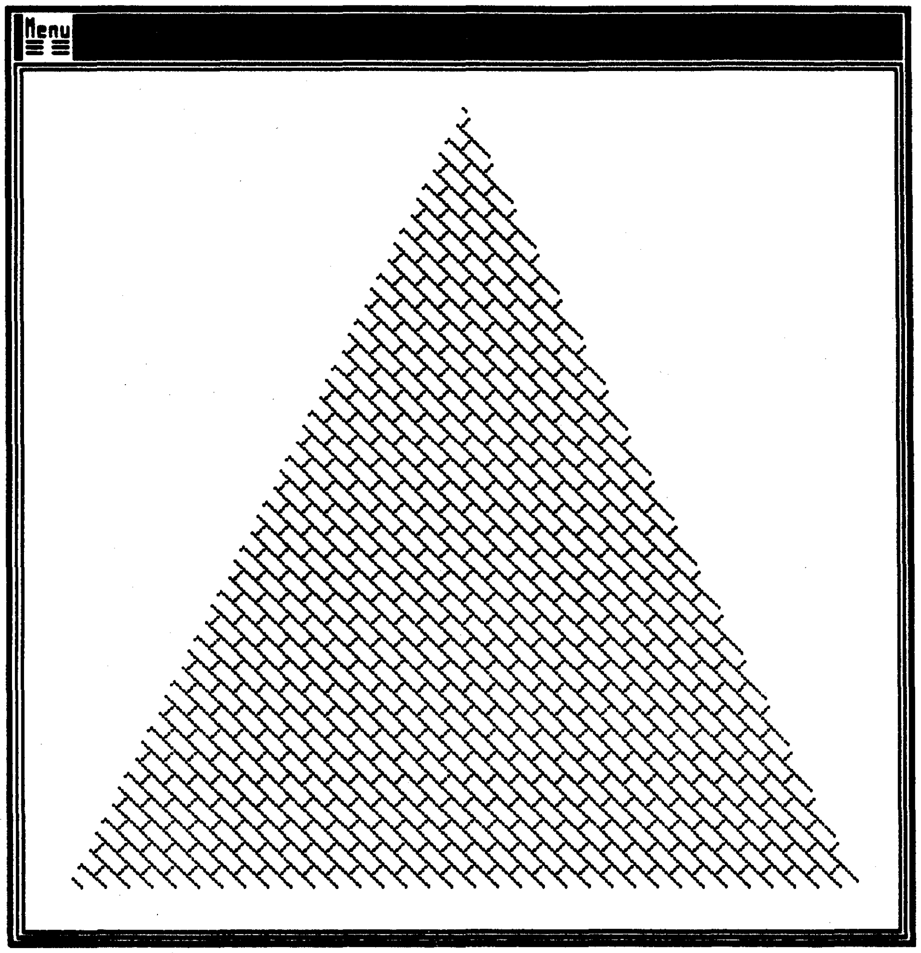
VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword integer that identifies an attribute block from which the writing color index is obtained.

screen output



```
$ run get_writindex
The current writing index is  1
FORTRAN PAUSE
$
```



ZK-5269-86

UIS$GET_WRITING_MODE

Returns the writing mode.

FORMAT

mode = **UIS$GET_WRITING_MODE** *vd_id, atb*

RETURNS

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by value**

Longword value returned as a UIS writing mode in the variable *mode* or R0 (VAX MACRO). See Section 9.4 for more information about writing modes.

UIS$GET_WRITING_MODE signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword integer that identifies an attribute block from which the writing mode is obtained.

UIS$GET_WS_COLOR

Returns the R (red), G (green), and B (blue) values associated with the workstation standard color.

FORMAT

UIS$GET_WS_COLOR *vd_id, color_id, retr, retg, retb [,wd_id]*

RETURNS

UIS$GET_WS_COLOR signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

color_id

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Workstation standard color. The **color_id** argument is the address of a longword integer that identifies a symbolic code for the workstation standard color. If the **color_id** argument is invalid, an error is signaled.

The following table lists possible workstation standard color symbols and their current values.

Standard Color	Symbol
Background	UIS$C_WS_BCOLOR
Foreground	UIS$C_WS_FCOLOR
Black	UIS$C_WS_BLACK
White	UIS$C_WS_WHITE
Red	UIS$C_WS_RED
Green	UIS$C_WS_GREEN
Blue	UIS$C_WS_BLUE
Cyan	UIS$C_WS_CYAN
Yellow	UIS$C_WS_YELLOW
Magenta	UIS$C_WS_MAGENTA
Grey (25%)	UIS$C_WS_GREY25
Grey (50%)	UIS$C_WS_GREY50
Grey (75%)	UIS$C_WS_GREY75

retr

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Red value. The **retr** argument is the address of an f_floating point longword that receives the red value. The red value is in the range of *0.0* to *1.0*, inclusive.

retg

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Green value. The **retg** argument is the address of an f_floating point longword that receives the green value. The green value is in the range of *0.0* to *1.0*, inclusive.

retb

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Blue value. The **retb** argument is the address of an f_floating point longword that receives the blue value. The blue value is in the range of *0.0* to *1.0*, inclusive.

wd_id

See Section 18.3.2 for a description of this argument.

UIS$GET_WS_INTENSITY

Returns the intensity values associated with a workstation standard color.

FORMAT

UIS$GET_WS_INTENSITY *vd_id, color_id, reti [,wd_id]*

RETURNS

UIS$GET_WS_INTENSITY signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

color_id

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Workstation standard color identifier. The **color_id** argument is the address of a longword that identifies a symbolic code for the workstation standard color. If the **color_id** argument is invalid, an error is signaled.

The following table lists possible workstation standard color symbols.

Standard Color	Symbol
Background	UIS$C_WS_BCOLOR
Foreground	UIS$C_WS_FCOLOR
Black	UIS$C_WS_BLACK
White	UIS$C_WS_WHITE
Red	UIS$C_WS_RED
Green	UIS$C_WS_GREEN
Blue	UIS$C_WS_BLUE
Cyan	UIS$C_WS_CYAN
Yellow	UIS$C_WS_YELLOW
Magenta	UIS$C_WS_MAGENTA
Grey (25%)	UIS$C_WS_GREY25
Grey (50%)	UIS$C_WS_GREY50
Grey (75%)	UIS$C_WS_GREY75

reti

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Intensity value. The **reti** argument is the address of an f_floating longword that receives the intensity value. The intensity value is in the range of *0.0* to *1.0*, inclusive.

wd_id

See Section 18.3.2 for a description of this argument.

If this argument is specified, then it must be a valid **wd_id** associated with the virtual display, and the returned values are the realized intensities for the specific device for which the window was created.

UIS$HLS_TO_RGB

Converts color representation values of hue, lightness, and saturation (HLS) to red, green, and blue (RGB) values.

FORMAT

UIS$HLS_TO_RGB *H, L, S, retr, retg, retb*

RETURNS

UIS$HLS_TO_RGB signals all errors; no condition values are returned.

ARGUMENTS

H

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Hue. The **H** argument is the address of an f_floating number that defines the hue of a color.

L

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Lightness. The **L** argument is the address of an f_floating number that defines the lightness of a color.

S

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Saturation. The **S** argument is the address of an f_floating number that defines color saturation.

retr

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Red value. The **retr** argument is the address of an f_floating point longword that receives the red value.

retg

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Green value. The **retg** argument is the address of an f_floating point longword that receives the green value.

retb

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Blue value. The **retb** argument is the address of an f_floating point longword that receives the blue value.

UIS$HSV_TO_RGB

Converts color representation values of hue, saturation, and value (HSV) to red, green, and blue (RGB) values.

FORMAT

UIS$HSV_TO_RGB *H, S, V, retr, retg,retb*

RETURNS

UIS$HSV_TO_RGB signals all errors; no condition values are returned.

ARGUMENTS

H

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Hue. The **H** argument is the address of an f_floating number that defines the hue of a color.

S

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Saturation. The **S** argument is the address of an f_floating number that defines the saturation of a color.

V

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Value. The **V** argument is the address of an f_floating number that defines the value of a color.

retr

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Red value. The **retr** argument is the address of an f_floating longword that receives the red color value.

retg

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Green value. The **retg** argument is the address of an f_floating longword that receives the green color value.

retb

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Blue value. The **retb** argument is the address of an f_floating longword that receives the blue color value.

UIS$IMAGE

Draws a raster image in a specified rectangle in the display viewport.

FORMAT

UIS$IMAGE *vd_id, atb, x_1, y_1, x_2, y_2, rasterwidth, rasterheight, bitsperpixel, rasteraddr*

RETURNS

UIS$IMAGE signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword integer that identifies an attribute block that modifies the image.

x_1, y_1
x_2, y_2

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

World coordinates of the rectangle in the virtual display. The x_1 and y_1 arguments are the addresses of f_floating point numbers that define the lower-left corner of the rectangle in the virtual display. The x_2 and y_2 arguments are the addresses of f_floating point numbers that define the upper-right corner of the rectangle in the virtual display.

rasterwidth
rasterheight

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Width and height of the raster image. The **rasterwidth** argument is the address of a longword that defines the width of the raster image in pixels. The **rasterheight** is the address of a longword that defines the height of the raster image in pixels.

bitsperpixel

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Number of bits per pixel in the raster image. The **bitsperpixel** argument is the address of a longword that defines the number of bits per pixel in the raster image. The **bitsperpixel** argument is currently required to be *1* or *8*.

If the value *8* is specified for **bitsperpixel** on a single plane system, the results are unpredictable.

rasteraddr

VMS Usage:	**vector_longword_unsigned**
type:	**longword_unsigned**
access:	**read only**
mechanism:	**by reference**

Bitmap image. The **rasteraddr** argument is the address of an array that defines a bitmap image. You must first create a bitmap by defining a data structure such as a record or array. When you assign values to the field or array element in the data structure, you are setting the bits of the image to be drawn by UIS$IMAGE. See the Description section for information about setting bits.

DESCRIPTION

The bitmap image is drawn to the display viewport as a raster image. The raster image dimensions are described by width, height, and bits per pixel. Width and height give the number of pixels in each dimension, and bits per pixel represents the number of bits in each pixel. The raster is read from memory as "height" bit vectors; each vector is "width" pixels long and each pixel is "bits/pixel" bits long.

If the destination rectangle is larger than the raster size by at least an integer multiple, the raster is automatically scaled on a per pixel basis to the space available. Thus, a 1 x 1 raster can be written into an arbitrarily large destination rectangle, and the entire region is filled with the pattern.

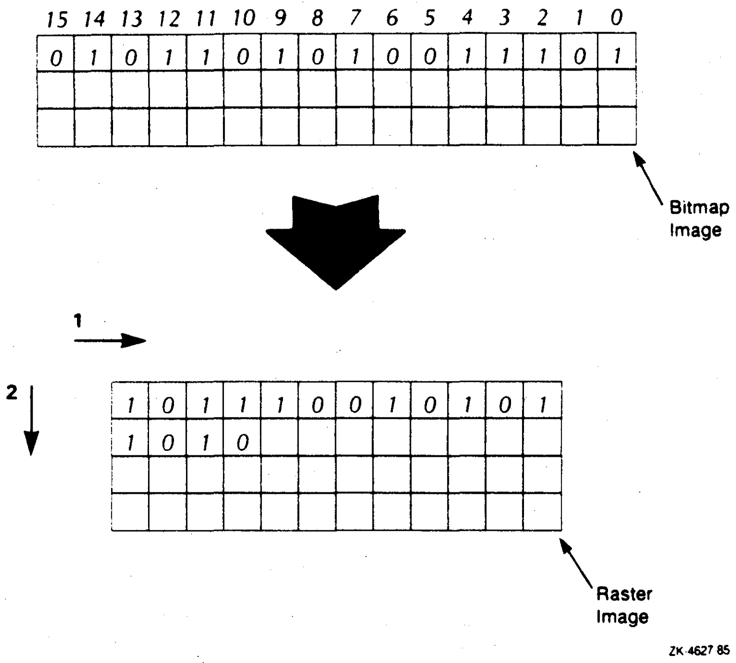
If the destination rectangle is not an exact multiple of the raster size, then the remaining space on the right and top will not be written.

The procedure to map values in the bitmap to the raster image is as follows:

1. Each bit in the raster is set from left-most bit to the right-most bit
2. Each row is filled from the top row to the bottom row.

NOTE: **The raster image is not byte- or word-aligned.**

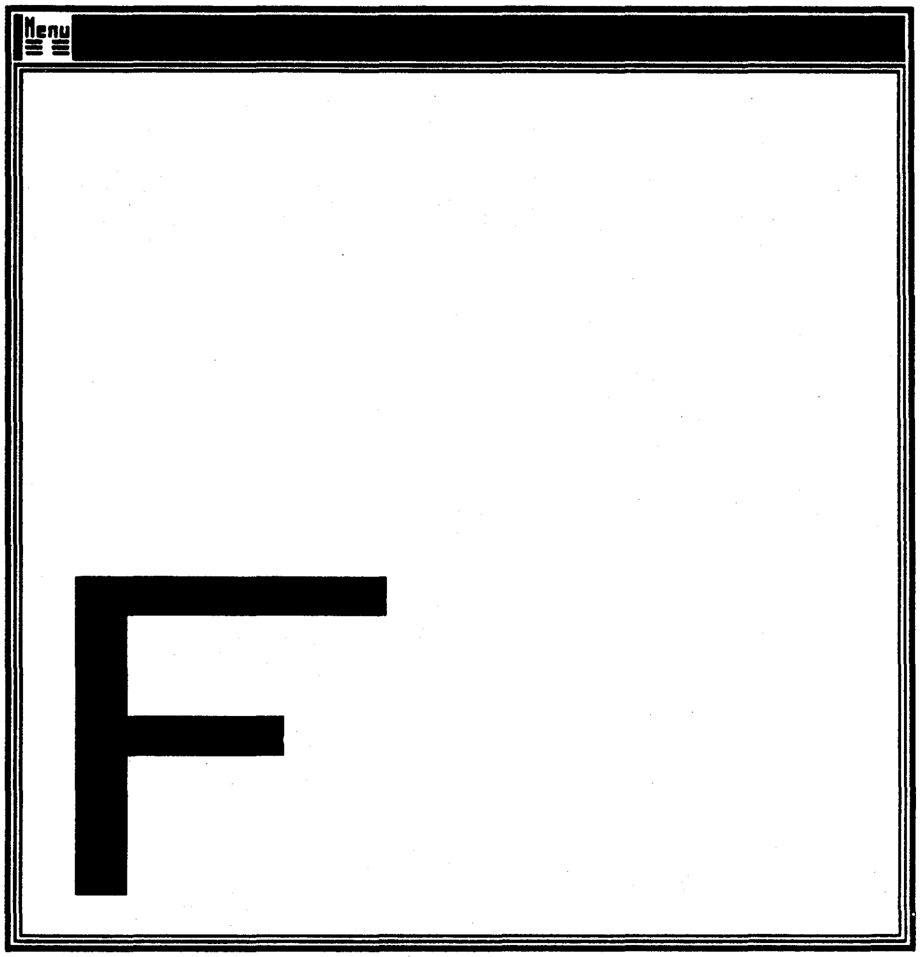
The following figure illustrates the setting of bits in the bitmap.



EXAMPLE

```
.
.
.
        INTEGER*2 BITMAP(20)
        DATA BITMAP/2*0,2*16380,5*12,2*1020,7*12,2*0/

        VD_ID=UIS$CREATE_DISPLAY(0.0,0.0,40.0,40.0,10.0,10.0)
        WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION')

        CALL UIS$IMAGE(VD_ID,0,0.0,0.0,20.0,20.0,16,20,1,BITMAP)
.
.
.
```

screen output



ZK-5267-86

UIS$INSERT_OBJECT

Inserts the specified object into the display list at the position specified by the insertion pointer. See UIS$SET_INSERTION_POSITION for information about setting the pointer in the display list.

FORMAT

UIS$INSERT_OBJECT $\left\{ \begin{array}{l} \textit{obj\_id} \\ \textit{seg\_id} \end{array} \right\}$

RETURNS

UIS$INSERT_OBJECT signals all errors; no condition values are returned.

ARGUMENTS

obj_id
See Section 18.3.3 for a description of this argument.

seg_id
See Section 18.3.4 for a description of this argument.

UIS$LINE

Draws an unfilled point, line, or series of unconnected lines depending on the number of positions specified.

FORMAT

UIS$LINE *vd_id, atb,* x_1, y_1 *[,*x_2,y_2 *[,...*x_n,y_n*]]*

RETURNS

UIS$LINE signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword integer that identifies an attribute block that modifies line style and line width or both.

x
y

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

World coordinate pair. The **x** and **y** arguments are the addresses of f_ floating point numbers that define a point in the virtual display. If the arguments are repeated to specify a second position, a line is created. You can specify up to 126 world coordinate pairs as arguments. See the Description section below for more information about this argument.

DESCRIPTION

If one position is specified, then a point is drawn. If two positions are specified, a single vector is drawn. If more than two positions are specified, unconnected lines are drawn. Up to 252 arguments can be specified, a maximum of a 126 unconnected lines are drawn using this routine. If a larger number of points must be specified in a single call, UIS$LINE_ARRAY should be used.

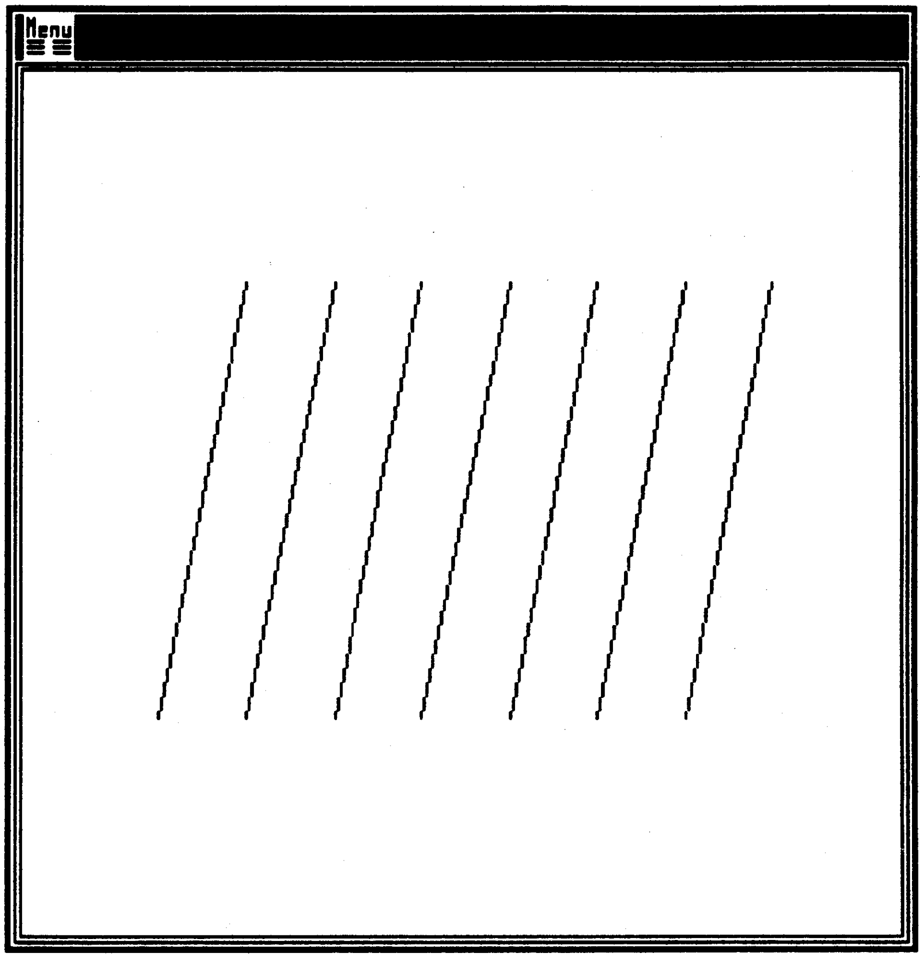
The points or lines are drawn with the line pattern and width for the attribute block. UIS$LINE ignores the fill pattern attribute.

EXAMPLE

```
   .
   .
   .
call uis$line(vd_id,0,3.0,5.0,5.0,15.0,5.0,5.0,7.0,15.0,7.0,5.0,
2       9.0,15.0,
2       9.0,5.0,11.0,15.0,11.0,5.0,13.0,15.0,
2       13.0,5.0,15.0,15.0,15.0,5.0,17.0,15.0)
   .
   .
   .
```

A single call to UIS$LINE draws five unconnected lines.

screen output



ZK-5419-86

UIS$LINE_ARRAY

Draws an unfilled point, line, or series of unconnected lines depending on the number of positions specified. This routine performs the same functions as UIS$LINE except that x and y coordinates are stored in arrays.

FORMAT

UIS$LINE_ARRAY *vd_id, atb, count, x_vector, y_vector*

RETURNS

UIS$LINE_ARRAY signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword integer that identifies an attribute block that modifies line style or line width or both.

count

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Number of points. The **count** argument is the address of longword integer that denotes the number of world coordinate pairs defined in the arguments **x_vector** and **y_vector**.

x_vector
y_vector

VMS Usage:	**vector_longword_signed**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Array of *x* and *y* world coordinates. The **x_vector** argument is the address of an array of f_floating numbers whose elements are the *x* world coordinate values of points defined in the virtual display. The **y_vector** argument is the address of an array of f_floating numbers whose elements are the *y* world coordinate values of points defined in the virtual display.

DESCRIPTION

You can plot up to 32,767 points in a single call. UIS$LINE_ARRAY is the same as UIS$LINE, except that you specify the *x* and *y* coordinates with two arrays, each of length *count* points.

UIS$MEASURE_TEXT

Measures a text string as if it were output in a virtual display.

FORMAT

UIS$MEASURE_TEXT *vd_id, atb, text_string, retwidth, retheight, [,ctllist, ctllen] [,posarray]*

RETURNS

UIS$MEASURE_TEXT signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword integer that identifies an attribute block that modifies text output.

text_string

VMS Usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

Text string. The **text_string** argument is the address of a character string descriptor of a text string.

retwidth
retheight

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

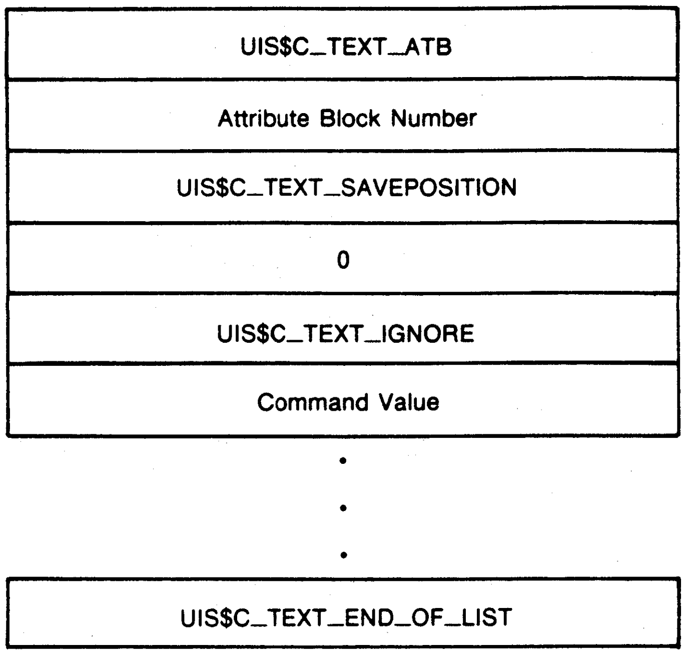
World coordinate width and height. The **retwidth** and **retheight** arguments are the addresses of f_floating point longwords that receive the world coordinate width and height of the text.

ctllist

VMS Usage:	**vector_longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Text formatting control list. The **ctllist** argument is the address of an array of longwords that describe the font, text rendition, format, and positioning of fragments of the text string. See UIS$TEXT for a description of the control list and its commands.

The control list consists of a sequence of data elements, each two longwords in length. The first longword of each element is a tag. The second longword is either a value particular to the type of element specified or zero. The following diagram shows the structure of a text control list.



ZK-5426-86

The following table lists valid formatting commands and functions.

Formatting Command	Function
Commands Without Values[1]	
UIS$C_TEXT_NOP	Nil operation
UIS$C_TEXT_RESTORE_POSITION	Restores the current writing position
UIS$C_TEXT_SAVE_POSITION	Saves the current writing position
Commands Requiring Values	
UIS$C_TEXT_ATB	Specifies an attribute block number
UIS$C_TEXT_HPOS_ABSOLUTE	Specifies a new current x position
UIS$C_TEXT_HPOS_RELATIVE	Modifies the current x position by a delta
UIS$C_TEXT_IGNORE	Skips *n* characters
UIS$C_TEXT_NEW_LINE	Skips *n* new lines and positions at the left margin

[1]Second longword must be zero.

Formatting Command	Function
Commands Requiring Values	
UIS$C_TEXT_TAB_ABSOLUTE	Writes white space to the new absolute position
UIS$C_TEXT_TAB_RELATIVE	Writes white space to the new relative position
UIS$C_TEXT_VPOS_ABSOLUTE	Writes a new current y position
UIS$C_TEXT_VPOS_RELATIVE	Modifies the current y position by a delta
UIS$C_TEXT_WRITE	Writes *n* characters
Commands Not Requiring a Second Longword	
UIS$C_TEXT_END_OF_LIST	Terminates the control list

When UIS encounters illegal commands and values within the control list, it skips the invalid item and signals an error.

ctllen

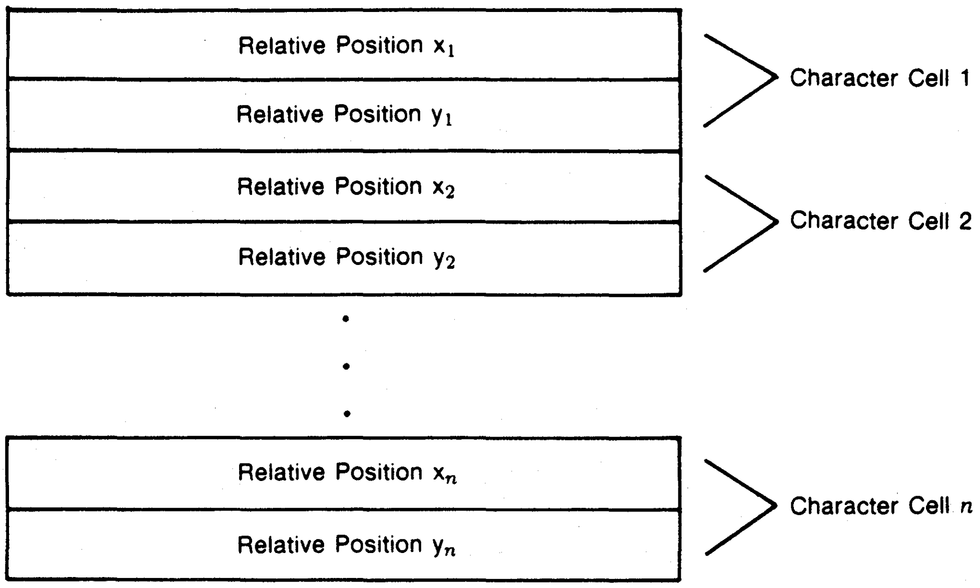
VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Length of the text formatting control list. The **ctllen** argument is the address of a longword that specifies the length of the text formatting control list in longwords.

posarray

VMS Usage:	**vector_longword_signed**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Character position array. The **posarray** argument is the address of an array of longwords that receives the character positions in world coordinates, that is, relative offsets at which each character would have been displayed. Following is a diagram showing the format of the character position array.



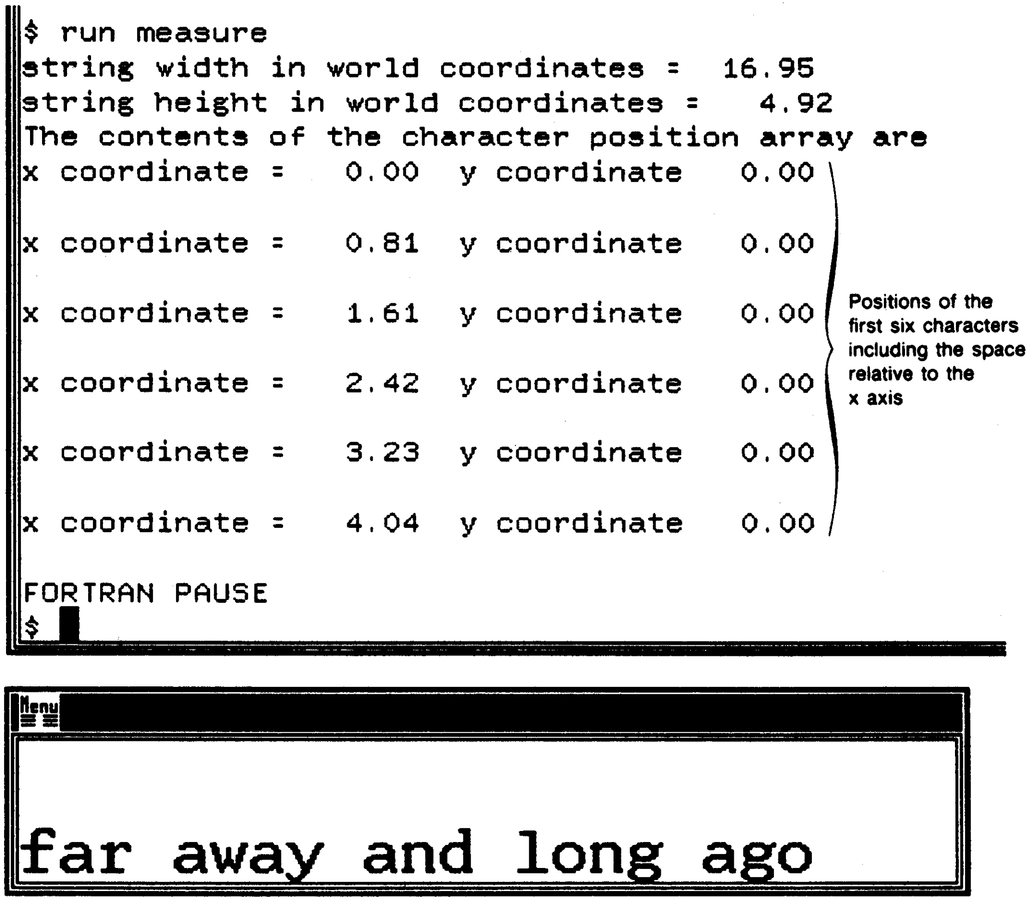
ZK-5425-86

The width and height of the text string is calculated according to the formatting described in the **atb** and **ctllist** arguments.

DESCRIPTION

You use UIS$MEASURE_TEXT in justification and text positioning applications. The routine returns the height and width of the text string in world coordinates.

screen output



```
$ run measure
string width in world coordinates =  16.95
string height in world coordinates =   4.92
The contents of the character position array are
x coordinate =   0.00  y coordinate   0.00

x coordinate =   0.81  y coordinate   0.00

x coordinate =   1.61  y coordinate   0.00

x coordinate =   2.42  y coordinate   0.00

x coordinate =   3.23  y coordinate   0.00

x coordinate =   4.04  y coordinate   0.00

FORTRAN PAUSE
$
```

Positions of the first six characters including the space relative to the x axis
Menu
far away and long ago

ZK 5266-86

UIS$MOVE_AREA

Shifts a portion of a virtual display to another position in the display window.

FORMAT **UIS$MOVE_AREA** *vd_id, x_1, y_1, x_2, y_2, new_x, new_y*

RETURNS UIS$MOVE_AREA signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

x_1, y_1
x_2, y_2

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

World coordinates of the source rectangle. The x_1 and y_1 arguments are the addresses of f_floating point numbers that define the lower-left corner of the source rectangle. The x_2 and y_2 are the addresses of f_floating point numbers that define the upper-right corner of the source rectangle.

new_x
new_y

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

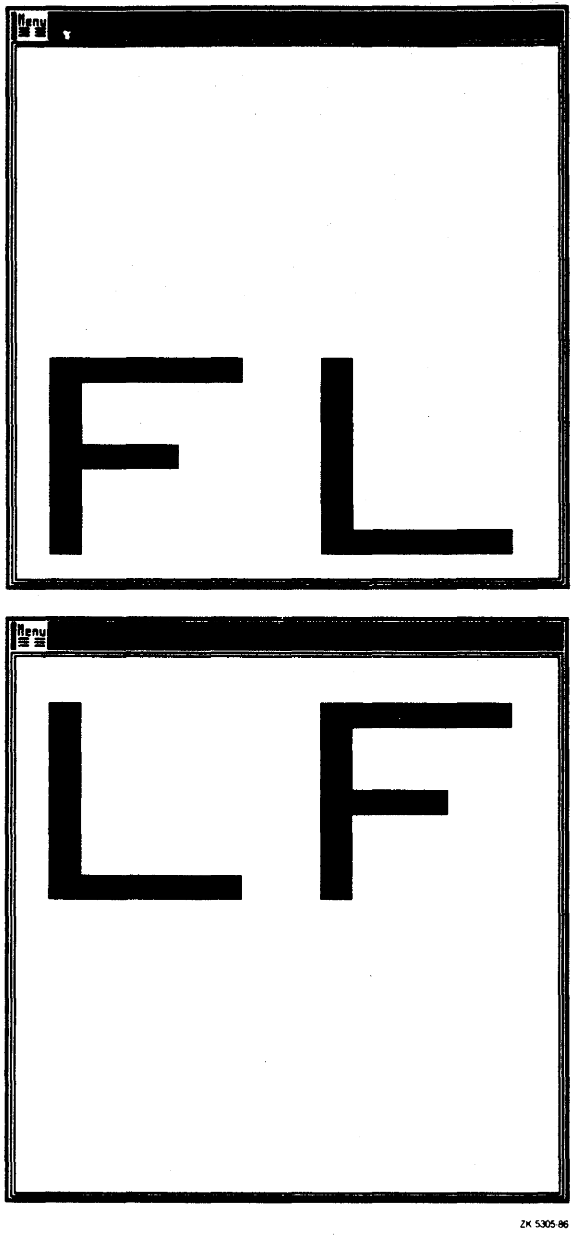
World coordinate pair. The **new_x** and **new_y** arguments are the addresses of f_floating point numbers that define the lower-left corner of the destination rectangle. The proportions of the coordinate space of the destination rectangle are the same as those of the source rectangle.

DESCRIPTION Note that display objects only partially contained within the specified source rectangle, although partially moved within existing display windows, are completely moved within the display list.

The nonoccluding portion of the source rectangle (if any) is erased after the operation.

NOTE: **To avoid distortion within the destination rectangle, the aspect ratios of the source rectangle and the display viewport must be equal.**

screen output



UIS$MOVE_VIEWPORT

Moves the display viewport on the workstation screen.

FORMAT

UIS$MOVE_VIEWPORT *wd_id, attributes*

RETURNS

UIS$MOVE_VIEWPORT signals all errors; no condition values are returned.

ARGUMENTS

wd_id

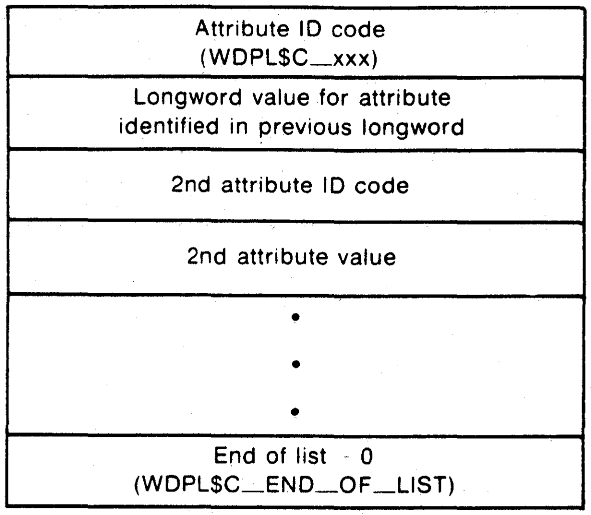
See Section 18.3.2 for a description of this argument.

attributes

VMS Usage:	**item_list_pair**
type:	**longword**
access:	**read only**
mechanism:	**by reference**

Display viewport attribute list. The **attributes** argument is the address of data structure that contains longword pairs, or *doublets*. The first longword stores an attribute ID code and the second longword holds the attribute value (which can be real or integer).

The following figure describes the structure of the window attributes list.



ZK-4581-85

Only positional attributes are significant.

UIS$MOVE_WINDOW

Redefines the world coordinates of the specified display window.

FORMAT

UIS$MOVE_WINDOW *vd_id, wd_id,* x_1, y_1, x_2, y_2

RETURNS

UIS$MOVE_WINDOW signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

wd_id
See Section 18.3.2 for a description of this argument.

x_1, y_1
x_2, y_2

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

World coordinates of the new display window. The x_1 and y_1 arguments are the addresses of f_floating point numbers that define that lower-left corner of the display window. The x_2 and y_2 arguments are the addresses of f_floating point numbers that define the upper-right corner of the new display window.

DESCRIPTION

UIS$MOVE_WINDOW redefines the world coordinates of the specified display window. As a result, what is displayed in the associated display viewport can change. You can pan around a virtual display or scroll through a virtual display. If the display window rectangle changes dimensions or aspect ratio, then scaling is performed to map the new window size to the existing display viewport size.

UIS$NEW_TEXT_LINE

Moves the current text position along the actual path of text drawing to the starting margin, and then in the direction of the minor text path. Depending on the minor text path, the width or height of the character cell is used for spacing between characters and lines.

FORMAT

UIS$NEW_TEXT_LINE *vd_id, atb*

RETURNS

UIS$NEW_TEXT_LINE signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword integer that identifies an attribute block that modifies text output.

DESCRIPTION

Font, text path, character spacing, and text slope attributes influence the behavior.

UIS$PLOT

Draws a filled or unfilled point, line, or polygon depending on the number of positions specified.

FORMAT

UIS$PLOT *vd_id, atb,* x_1, y_1 [,x_2,y_2 [,...x_n,y_n]]

RETURNS

UIS$PLOT signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword integer that identifies an attribute block that modifies line style and line width or both.

x
y

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

World coordinate pair. The **x** and **y** arguments are the addresses of f_ floating point numbers that define a point in the virtual display. If the arguments are repeated to specify a second position, a line is created. You can specify up to 126 world coordinate pairs as arguments. See the Description section below for more information about this argument.

DESCRIPTION

If you specify one position, a point is drawn. If you specify more than one position, a vector is drawn between the new point and the last point. For a connected polygon to be drawn, the last vector drawn must point back to the first set of points. When you use this routine, you can specify up to 252 arguments. This would produce a maximum 126-point polygon. If you wish to specify a larger number of points in a single call, use the UIS$PLOT_ARRAY routine.

The points or lines are drawn with the line pattern and width for the attribute block, and if the fill pattern attribute is enabled for the attribute block, the enclosed area is filled with the current fill pattern. Note that it is not necessary to have a connected polygon in order to have a filled polygon. The fill pattern will go only as far as what the boundary appears to be for the particular polygon.

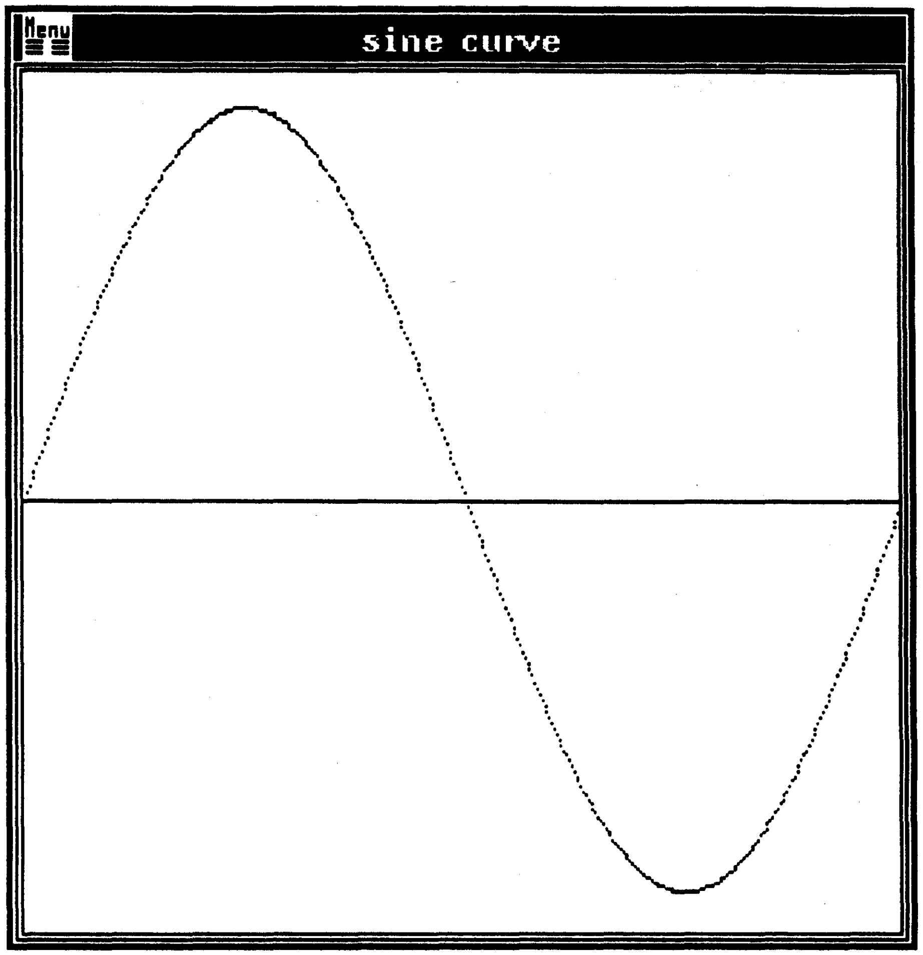
NOTE: VAX PASCAL application programs should use UIS$PLOT_ARRAY to create lines and polygons.

EXAMPLE

```
    .
    .
    .
        REAL*4 I

        VD_ID=UIS$CREATE_DISPLAY(0.0,-1.1,360.0,1.1,10.0,10.0)
        WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','SINE CURVE')
        CALL UIS$PLOT(VD_ID,0,0.0,0.0,360.0,0.0)

        DO I=1,360
        CALL UIS$PLOT(VD_ID,0,I,SIND(I))
        ENDDO
    .
    .
    .
```

This example draws a sine curve.

screen output



ZK-5257-86

UIS$PLOT_ARRAY

Draws an unfilled or filled point, line or polygon depending on the number of positions specified. This routine performs the same functions as UIS$PLOT.

FORMAT

UIS$PLOT_ARRAY *vd_id, atb, count, x_vector, y_vector*

RETURNS

UIS$PLOT_ARRAY signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword integer that identifies an attribute block that modifies line style or line width or both.

count

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Number of points. The **count** argument is the address of longword integer that denotes the number of world coordinate pairs defined in the arguments **x_vector** and **y_vector**.

x_vector
y_vector

VMS Usage:	**vector_longword_signed**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Array of *x* and *y* world coordinates. The **x_vector** argument is the address of an array of f_floating numbers whose elements are the *x* world coordinate values of points defined in the virtual display. The **y_vector** argument is the address of an array of f_floating numbers whose elements are the *y* world coordinate values of points defined in the virtual display.

DESCRIPTION

You can plot up to 65,535 points in a single call. UIS$PLOT_ARRAY is the same as UIS$PLOT except that you specify the *x* and *y* coordinates with two arrays, each of length *count* points.

UIS$POP_VIEWPORT

Pops the viewport associated with the display window to the forefront of the screen, over any other viewports that currently occlude it.

FORMAT

UIS$POP_VIEWPORT *wd_id*

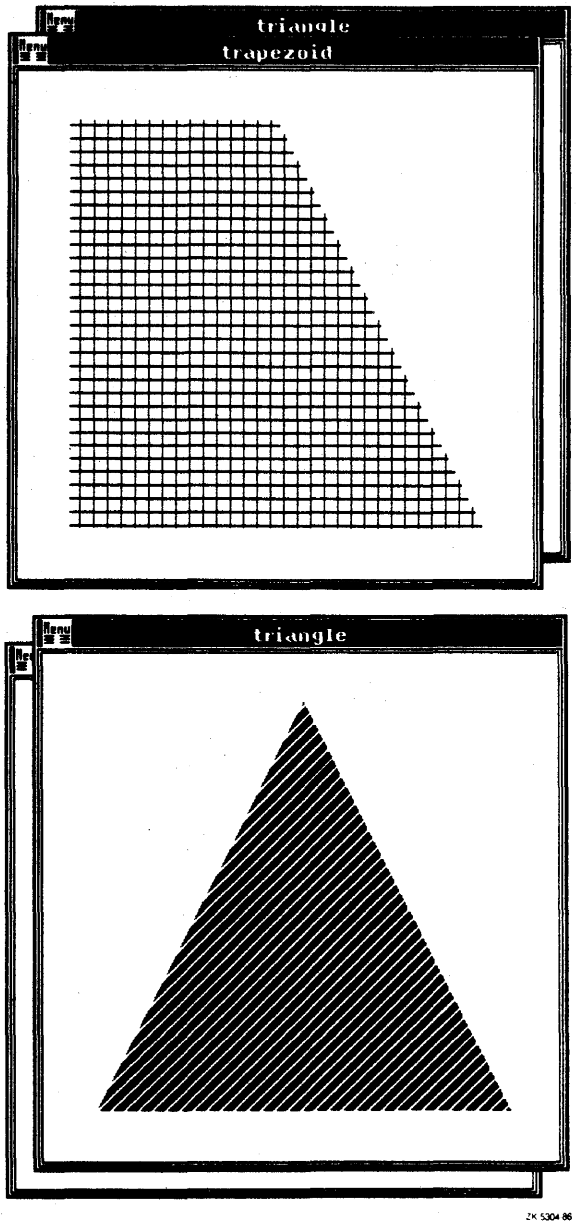
RETURNS

UIS$POP_VIEWPORT signals all errors; no condition values are returned.

ARGUMENT

wd_id
See Section 18.3.2 for a description of this argument.

screen output



UIS$PRESENT

Verifies that UIS software is installed on the system.

FORMAT

status = **UIS$PRESENT** *[major_version][,minor_version]*

RETURNS

VMS Usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword value returned in the variable *status* or R0 (VAX MACRO). A value of *1* TRUE indicates that UIS is installed on the system. Otherwise, the error status SHR$_PROD_NOTINS is returned if UIS$PRESENT is executed on a VAX/VMS system running the stub UIS shareable image. The stub shareable image is currently installed on non-VAXstation systems.

ARGUMENTS

major_version

VMS Usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**write only**
mechanism:	**by reference**

Major version number. The **major_version** argument is the address of a word that receives the major version number. For UIS Version 3.0, the major version number 3 is returned.

minor_version

VMS Usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**write only**
mechanism:	**by reference**

Minor version number. The **minor_version** argument is the address of a word that receives the minor version number. For UIS Version 3.0, the minor version number 0 is returned.

UIS$PRIVATE

Associates application-specific data with the most recently output graphic information (graphics or text) or with the specified graphic object.

FORMAT

$$\textbf{UIS\$PRIVATE} \quad \left\{ \begin{array}{l} \textit{obj\_id} \\ \textit{vd\_id} \end{array} \right\} , \textit{facnum, buffer}$$

RETURNS

UIS$PRIVATE signals all errors; no condition values are returned.

ARGUMENTS

obj_id
See Section 18.3.3 for a description of this argument.

vd_id
See Section 18.3.1 for a description of this argument.

facnum

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Facility number. The **facnum** argument is the address of a longword that identifies the creator of the private data.

Values defined with the high bit set are reserved to DIGITAL.

buffer

VMS Usage:	**vector_byte_unsigned**
type:	**byte (unsigned)**
access:	**read only**
mechanism:	**by descriptor**

Location of the private data. The **buffer** argument is a descriptor of an array of bytes. The byte array contains the private data.

DESCRIPTION

If you select a graphic item and store it in a file, the application-specific data is copied with it. If nothing is output after the beginning of a segment, the data is associated with the segment.

Many private data items can be associated with the same graphic object.

UIS$PUSH_VIEWPORT

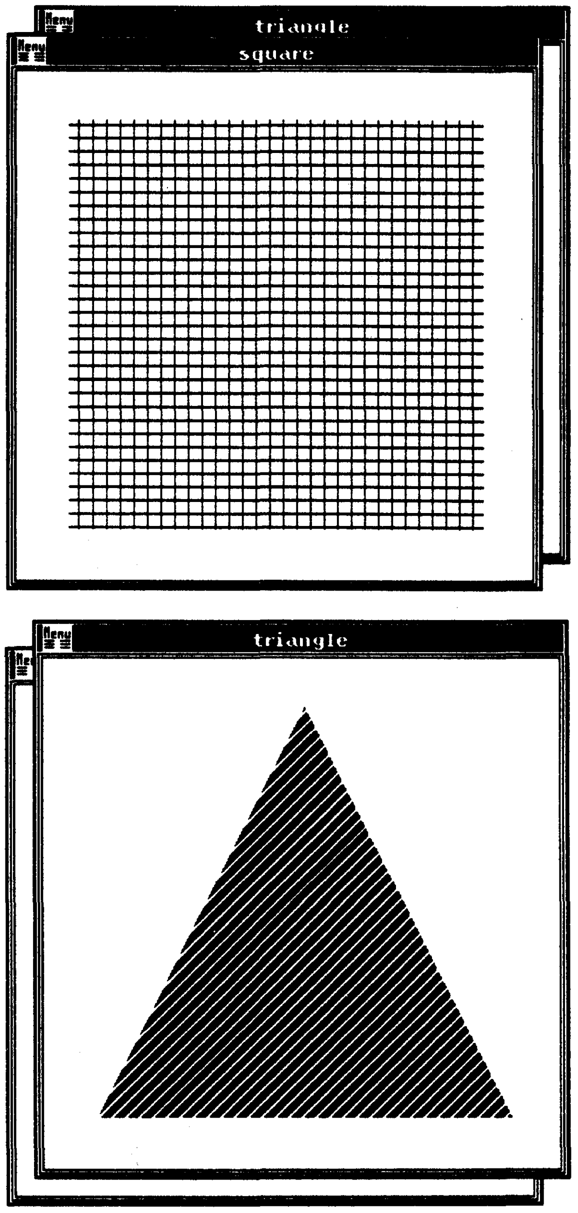
Pushes the viewport associated with the display window to the background of the screen, behind any other viewports it occludes.

FORMAT **UIS$PUSH_VIEWPORT** *wd_id*

RETURNS UIS$PUSH_VIEWPORT signals all errors; no condition values are returned.

ARGUMENTS ***wd_id***
See Section 18.3.2 for a description of this argument.

screen output



UIS$READ_CHAR

Allows an application to read a single character from the keyboard.

FORMAT

keybuf = **UIS$READ_CHAR** *kb_id [,flags]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by reference**

Longword integer returned key information in the variable *keybuf* or R0 (VAX MACRO). The *keybuf* variable is the address of a longword buffer that receives the key information. The low two bytes are the key code. The key codes are based on the codes found in the module $SMGDEF in SYS$LIBRARY:STARLET.MLB. Bit <31> is set to *1* to indicate that the key is down. For additional information about **keybuf**, see the Description section.

UIS$READ_CHAR signals all errors; no condition values are returned.

ARGUMENTS

kb_id

See Section 18.3.8 for more information about the **kb_id** argument.

flags

VMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Flags. The **flags** argument is the address of a longword mask that controls whether UIS$READ_CHAR executes immediately or until a character is received. If bit <0> is clear, UIS$READ_CHAR waits until a character is typed. If bit <0> is set and no character is currently waiting, UIS$READ_CHAR returns a value of *0*.

Specify UIS$M_NOWAIT to set bit <0> in the longword mask.

DESCRIPTION

The following table defines the bits in the high- and lower-order word.

Field	Symbol
1-16	UIS$W_KEY_CODE
28	UIS$V_KEY_SHIFT[1]
29	UIS$V_KEY_CTRL[1]
30	UIS$V_KEY_LOCK[1]
31	UIS$V_KEY_DOWN[1]

[1]This symbol is returned as SET if the corresponding key on the keyboard was down when the input event occurred.

UIS$RESIZE_WINDOW

Deletes the old display window and creates a new window. The routine reexecutes the display list of the virtual display, if it exists.

FORMAT

UIS$RESIZE_WINDOW *vd_id, wd_id [,new_abs_x, new_abs_y] [,new_width, new_height] [,new_wc_x_1, new_wc_y_1, new_wc_x_2, new_wc_y_2,]*

RETURNS

UIS$RESIZE_WINDOW signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

wd_id
See Section 18.3.2 for a description of this argument.

new_abs_x
new_abs_y

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Absolute device coordinate pair. The **new_abs_x** and **new_abs_y** arguments are the addresses of f_floating point numbers that define the location of the newly resized display viewport in centimeters.

new_width
new_height

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Width and height of the newly resized display viewport. The **width** and **height** arguments are the addresses of f_floating point numbers that define the width and height of the newly resized display viewport in centimeters.

new_wc_x_1, new_wc_y_1
new_wc_x_2, new_wc_y_2

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

World coordinates of the newly resized display window. The x_1 and y_1 arguments are the addresses of f_floating point numbers that define the location of the lower-left corner of the resized display window in world coordinates. The x_2 and y_2 arguments are the addresses of f_floating point numbers that define the location of the upper-right corner of the resized display window in world coordinates.

DESCRIPTION

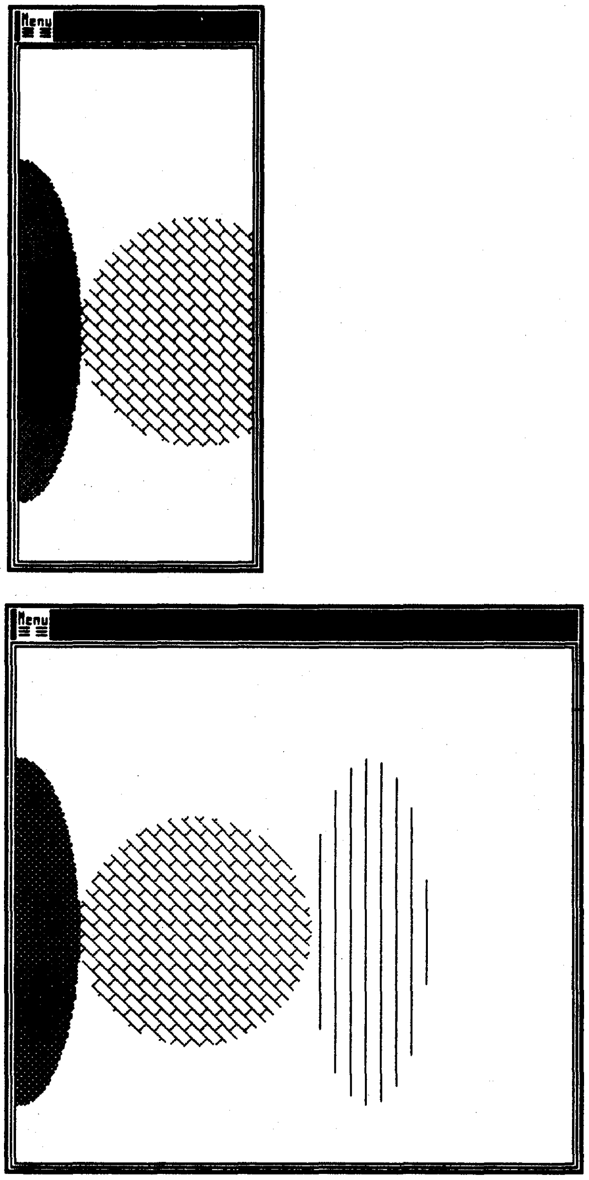
The viewport resize operation of the user interface uses UIS$RESIZE_WINDOW by default.

If UIS$RESIZE_WINDOW is called outside an AST routine, the value of all unspecified parameters defaults to those specified in UIS$CREATE_WINDOW.

If UIS$RESIZE_WINDOW is called within an AST routine, the value of all unspecified parameters defaults to the current values associated with the absolute position, dimensions, and world coordinate range of the stretchy box.

ASTs established for pointer movements, mouse button transitions, and custom cursor patterns must be reestablished to include screen area added to the window.

screen output



UIS$RESTORE_CMS_COLORS

Resets the appropriate entries in the hardware color map to the current RGB values in the color map segment.

FORMAT

UIS$RESTORE_CMS_COLORS *cms_id*

RETURNS

UIS$RESTORE_CMS_COLORS signals all errors; no condition values are returned.

ARGUMENT

cms_id

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Color map segment identifier. The **cms_id** argument is the address of a longword that uniquely identifies the color map segment. See UIS$CREATE_COLOR_MAP_SEG for more information about the **cms_id** argument.

DESCRIPTION

An application running in an unfavorable environment (where other applications are sharing hardware color map entries) can use UIS$RESTORE_CMS_COLORS to reestablish all its entries when it is the active application. Normally, this call is not required since the UIS window management software transparently handles the multiplexing of the hardware color map. If possible, the update is synchronized to the display's vertical retrace.

UIS$RGB_TO_HLS

Converts red, green, and blue (RGB) color representation values to hue, lightness, and saturation (HLS) color values.

FORMAT

UIS$RGB_TO_HLS *R, G, B, reth, retl, rets*

RETURNS

UIS$RGB_TO_HLS signals all errors; no condition values are returned.

ARGUMENTS

R

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Red value. The **R** argument is the address of a longword that defines the red color value.

G

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Green value. The **G** argument is the address of a longword that defines the green color value.

B

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Blue value. The **B** argument is the address of a longword that defines the blue color value.

reth

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Hue. The **reth** argument is the address of an f_floating point longword that receives the hue color value.

retl

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Lightness. The **retl** argument is the address of an f_floating point longword that receives the lightness value.

rets

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Saturation. The **rets** argument is the address of an f_floating point longword that receives the color saturation value.

UIS$RGB_TO_HSV

Converts color representation values of red, green, and blue (RGB) to hue, saturation, and value (HSV).

FORMAT

UIS$RGB_TO_HSV *R, G, B, reth, rets, retv*

RETURNS

UIS$RGB_TO_HSV signals all errors; no condition values are returned.

ARGUMENTS

R

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Red value. The **R** argument is the address of an f_floating number that defines the red color value.

G

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Green value. The **G** argument is the address of an f_floating number that defines the green color value.

B

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Blue value. The **B** argument is the address of an f_floating number that defines the blue color value.

reth

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Hue. The **reth** argument is the address of an f_floating longword that receives the hue value.

rets

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Saturation. The **rets** argument is the address of an f_floating longword that receives the saturation value.

retv

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Value. The **retv** argument is the address of an f_floating longword that receives the value of the color.

UIS$SET_ADDOPT_AST

Specifies execution of a user-requested AST routine whenever the Additional Options menu item is selected in the Window Options Menu.

FORMAT

UIS$SET_ADDOPT_AST *wd_id, [astadr [,astprm]]*

RETURNS

UIS$SET_ADDOPT_AST signals all errors; no condition values are returned.

ARGUMENTS

wd_id
See Section 18.3.2 for a description of this argument.

astadr

VMS Usage:	**ast_procedure**
type:	**procedure entry mask**
access:	**read only**
mechanism:	**by reference**

AST routine. The **astadr** argument is the address of a procedure entry mask of a user-supplied subroutine that is called at AST level whenever you select the Additional Options item in the Window Options Menu.

astprm
See Section 18.3.7 for a description of this argument.

DESCRIPTION

Additional options are disabled by default.

UIS$SET_ALIGNED_POSITION

Sets the current position for text output at the upper-left corner of the character cell of the next character. See UIS$GET_ALIGNED_POSITION for information about returning text alignment data.

FORMAT

UIS$SET_ALIGNED_POSITION *vd_id, atb, x, y*

RETURNS

UIS$SET_ALIGNED_POSITION signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** is the address of a longword that identifies an attribute block.

x

y

VMS Usage:	**floating_number**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

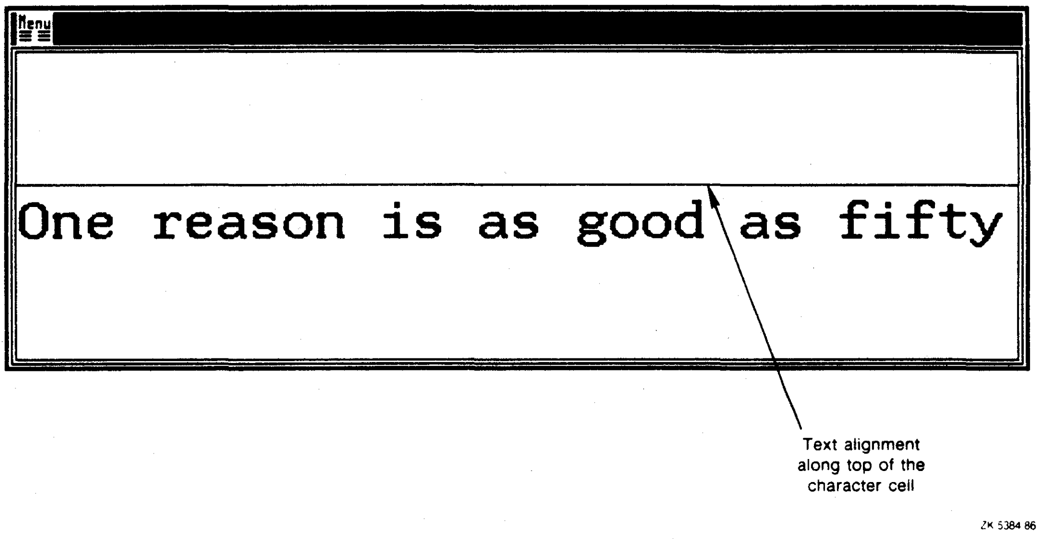
World coordinate pair. The x and y arguments are the addresses of f_ floating point numbers that define the current position for text output.

DESCRIPTION

UIS$SET_ALIGNED_POSITION is useful in applications that know the position of the upper left corner but do not know enough about the font baseline to determine the proper alignment point. The position is converted into the proper alignment point using the font specified in the given attribute block.

UIS maintains the current text position as a baseline position.

screen output



One reason is as good as fifty

UIS$SET_ARC_TYPE

Sets the current arc type used in the UIS$ELLIPSE and UIS$CIRCLE routines.

FORMAT

UIS$SET_ARC_TYPE *vd_id, iatb, oatb, arc_type*

RETURNS

UIS$SET_ARC_TYPE signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

iatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.5 for a description of this argument.

oatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.6 for a description of this argument.

arc_type

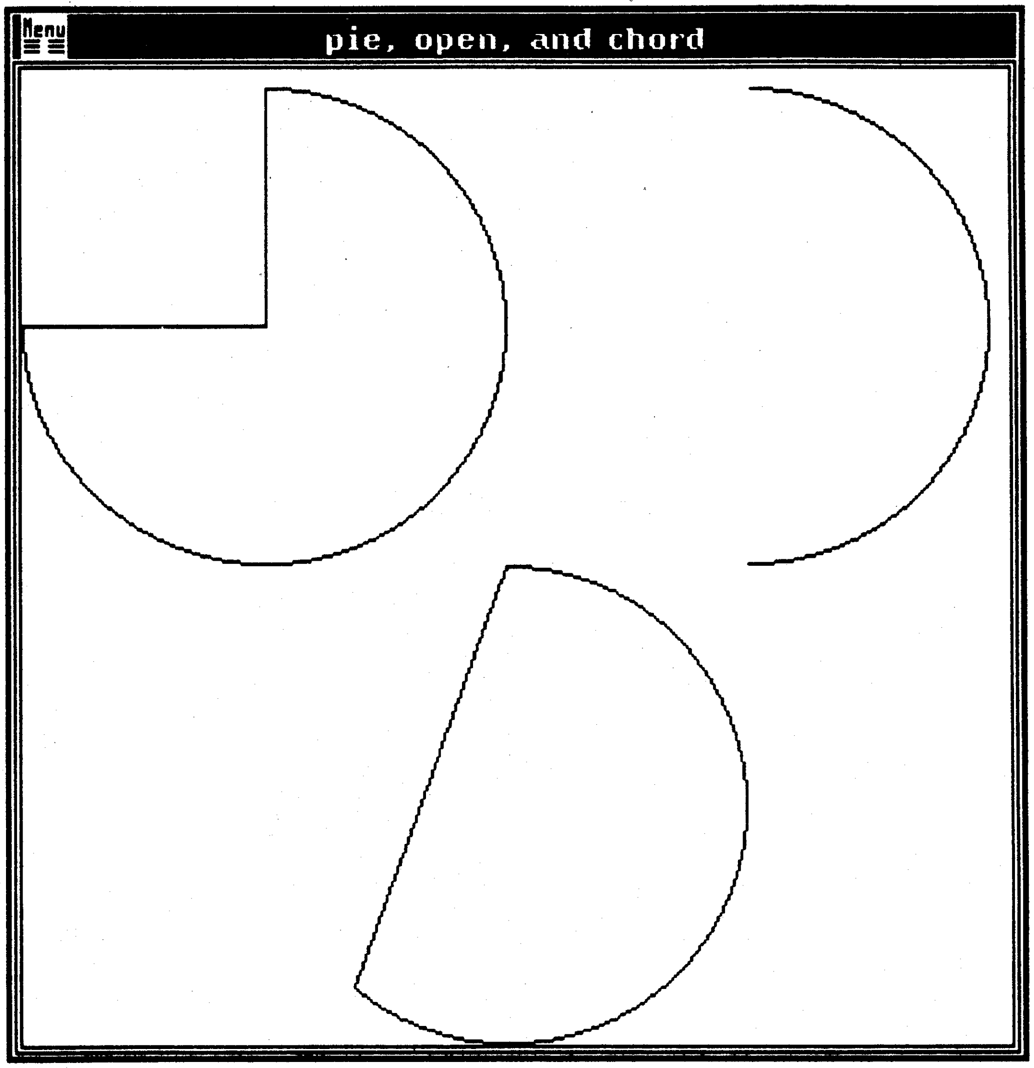
VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Arc type code. The **arc_type** argument is the address of a longword value that redefines the attribute setting of the input attribute block. Specify one of the following constants UISC_ARC_PIE, UISC_CHORD, or UIS$C_ARC_OPEN.

The following table lists symbols for arc types and their functions.

Symbol	Function
UIS$C_ARC_CHORD	Draws a line connecting the end points of the arc
UIS$C_ARC_OPEN	Does not draw any lines (default)
UIS$C_ARC_PIE	Draws radii to the end points of the arc

screen output



ZK-5256-86

UIS$SET_BACKGROUND_INDEX

Sets the background color index for text and graphics output.

FORMAT

UIS$SET_BACKGROUND_INDEX *vd_id, iatb, oatb, index*

RETURNS

UIS$SET_BACKGROUND_INDEX signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

iatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.5 for a description of this argument.

oatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.6 for a description of this argument.

index

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Color map index. The **index** argument is the address of a longword that specifies the color map index. If the index exceeds the maximum index for the associated color map, an error is signaled.

UIS$SET_BUTTON_AST

Allows an application to find out when a button on the pointing device is depressed or released in a given rectangle within a display viewport.

FORMAT

UIS$SET_BUTTON_AST *vd_id, wd_id [,astadr [,astprm] ,keybuf] [, x_1, y_1, x_2, y_2]*

RETURNS

UIS$SET_BUTTON_AST signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

wd_id
See Section 18.3.2 for a description of this argument.

astadr

VMS Usage:	**ast_procedure**
type:	**procedure entry mask**
access:	**read only**
mechanism:	**by reference**

AST routine. The **astadr** argument is the address of an entry mask to a procedure that is called at AST level whenever you depress or release a pointer button. To cancel the AST-enabling request of UIS$SET_BUTTON_AST, specify *0* in the **astadr** argument.

astprm
See Section 18.3.7 for a description of this argument.

keybuf

VMS Usage:	**address**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by reference**

Key buffer. The **keybuf** argument is the address of a longword buffer that receives button information whenever a pointer button is depressed or released. The low two bytes are the key code. The buttons are located on the left, center, and right of the device and are defined as UIS$C_POINTER_BUTTON_1, UIS$C_POINTER_BUTTON_2, UIS$C_POINTER_BUTTON_3, and UIS$C_POINTER_BUTTON_4, respectively.

The bit <31> is set to *1* if the button has been pressed and to *0* if the button has been released. The buffer is not overwritten with subsequent button transitions until the AST routine completes.

The following table defines the bits in the high- and lower-order word.

Field	Symbol
1-16	UIS$W_KEY_CODE
28	UIS$V_KEY_SHIFT[1]
29	UIS$V_KEY_CTRL[1]
30	UIS$V_KEY_LOCK[1]
31	UIS$V_KEY_DOWN[1]

[1]This symbol is returned as SET if the corresponding key on the keyboard was down when the input event occurred.

x_1, y_1
x_2, y_2

VMS Usage: **floating_point**
type: **f_floating**
access: **read only**
mechanism: **by reference**

World coordinates of a rectangle in the display window. The x_1 and y_1 arguments are the addresses of f_floating point numbers that define the lower-left corner of a rectangle in the display window. The x_2 and y_2 arguments are the addresses of f_floating point numbers that define the upper-right corner of a rectangle in the display window. If no rectangle is specified, the entire display window is assumed.

DESCRIPTION

This function can be called any number of times for different rectangles within the same display window or many display windows.

To disable UIS$SET_BUTTON_AST, omit the **astadr**, **astprm**, and **keybuf** arguments.

Pointer Region Priorities

UIS pointer regions are placed on the VAXstation screen in the order in which they are created. Therefore, if you create two overlapping viewports, and then use UIS$SET_POINTER_PATTERN, UIS$SET_BUTTON_AST, or UIS$SET_POINTER_AST to define different pointer patterns for each viewport, the *correctness* of the result will depend on the order in which you both created the viewports and defined the cursor regions. For example, if you create the viewports and define the cursor patterns in the following manner, the viewport 1 cursor pattern will have a higher priority than viewport 2 cursor pattern in the overlapping region.

1. Create viewport 1
2. Create overlapping viewport 2
3. Define viewport 2 cursor pattern
4. Define viewport 1 cursor pattern

The preceding example causes the unexpected result that the viewport 1 cursor pattern will take priority over the viewport 2 cursor pattern in the overlapping region. This problem can be corrected by creating the viewports and defining the cursor patterns in the same order. To correct the problem, create the viewports and define cursor patterns in the following order:

1 Create viewport 1
2 Define viewport 1 cursor pattern
3 Create overlapping viewport 2
4 Define viewport 2 cursor pattern

The solution is for either UIS or your application to always pop the viewport before defining the cursor region for it.

UIS$SET_CHAR_ROTATION

Sets the angle of character rotation, measured counterclockwise relative to the actual path of text drawing.

FORMAT

UIS$SET_CHAR_ROTATION *vd_id ,iatb ,oatb ,angle*

RETURNS

UIS$SET_CHAR_ROTATION signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

iatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.5 for a description of this argument.

oatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.6 for a description of this argument.

angle

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Angle of character rotation. The **angle** argument is the address of an f_floating point number that defines the angle of character rotation in degrees counterclockwise about the baseline point relative to the actual path of text drawing.

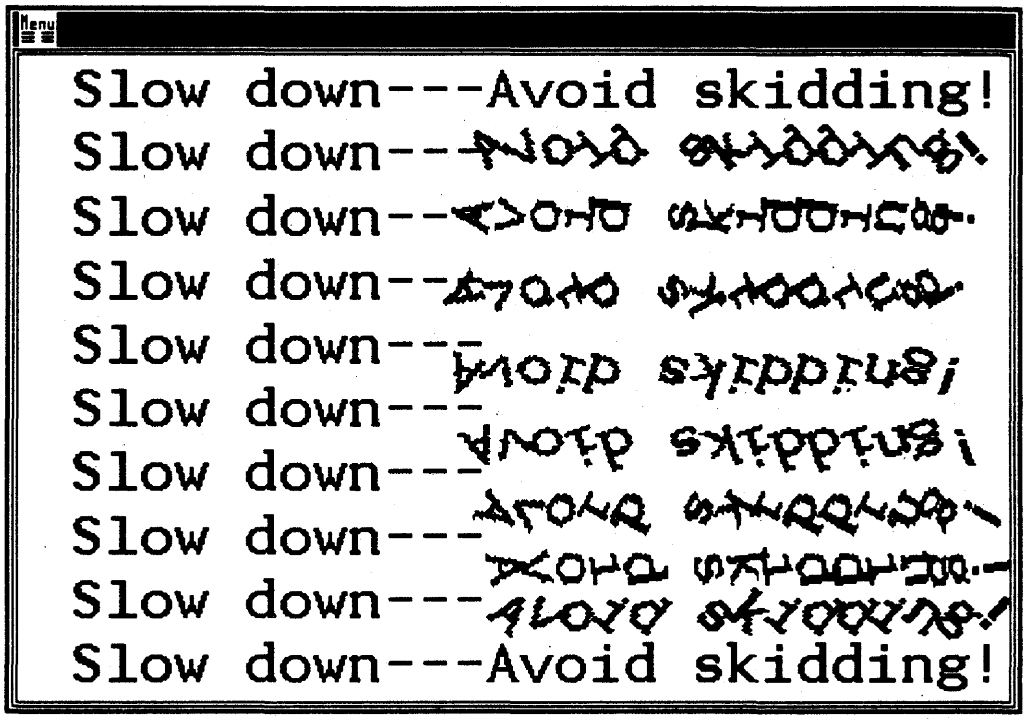
DESCRIPTION

For example, an angle of 0 degrees (the default) means that the character's baseline vector and the actual path of text drawing form an angle of 0 degrees.

EXAMPLE

```
.
.
.
CALL UIS$SET_FONT(VD_ID,0,1,'MY_FONT_5')
CALLUIS$SET_TEXT_MARGINS(VD_ID,1,1,1.0,20.0,18.0)
CALL UIS$SET_ALIGNED_POSITION(VD_ID,1,1.0,20.0)

DO I=0,360,40
CALL UIS$TEXT(VD_ID,1,'Slow down---')
CALL UIS$SET_CHAR_ROTATION(VD_ID,1,2,FLOAT(I))
CALL UIS$TEXT(VD_ID,2,'Avoid skidding!')
CALL UIS$NEW_TEXT_LINE(VD_ID,2)
ENDDO
.
.
.
```

screen output



```
Slow down---Avoid skidding!
Slow down---
Slow down---
Slow down---
Slow down---
Slow down---
Slow down---
Slow down---
Slow down---
Slow down---Avoid skidding!
```

ZK 5258 86

UIS$SET_CHAR_SIZE

Sets the world coordinate size of a specified character.

FORMAT

UIS$SET_CHAR_SIZE *vd_id, iatb, oatb [,char] [,width] [,height]*

RETURNS

UIS$SET_CHAR_SIZE signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

iatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.5 for a description of this argument.

oatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.6 for a description of this argument.

char

VMS Usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

Single character. The **char** argument is the address of a descriptor of a single character.

If **char** is not specified, the widest character in the font is chosen.

width
height

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Character width and height. The **width** argument is the address of an f_floating point longword that defines the character width in world coordinates. The **height** argument is the address of an f_floating point longword that defines the character height in world coordinates.

See DESCRIPTION section for information about omitting the **width** argument and **height** argument.

DESCRIPTION

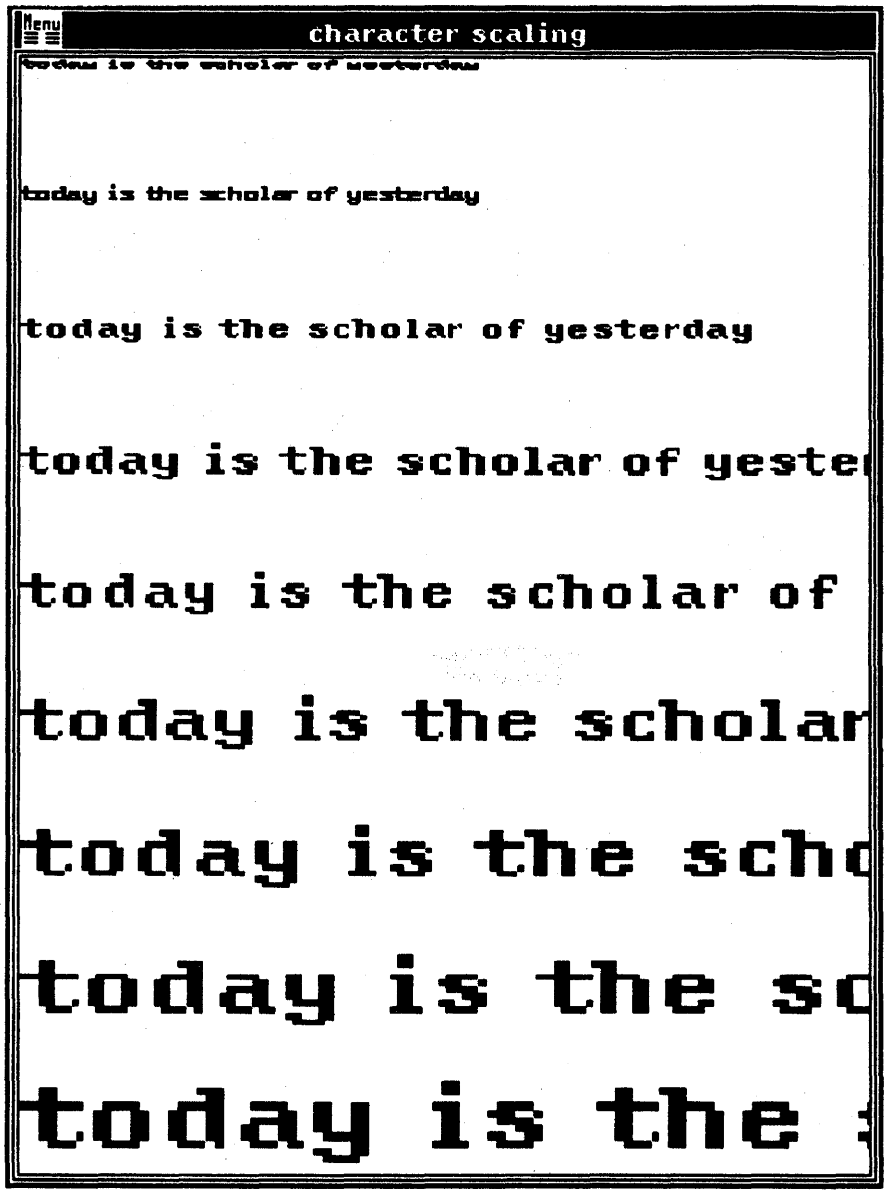
To disable character scaling, omit all of the following arguments: **char**, **width**, and **height**.

To scale characters to their nominal size as specified in the font, do not specify **width** and **height**. Scaling is only visible when you use a window that does not have the same aspect ratio as the virtual display. The particular character you specify in the argument **char** makes no difference in this case.

If you specify either **width** or **height** only, characters are scaled to the size you specify and in the direction you specify. In the unspecified direction, characters are scaled to maintain the same ratio of width and height as the unscaled characters.

Note that if you use this routine, you will not change the size of a single character only. Rather, all the characters in the font will be scaled to the correct proportion.

screen output



```
today is the scholar of yesterday
today is the scholar of yesterday
today is the scholar of yesterday
today is the scholar of yeste
today is the scholar of
today is the scholar
today is the sch
today is the sc
today is the
```

ZK-5456-86

UIS$SET_CHAR_SLANT

Sets the character slant angle.

FORMAT

UIS$SET_CHAR_SLANT *vd_id, iatb, oatb, angle*

RETURNS

UIS$SET_CHAR_SLANT signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

iatb

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.5 for a description of this argument.

oatb

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.6 for a description of this argument.

angle

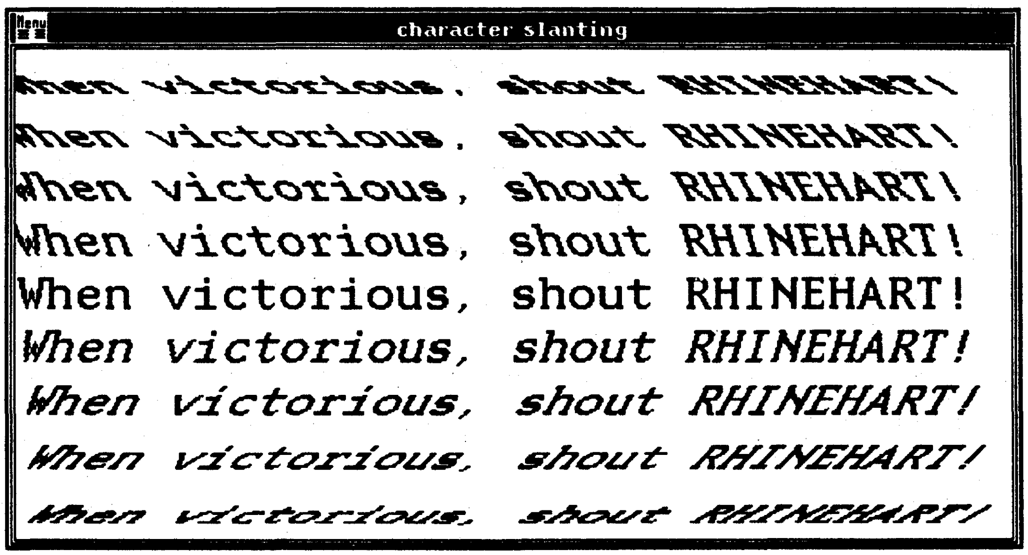
VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Angle of character slant. The **angle** argument is the address of an f_floating point number that defines the angle of character slant in degrees.

The character slant angle refers to an angle formed by the character's up vector and baseline vector.

For example, 0 degrees (the default) indicates that the character up vector is perpendicular to the baseline vector, and the character is not slanted. A counterclockwise movement from 0 degrees produces a negative angle of character slant. A clockwise movement from 0 degrees produces a positive angle of character slant.

screen output



```
character slanting

When victorious, shout RHINEHART!
When victorious, shout RHINEHART!
When victorious, shout RHINEHART!
When victorious, shout RHINEHART!
When victorious, shout RHINEHART!
When victorious, shout RHINEHART!
When victorious, shout RHINEHART!
When victorious, shout RHINEHART!
When victorious, shout RHINEHART!
```

ZK-5455-86

UIS$SET_CHAR_SPACING

Sets the attribute that controls the amount of additional spacing between text characters (*x* factor) and between text lines (*y* factor) when the UIS$NEW_LINE_TEXT routine is used.

FORMAT

UIS$SET_CHAR_SPACING *vd_id, iatb, oatb, dx, dy*

RETURNS

UIS$SET_CHAR_SPACING signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

iatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.5 for a description of this argument.

You can specify either the attribute block 0 or a previously modified attribute block.

oatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Output attribute block number. The **oatb** argument is the address of a longword value that identifies the newly modified attribute block that controls the spacing between characters.

dx
dy

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Additional *x* and *y* spacing factor. The **dx** argument is the address of an f_floating point longword value that defines the x spacing factor. If this argument is 0.0, no additional spacing is performed. The **dy** is the argument of an f_floating point longword value that defines the y spacing factor. If this argument is 0.0, no additional spacing is performed. Negative values are allowed, and characters can overlap.

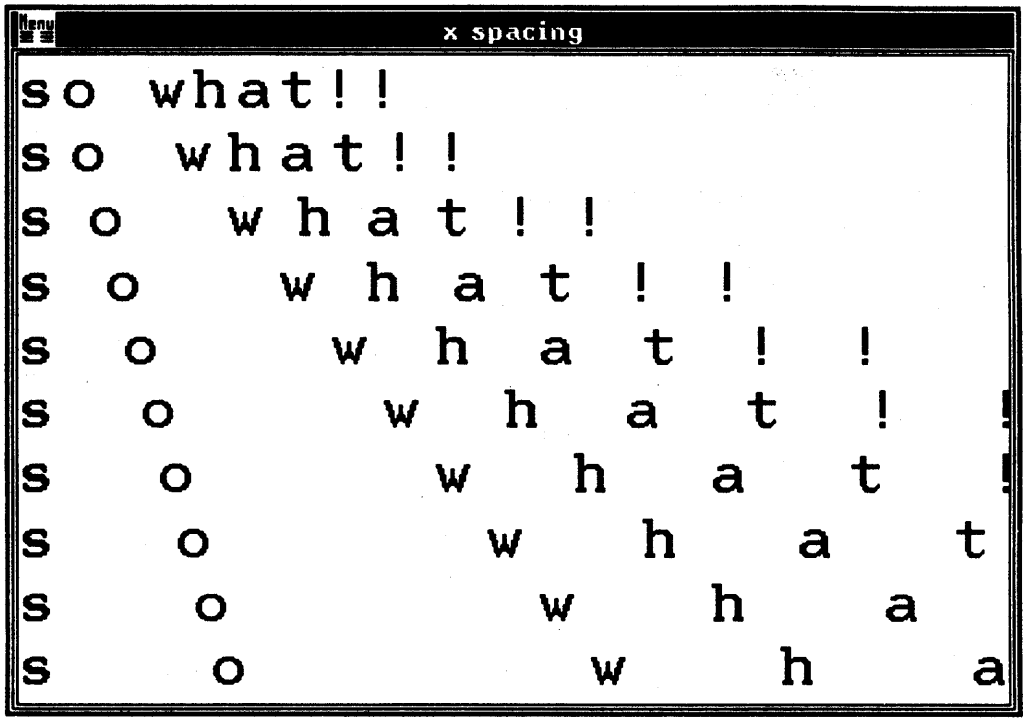
DESCRIPTION

The values of the *x* and *y* factors are multiplied by the width or height of the character, and the resulting value is used as the additional spacing distance.

Proportionally spaced characters maintain their appropriate spacing.

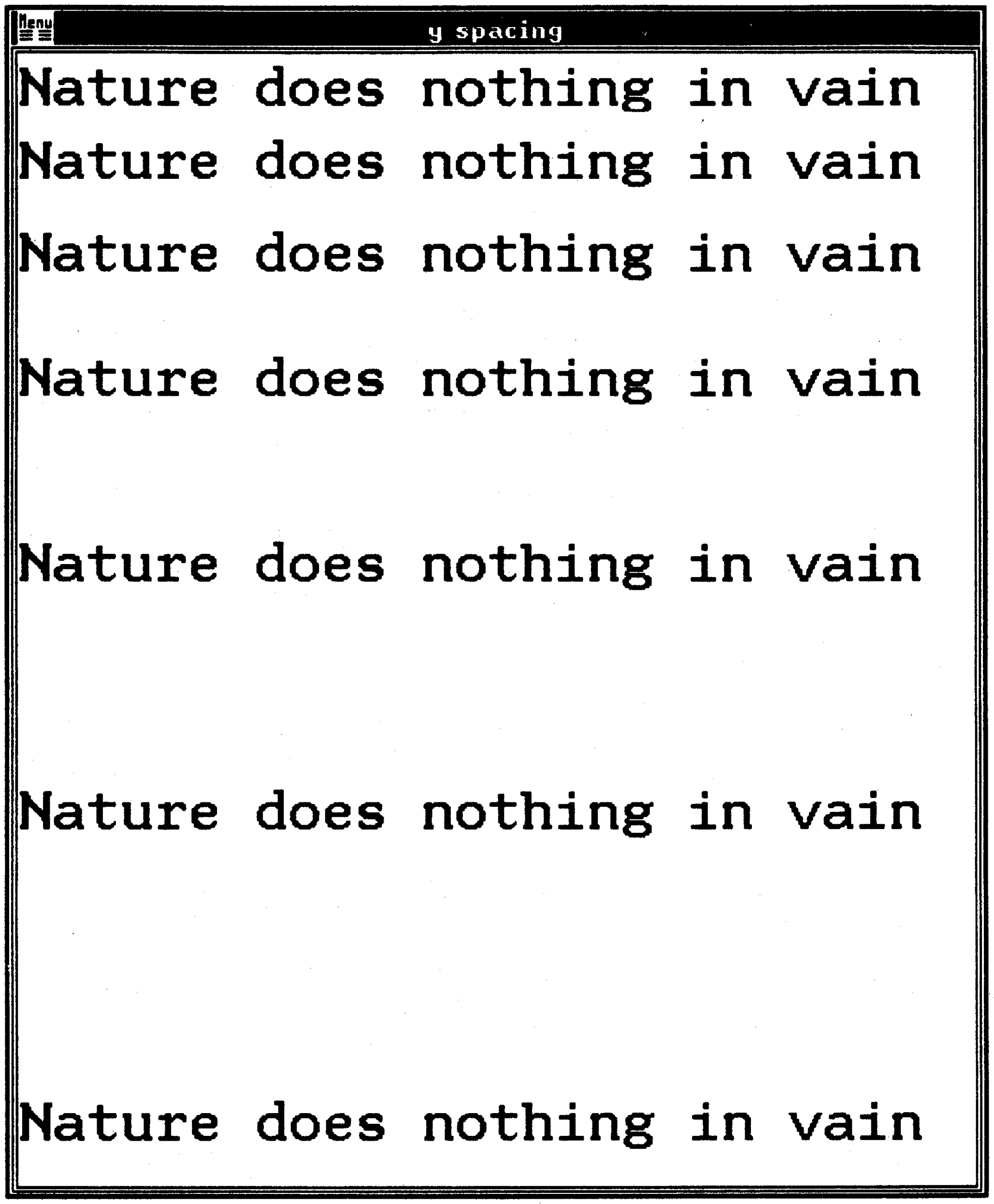
The default is no extra spacing.

screen output



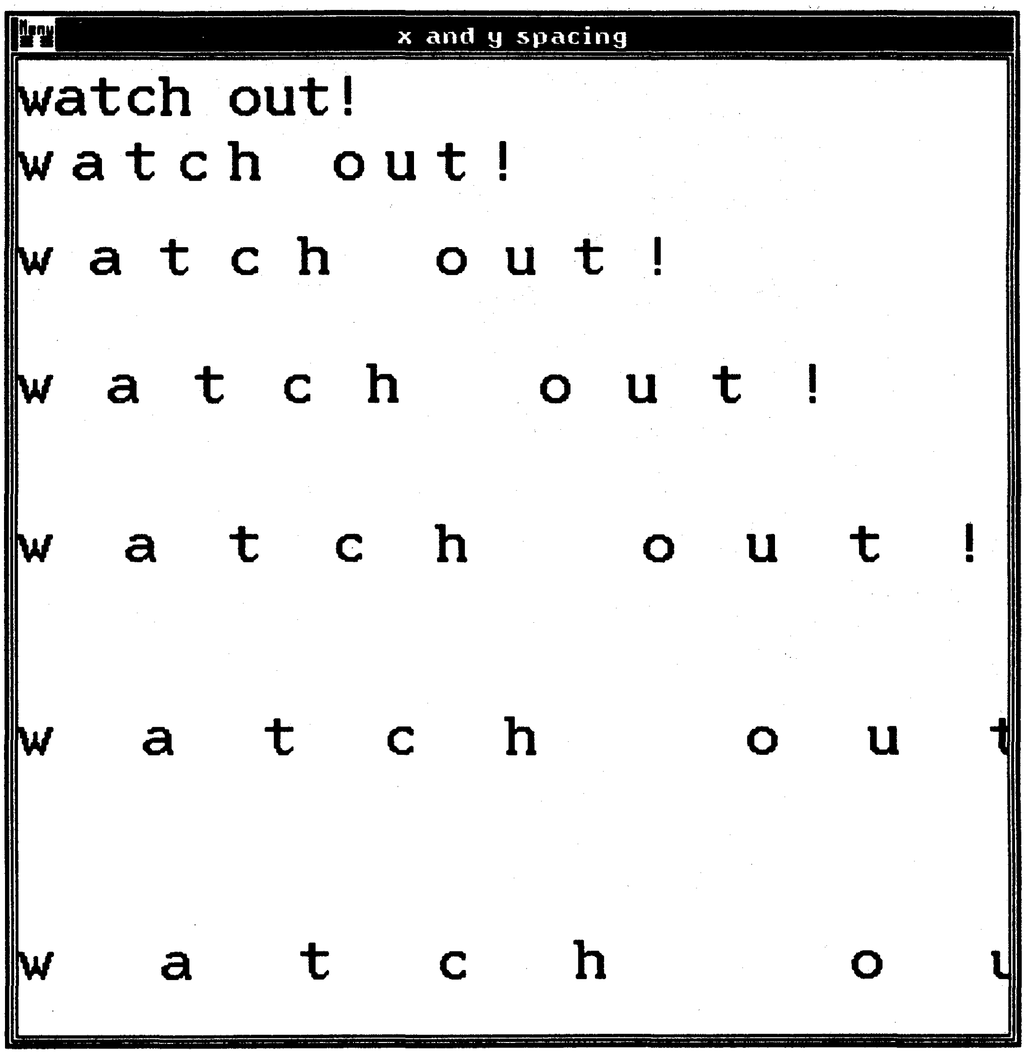
```
so what!!
s o   w h a t ! !
s  o     w  h  a  t  !  !
s   o       w   h   a   t   !   !
s    o         w    h    a    t    !    !
s     o           w     h     a     t     !     !
s      o             w      h      a      t      !
s       o               w       h       a       t
s        o                 w        h        a
s         o                   w         h         a
```

ZK 5254 86



```
Nature does nothing in vain
Nature does nothing in vain
Nature does nothing in vain
Nature does nothing in vain
Nature does nothing in vain
Nature does nothing in vain
Nature does nothing in vain
```

ZK 5253 86



```
watch out!
w a t c h  o u t !
w  a  t  c  h    o  u  t  !
w   a   t   c   h      o   u   t   !
w    a    t    c    h        o    u    t    !
w     a     t     c     h          o     u     t
w      a      t      c      h            o      u
```

ZK-5252-86

UIS$SET_CLIP

Sets a clipping rectangle in the virtual display and enables clipping for this attribute block.

FORMAT

UIS$SET_CLIP *vd_id, iatb, oatb [,x_1, y_1, x_2, y_2]*

RETURNS

UIS$SET_CLIP signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

iatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.5 for a description of this argument.

Specify either the attribute block 0 or a previously modified attribute block.

oatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.6 for a description of this argument.

x_1, y_1
x_2, y_2

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

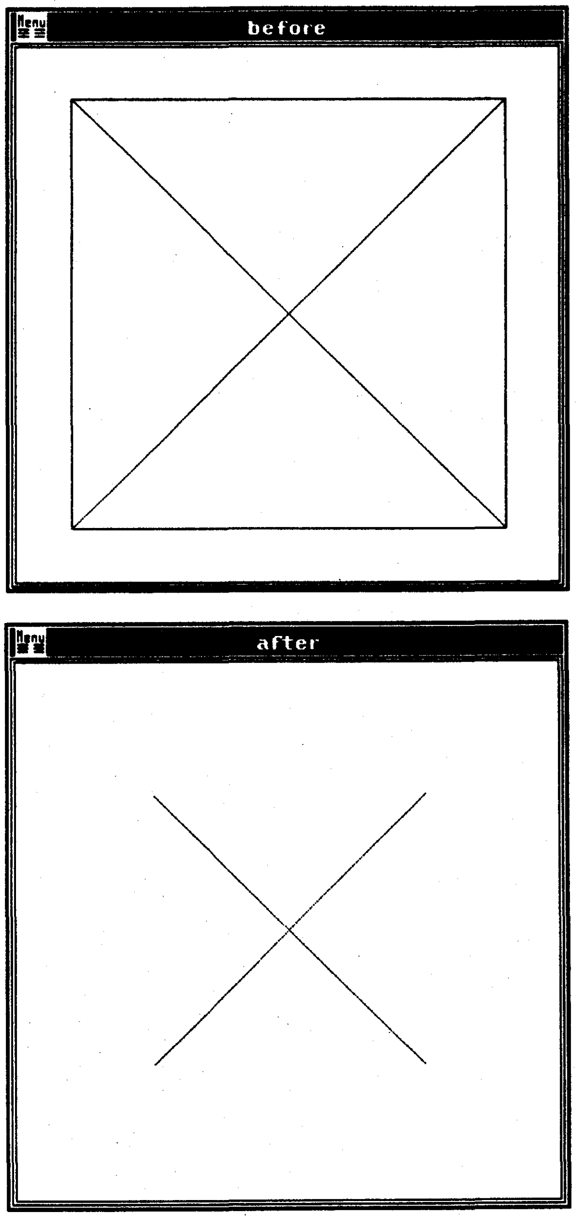
World coordinates of the clipping rectangle. The x_1 and y_1 arguments are the addresses of f_floating point numbers that define the lower left corner of the clipping rectangle in world coordinates. The x_2 and y_2 arguments are the addresses of f_floating point numbers that define the upper right corner of the clipping rectangle in world coordinates. Only graphic objects and portions of graphic objects drawn **within** the clipping rectangle are seen.

If the world coordinates of the clipping rectangle corners are not specified, then clipping is disabled for this attribute block.

EXAMPLE

```
        .
        .
        .
             WD_ID1=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','AFTER')
             CALL UIS$ERASE(VD_ID)
             CALL UIS$SET_CLIP(VD_ID,0,1,5.0,5.0,15.0,15.0)

             CALL UIS$PLOT(VD_ID,1,2.0,2.0,18.0,2.0,18.0,18.0,2.0,18.0,
             2       2.0,2.0)
             CALL UIS$PLOT(VD_ID,1,2.0,2.0,18.0,18.0,)
             CALL UIS$PLOT(VD_ID,1,2.0,18.0,18.0,2.0)
        .
        .
        .
```

screen output



UIS$SET_CLOSE_AST

Specifies a user-requested AST routine to be executed when the "Delete" menu item is selected in the Window Options Menu.

FORMAT

UIS$SET_CLOSE_AST *wd_id [,astadr [,astprm]]*

RETURNS

UIS$SET_CLOSE_AST signals all errors; no condition values are returned.

ARGUMENTS

wd_id
See Section 18.3.2 for a description of this argument.

astadr

VMS Usage:	**ast_procedure**
type:	**procedure entry mask**
access:	**read only**
mechanism:	**by reference**

AST routine. The **astadr** argument is the address of a procedure entry mask of a user-supplied subroutine that is called at AST level whenever you select the delete item in the Window Options Menu. See the Description section for more information about disabling close AST routines.

astprm
See Section 18.3.7 for more information on this argument.

DESCRIPTION

Typically, you call UIS$SET_CLOSE_AST to override the default window closing behavior. If you do not specify a CLOSE AST routine, UIS calls UIS$CLOSE_WINDOW by default. If this behavior is not sufficient, the application program can use its own close routine to call UIS$SET_CLOSE_AST.

If the application has previously enabled close ASTs but no longer needs to do special tasks when closing a window, it can specify UIS$CLOSE_WINDOW as the **astadr** parameter to reenable the default UIS action.

You can completely disable the ability to close a window in any of the following ways:

- Specify *0* in the **astadr** argument
- Specify only the **wd_id** argument.
- Omit the **astadr** and **astprm** arguments.

When window closing is disabled, the "Delete" menu item in the Window Options Menu changes from boldface to lightface.

To reenable the default window closing behavior, specify UIS$C_DEFAULT_CLOSE as the **astadr** argument in a subsequent call to UIS$SET_CLOSE_AST.

UIS$SET_COLOR

Sets a single entry in the virtual color map associated with the virtual display. The color map entry is an RGB value for a specific color.

FORMAT

UIS$SET_COLOR *vd_id, index, R, G, B*

RETURNS

UIS$SET_COLOR signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

index

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Color map index. The **index** argument is the address of a longword value that identifies an entry in the color map. If the index exceeds the maximum index for the associated color map, an error is signaled.

R

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Red value. The **R** argument is the address of an f_floating point number that defines the red value. The red value is in the range of *0.0* to *1.0*, inclusive.

G

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Green value. The **G** argument is the address of an f_floating point number that defines the green value. The green value is in the range of *0.0* to *1.0*, inclusive.

B

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Blue value. The **B** argument is the address of an f_floating point number that defines the blue value. The blue value is in the range of *0.0* to *1.0*, inclusive.

DESCRIPTION

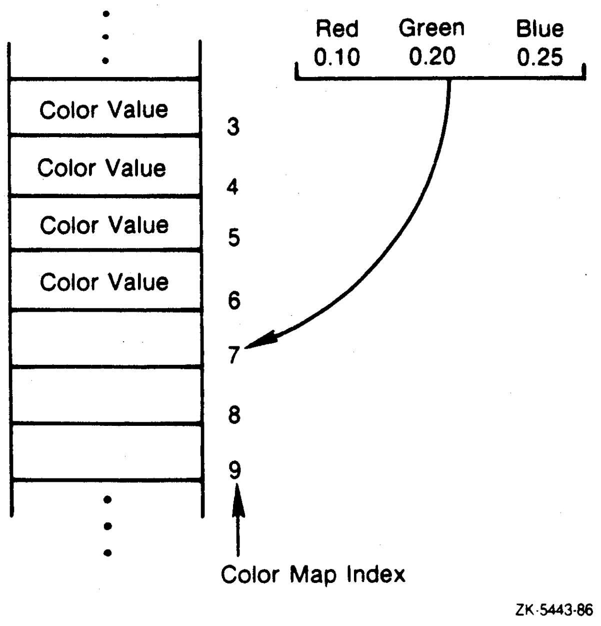
To maximize compatibility between monochrome and color display devices, UIS$SET_COLOR performs an internal transformation of the red, green, and blue values when the actual workstation display is monochromatic.

A single intensity value in the range of *0.0* to *1.0* is derived using the following formula.

```
I = (0.30*R) + (0.59*G) + (0.11*B)
```

On monochrome systems, this derived intensity value is then compared to *0.5*. If the value is greater than or equal to *0.5*, then white pixels are written. Otherwise, black pixels are written.

illustration



UIS$SET_COLORS

Sets more than one color entry in the virtual color map.

FORMAT

UIS$SET_COLORS *vd_id, index, count, r_vector, g_vector, b_vector*

RETURNS

UIS$SET_COLORS signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

index

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Starting color map index. The **index** argument is the address of a longword that defines the starting index in the virtual color map.

If the index exceeds the maximum index for the virtual color map, an error is signaled.

count

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Number of indices. The **count** argument is the address of a longword that contains the number of indices including the starting index of the color map. If the count exceeds the maximum number of virtual color map entries, an error is signaled.

r_vector

VMS Usage:	**vector_longword_signed**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Red values. The **r_vector** argument is the address of an array of f_floating point numbers that define the red values.

g_vector

VMS Usage:	**vector_longword_signed**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Green values. The **g_vector** argument is the address of an array of f_floating point numbers that define the green values.

b_vector

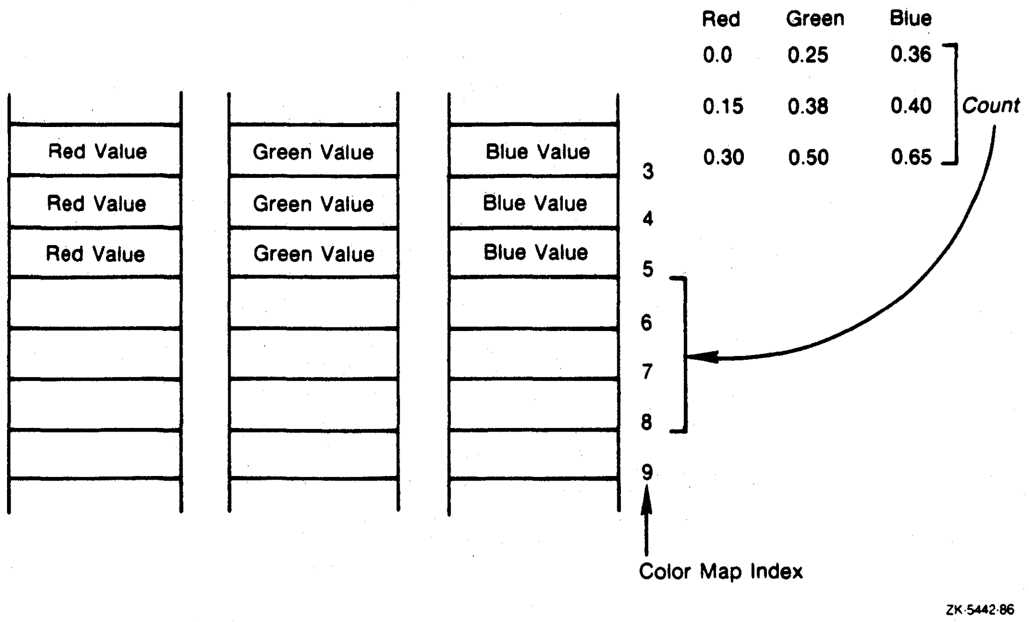
VMS Usage:	**vector_longword_signed**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Blue values. The **b_vector** argument is the address of an array of f_floating point numbers that define the blue values.

DESCRIPTION

On color and intensity systems, color map updates of greater than approximately 80 entries cause visible screen disturbance, which appears as a black bar across the top inch of the display screen. This anomaly is caused by a hardware restriction that precludes large lookup table updates within the vertical blanking interval of the raster scan.

illustration



UIS$SET_EXPAND_ICON_AST

Specifies a user-requested AST routine to be executed whenever an icon is to be replaced with its associated display viewport.

FORMAT

UIS$SET_EXPAND_ICON_AST *wd_id [,astadr [,astprm]]*

RETURNS

UIS$SET_EXPAND_ICON_AST signals all errors; no condition values are returned.

ARGUMENTS

wd_id
See Section 18.3.2 for a description of this argument.

astadr

VMS Usage:	**ast_procedure**
type:	**procedure entry mask**
access:	**read only**
mechanism:	**by reference**

AST routine. The **astadr** argument is the entry mask address of a user-written procedure called at AST level whenever you select the Expand Icon item in the Window Options Menu.

To cancel the AST-enabling request of UIS$SET_EXPAND_ICON_AST, specify *0* in the **astadr** argument.

astprm
See Section 18.3.7 for more information on this argument.

DESCRIPTION

To disable the user interface for replacing an icon with a display viewport, call UIS$SET_EXPAND_ICON_AST with the **wd_id** argument only.

To reenable the default behavior of UIS$SET_EXPAND_ICON_AST, specify the constant UIS$C_DEFAULT_EXPAND_ICON as the **astadr** argument or specify only the **wd_id** argument.

UIS$SET_FILL_PATTERN

Sets the current fill pattern used in area fill operations.

FORMAT **UIS$SET_FILL_PATTERN** *vd_id, iatb, oatb [,index]*

RETURNS UIS$SET_FILL_PATTERN signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

iatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.5 for a description of this argument.

You can specify either the attribute block 0 or a previously modified attribute block.

oatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.6 for a description of this argument.

Identifies the newly modified attribute block that controls the fill pattern.

index

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Index of the fill pattern in the current font. The **index** argument is the address of a longword value that identifies a character glyph in the current font. The value specified in the **index** argument modifies the current fill pattern index specified in the input attribute block.

If the **index** argument is not specified, fill patterns are disabled.

DESCRIPTION

The fill pattern is expressed as a character glyph in the font currently associated with the same attribute block. Several font files are usually reserved to store fill patterns (rasters). At present, UIS does not support fill patterns greater than 32 bits wide.

UIS provides a font file containing a variety of fill patterns. This font file is referenced by UIS$FILL_PATTERNS. Entries in the UIS$FILL_PATTERNS font are symbolically referenced by the symbols PATT$C_xxx.

To get a listing of all fill pattern symbols available to application programs, see 6.6 for a list of symbol definition files.

Refer to Appendix Appendix D for illustrations of each UIS fill pattern.

EXAMPLE

```
  .
  .
  .
 PROGRAM  FILL
 IMPLICIT INTEGER(A-Z)
C
 INCLUDE 'SYS$LIBRARY:UISENTRY'
 INCLUDE 'SYS$LIBRARY:UISUSRDEF'

 VD_ID=UIS$CREATE_DISPLAY(-5.0,-5.0,50.0,50.0,20.0,20.0)
 WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','FILL')
C  create an unfilled polygon with 4 vertices
 CALL UIS$PLOT (VD_ID,0,   2.0, 2.0,
 1    18.0, 2.0,
 1      18.0,18.0,
 1     2.0,18.0)

C  create an unfilled polygon with 5 vertices
 CALL UIS$PLOT (VD_ID,0,   22.0, 2.0,
 1     38.0, 2.0,
 1     38.0,18.0,
 1     22.0,18.0,
 1     22.0, 2.0)
C  set up polygons for fill patterns
 CALL UIS$SET_FONT(VD_ID,0,1,'UIS$FILL_PATTERNS')
C       set font to UIS$FILL_PATTERNS
C
 CALL UIS$SET_FILL_PATTERN(VD_ID,1,1,PATT$C_UPDIAG1_7)
C                      ATB #1
C
 CALL UIS$SET_FILL_PATTERN(VD_ID,1,2,PATT$C_FOREGROUND)
C                      ATB #2
C
C  create a filled polygon with 4 vertices
 CALL UIS$PLOT (VD_ID,1,   2.0,22.0,
 1    18.0,22.0,
 1      18.0,38.0,
 1     2.0,38.0)
C                      ATB #1
C
```

```
C  create a filled polygon with 5 vertices
 CALL UIS$PLOT (VD_ID,2,   22.0,22.0,
 1      38.0,22.0,
 1      38.0,38.0,
 1      22.0,38.0,
 1      22.0,22.0)
C                        ATB #2
C
 PAUSE
 END
   .
   .
   .
```

The preceding example draws four squares (polygons).

- An open-ended square
- A closed square
- A square with a diagonal FILL pattern
- A square with the foreground FILL pattern.

The squares on the left (see next page) are each drawn with four vertices. Note that the fill pattern goes only as far as the boundary of the four vertices already drawn. The squares on the right are each drawn with five vertices. Note that despite the five vertices, the square is still filled as in the case of the four-vertex, unconnected square.

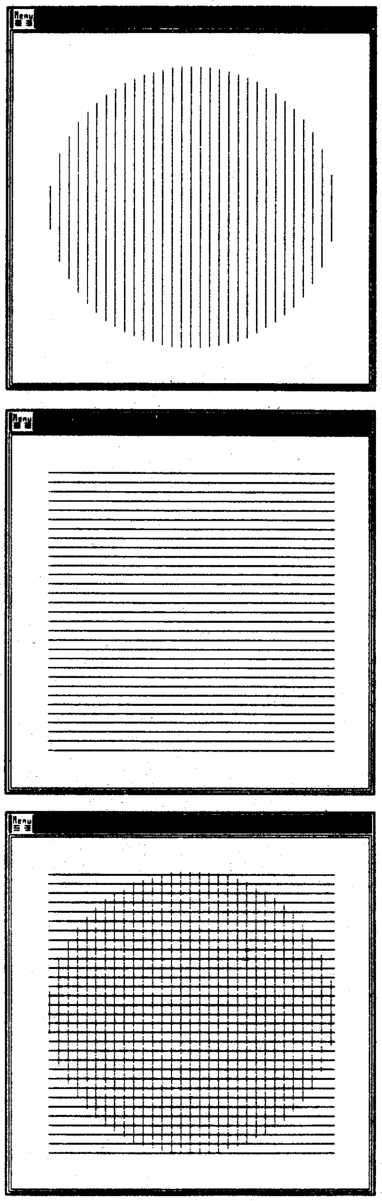
To enable fill patterns for a single graphic object, you must complete the following two-step process:

1 Modify the font attribute specifying the fill pattern file in SYS$FONT. Use the logical name UIS$FILL_PATTERNS.

2 Modify the fill pattern file specifying the fill pattern to be used.

Note the different ATB (attribute block) associated with the two filled polygons.

screen output



ZK 5307 86

UIS$SET_FONT

Specifies the fonts to be used in text drawing (UIS$TEXT) and area filling (UIS$PLOT).

FORMAT

UIS$SET_FONT *vd_id, iatb, oatb, font_id*

RETURNS

UIS$SET_FONT signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

iatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.5 for a description of this argument.

The font attribute in the input attribute block is modified to reflect the new font file specified in the **font_id** argument. Specify either the attribute block 0 or a previously modified attribute block.

oatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.6 for a description of this argument.

font_id

VMS Usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

Font file name string. The **font_id** argument is the address of a character string descriptor pointing to a file specification that identifies the desired font. System font files are located in the SYS$FONT directory. Fonts should be specified using only the file name. You do not need to specify the file type.

DESCRIPTION

See UIS$SET_FILL_PATTERN.

UIS$SET_GAIN_KB_AST

Specifies an AST routine to be executed when the specified virtual keyboard is attached to the physical keyboard.

FORMAT

UIS$SET_GAIN_KB_AST *kb_id [,astadr [,astprm]]*

RETURNS

UIS$SET_GAIN_KB_AST signals all errors; no condition values are returned.

ARGUMENTS

kb_id

See Section 18.3.8 for more information about the **kb_id** argument.

astadr

VMS Usage:	**ast_procedure**
type:	**procedure entry mask**
access:	**read only**
mechanism:	**by reference**

AST routine. The **astadr** argument is the entry mask address of a procedure called at AST level whenever a specified virtual keyboard is attached to the physical keyboard.

astprm

See Section 18.3.7 for more information on this argument.

DESCRIPTION

To disable UIS$SET_GAIN_KB_AST, omit the **astadr** and **astprm** arguments.

UIS$SET_INSERTION_POSITION

Positions the editing pointer in the display list.

FORMAT

UIS$SET_INSERTION_POSITION { *obj_id* | *seg_id* | *vd_id* } [*,flags*]

RETURNS

UIS$SET_INSERTION_POSITION signals all errors; no condition values are returned.

ARGUMENTS

obj_id

See Section 18.3.3 for a description of this argument.

seg_id

See Section 18.3.4 for a description of this argument. Note that when you specify **seg_id** as the first argument, the second argument is **not** specified.

vd_id

See Section 18.3.1 for a description of this argument.

flags

VMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Flags. The **flags** argument is the address of a longword mask whose bits define how entries are added to the display list.

The following table lists the flags and their functions.

Flags	Description
UIS$M_DL_INSERT_AT_BEGIN	Inserts object before first object in the specified structure.
UIS$M_DL_INSERT_AFTER_ OBJECT	Inserts object before specified object in the same segment as the specified object.
UIS$M_DL_INSERT_BEFORE_ OBJECT	Inserts object after specified object in the same segment as the specified object.

See the DESCRIPTION section for more information about how these flags are evaluated.

DESCRIPTION

UIS$SET_INSERTION_OBJECT examines different options in the **flags** argument depending on the type of object you specify in the first argument. The following table lists the effect of the flags on the different types of objects.

Flags Checked	Effect
Specifying the Virtual Display Identifier	
UIS$M_DL_INSERT_AT_BEGIN[1]	If this bit is set, the editing pointer is placed at the beginning of the root segment and all new objects are inserted there. If this bit is not set, the editing pointer is placed at the end of the root segment and all new objects are appended to the end of the root segment.
Specifying the Segment Identifier	
All three bits[2]	If any bit is set, UIS$SET_INSERTION_ POSITION sets the editing pointer at the place directed by that bit. If no bits are set, the editing pointer is placed at the end of the specified segment and any new objects are appended to the end of the specified segment.
Specifying the Object Identifier	
UIS$M_DL_INSERT_AFTER_ OBJECT[1] UIS$M_DL_INSERT_BEFORE_ OBJECT	If any bit is set, UIS$SET_INSERTION_ POSITION sets the editing pointer at the place directed by that bit. If no bits are set, the editing pointer is placed at the specified object and any new objects are inserted before the specified object.

[1]If UIS$M_DL_INSERT_BEFORE_OBJECT or UIS$M_DL_INSERT_AFTER_OBJECT are set, the routine signals an error.

[2]If two bits are set, the routine signals an error.

UIS$SET_INTENSITIES

Loads one or more intensity values in the virtual color map.

FORMAT **UIS$SET_INTENSITIES** *vd_id, index, count, i_vector*

RETURNS UIS$SET_INTENSITIES signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

index

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Starting color map index. The **index** argument is the address of a longword that identifies the starting color map index in the virtual color map.

If an index exceeds the maximum index for the virtual color map, an error is signaled.

count

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Number of indices. The **count** argument is the address of a longword that defines the number of indices in the virtual color map (including the starting index) whose entries are to be loaded with intensity values.

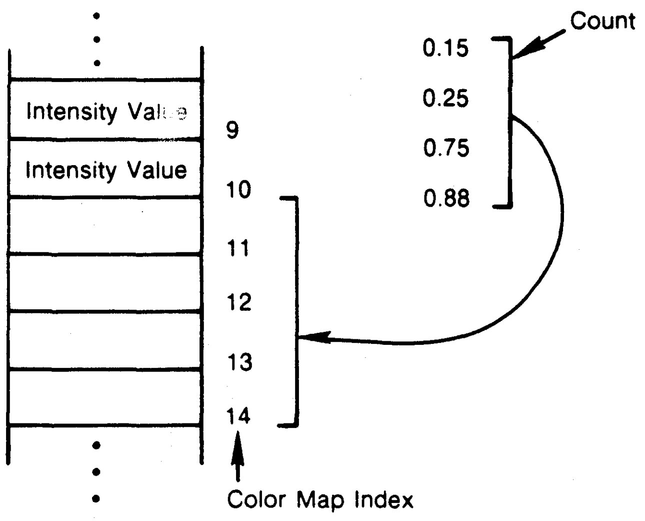
If **count** exceeds the maximum number of virtual color map entries, an error is signaled.

i_vector

VMS Usage:	**vector_longword_signed**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Intensity values. The **i_vector** argument is the address of an array of f_ floating point numbers that define the intensity values of the virtual color map entries.

illustration



ZK-5440-86

UIS$SET_INTENSITY

Loads a single entry in the virtual color map with an intensity value.

FORMAT

UIS$SET_INTENSITY *vd_id, index, I*

RETURNS

UIS$SET_INTENSITY signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

index

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

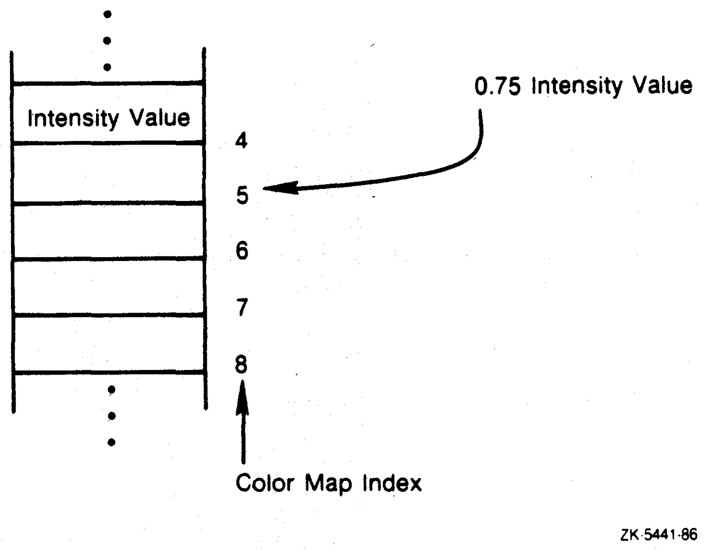
Color map index. The **index** argument is the address of a longword value that identifies an entry in the color map. If the index exceeds the maximum index for the associated color map, an error is signaled.

I

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Intensity value. The **I** argument is the address of an f_floating point number that defines the intensity. The intensity value is in the range of *0.0* to *1.0*, inclusive.

illustration



UIS$SET_KB_AST

Associates a key strike with the execution of a user-written AST routine.

FORMAT

UIS$SET_KB_AST *kb_id [,astadr [,astprm] ,keybuf]*

RETURNS

UIS$SET_KB_AST signals all errors; no condition values are returned.

ARGUMENTS

kb_id

See Section 18.3.8 for more information about the **kb_id** argument.

astadr

VMS Usage:	**ast_procedure**
type:	**procedure entry mask**
access:	**read only**
mechanism:	**by reference**

AST routine. The **astadr** argument is the entry mask address of a procedure that is called at AST level whenever you strike a key. To cancel a previous AST-enabling request of UIS$SET_KB_AST, specify *0* as the **astadr** argument.

astprm

See Section 18.3.7 for more information on this argument.

keybuf

VMS Usage:	**address**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Key buffer. The **keybuf** argument is the address of a longword buffer that receives the key information with the execution of each AST routine. The low two bytes are the key code. The key codes are based on the codes found in the module $SMGDEF in SYS$LIBRARY:STARLET.MLB. Bit <31> is set to *1* to indicate that the key is down. The AST routine is called only on the downstroke of the key. The buffer is not overwritten with subsequent keys until the AST routine completes.

The following table defines the bits in the high- and lower-order word.

Field	Symbol
1-16	UIS$W_KEY_CODE
28	UIS$V_KEY_SHIFT[1]
29	UIS$V_KEY_CTRL[1]
30	UIS$V_KEY_LOCK[1]
31	UIS$V_KEY_DOWN[1]

[1]This symbol is returned as SET if the corresponding key on the keyboard was down when the input event occurred.

DESCRIPTION

The terminal emulators use this routine to get all keyboard input. Other applications that perform asynchronous single character input can also use UIS$SET_KB_AST.

To disable UIS$SET_KB_AST, omit the **astadr** and **astprm** arguments.

UIS$SET_KB_ATTRIBUTES

Modifies the keyboard characteristics.

FORMAT

UIS$SET_KB_ATTRIBUTES *kb_id [,enable_items] [,disable_items] [,click_volume]*

RETURNS

UIS$SET_KB_ATTRIBUTES signals all errors; no condition values are returned.

ARGUMENTS

kb_id

See Section 18.3.8 for more information about the **kb_id** argument.

enable_items
disable_items

VMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Keyboard characteristics to be enabled. The **enable_items** argument is the address of a longword mask that identifies the keyboard characteristics to be enabled. The **disable_items** argument is the address of a longword mask that identifies the keyboard characteristics to be disabled.

click_volume

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Click volume level. The **click_volume** argument is the address of a longword value that modifies the keyboard click volume for keyboard input to this window. The value is in the range *1* to *8*, where the value *1* is the minimum volume level, and the value *8* is the maximum volume level. The default volume level is controlled by the workstation setup menu mechanism.

DESCRIPTION

All keyboard characteristics will be in effect only when the physical keyboard is attached to the specified virtual keyboard. Each virtual keyboard maintains its own keyboard characteristics and the human interface automatically switches the characteristics when the keyboard is associated with another virtual keyboard.

The enable and disable item lists are longword masks containing bits designating the characteristics to be enabled or disabled. The valid bits in the keyboard characteristics enable and disable masks are:

Symbol	Description[1]
UIS$M_KB_AUTORPT	Enable/disable keyboard autorepeat
UIS$M_KB_KEYCLICK	Enable/disable keyboard keyclick
UIS$M_KB_UDF6	Enable/disable up button transitions for F6 to F10 keys
UIS$M_KB_UDF11	Enable/disable up button transitions for F11 to F14 keys
UIS$M_KB_UDF17	Enable/disable up button transitions for F17 to F20 keys
UIS$M_KB_HELPDO	Enable/disable up button transitions for HELP and DO keys
UIS$M_KB_UDE1	Enable/disable up button transitions for E1 to E6 keys
UIS$M_KB_ARROW	Enable/disable up button transitions for arrow keys
UIS$M_KB_KEYPAD	Enable/disable up button transitions for numeric keypad keys

[1]By default down button transitions are enabled.

EXAMPLE

```
    .
    .
    .
enable_items=UIS$M_KB_HELPDO .OR. UIS$M_KB_UDE1 .OR. UIS$M_KB_ARROW
disable_items=UIS$M_KB_AUTORPT .OR. UIS$M_KB_KEYCLICK
CALL UIS$SET_KB_ATTRIBUTES(KB_ID, ENABLE_ITEMS,DISABLE_ITEMS)
    .
    .
    .
```

The preceding example describes how to enable and disable more than one keyboard characteristic at a time.

UIS$SET_KB_COMPOSE2

Loads a two-stroke compose sequence table for the specified virtual keyboard.

FORMAT

UIS$SET_KB_COMPOSE2 *kb_id [,table, tablelen]*

RETURNS

UIS$SET_KB_COMPOSE2 signals all errors; no condition values are returned.

ARGUMENTS

kb_id

See Section 18.3.8 for more information about the **kb_id** argument.

table

VMS Usage:	**vector_longword_unsigned**
type:	**longword array**
access:	**read only**
mechanism:	**by reference**

Compose table. The **table** argument is the address of an array that identifies the compose table. If no table is specified, the system default table is reestablished.

tablelen

VMS Usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**read only**
mechanism:	**by reference**

Length of the compose table in bytes. The **tablelen** argument is the address of word that defines the length of the compose table in bytes.

DESCRIPTION

You can use *compose sequences* to create characters that do not exist as standard keys on your keyboard.

Two-stroke sequences can be used on all keyboards except the North American keyboard. Two-stroke sequences do not use the COMPOSE key. Although faster to use than the three-stroke sequence, two-stroke sequences are limited to sequences starting with the following nonspacing diacritical marks: grave accent (`), acute accent (´), circumflex accent (ˆ), tilde mark (˜), dieresis mark (¨), and the ring mark. Instead of using the COMPOSE key, as in a three-stroke sequence, you use a nonspacing diacritical mark to initiate the two-stroke sequence. You then enter a standard character that, together with that diacritical mark, results in a valid compose sequence.

Please refer to the *VMS Workstation Video Device Driver Manual* for a description of this table and the macros to generate it. An application wishing to modify a table can use these macros to build a new table.

The Digital standard two-stroke compose table resides within the workstation driver. To change that, call the SYS$QIO system service to the QVSS device driver.

NOTE: **DIGITAL standard two-stroke compose sequences are not supported on the North American keyboard.**

UIS$SET_KB_COMPOSE3

Loads a three-stroke compose sequence for the specified virtual keyboard.

FORMAT

UIS$SET_KB_COMPOSE3 *kb_id [,table, tablelen]*

RETURNS

UIS$SET_KB_COMPOSE3 signals all errors; no condition values are returned.

ARGUMENTS

kb_id

See Section 18.3.8 for more information about the **kb_id** argument.

table

VMS Usage:	**vector_longword_unsigned**
type:	**longword array**
access:	**read only**
mechanism:	**by reference**

Compose table. The **table** argument is the address of an array that identifies the compose table.

tablelen

VMS Usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**read only**
mechanism:	**by reference**

Length of the compose table in bytes. The **tablelen** argument is the address of a word that defines the length of the compose table in bytes.

DESCRIPTION

You can use *compose sequences* to create characters that do not exist as standard keys on your keyboard. There are two types of compose sequences: two-stroke sequences and three-stroke sequences

You can perform three-stroke sequences on all keyboards by first pressing the COMPOSE key and then pressing two standard keys.

Refer to the *VMS Workstation Video Device Driver Manual* for a description of this table and the macros to generate it. An application wishing to modify a table can use these macros to build a new table.

The Digital standard two-stroke compose table resides within the workstation driver. To change that, call the SYS$QIO system service to the QVSS device driver.

UIS$SET_KB_KEYTABLE

Loads a keyboard equivalence table for the specified virtual keyboard.

FORMAT

UIS$SET_KB_KEYTABLE *kb_id [,table, tablelen]*

RETURNS

UIS$SET_KB_KEYTABLE signals all errors; no condition values are returned.

ARGUMENTS

kb_id

See Section 18.3.8 for more information about the **kb_id** argument.

table

VMS Usage:	**vector_longword_unsigned**
type:	**longword array**
access:	**read only**
mechanism:	**by reference**

Keyboard table. The **table** argument is the address of an array that contains the keyboard table. If no table is specified, the system default table is reestablished.

tablelen

VMS Usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**read only**
mechanism:	**by reference**

Length of the keyboard table. The **tablelen** argument is the address of a word that specifies the length of the keyboard table in bytes.

DESCRIPTION

UIS$SET_KB_KEYTABLE lets you change the ASCII character returned by a key on the keyboard.

Keyboard Table Description and Macros

Refer to the *VMS Workstation Video Device Driver Manual* for a description of the table and the macro to build it. An application wishing to modify a table can use these macros to build a new table.

Keyboard Table Modification Using the Programming Interface

The VMS workstation contains a copy of the North American table established as the default keyboard table. You can modify the default keyboard table at the driver (QVSS) level by calling the SYS$QIO system service.

Keyboard Table Modification Through the User Interface

If you want to create a keyboard table that any user can load using the Workstation Setup menus, see the command file DVORAK.COM in the directory SYS$EXAMPLES. It provides an example of how to create, compile, and install the DVORAK simplified keyboard. The user interface can be used to modify the default key table.

UIS$SET_LINE_STYLE

Sets the line style bit vector.

FORMAT **UIS$SET_LINE_STYLE** *vd_id, iatb, oatb, style*

RETURNS UIS$SET_LINE_STYLE signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

iatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.5 for a description of this argument.

oatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.6 for a description of this argument.

Controls the line style.

style

VMS Usage:	**mask_longword**
type:	**longword**
access:	**read only**
mechanism:	**by reference**

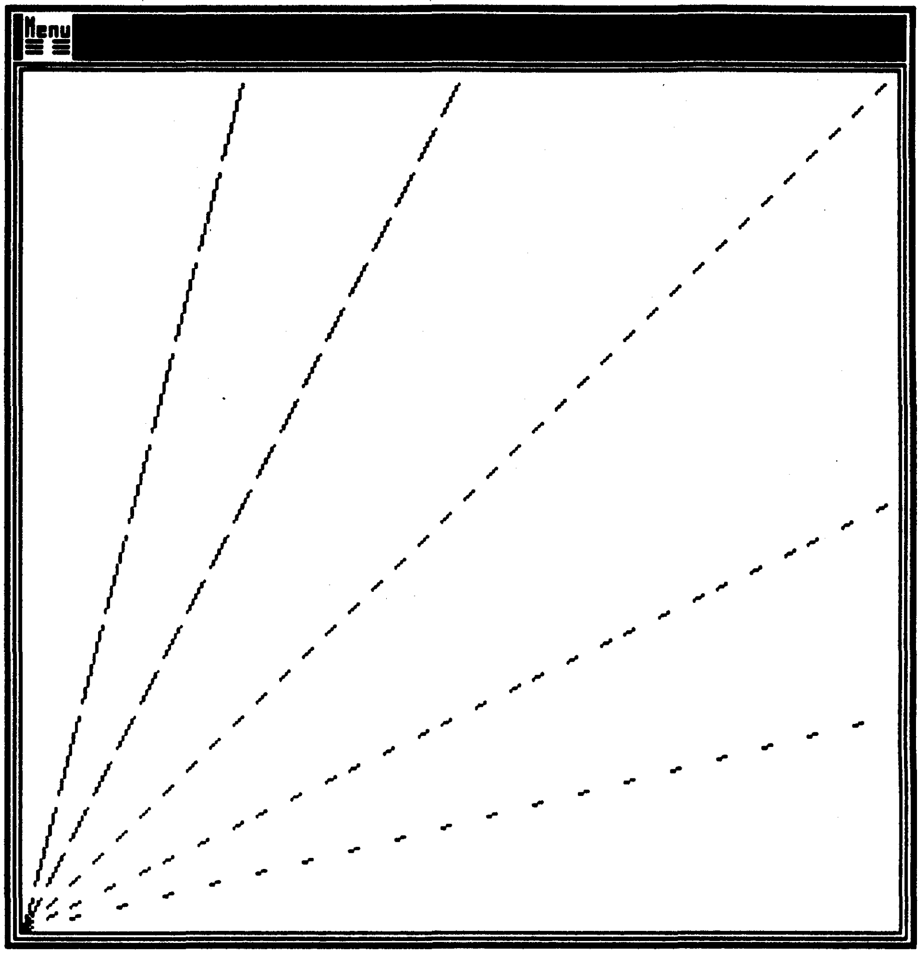
Line style bit vector. The **style** argument is the address of a longword bit vector that specifies whether to use foreground or background when drawing each pixel. It is repeated as many times as necessary to draw all the pixels in the line.

EXAMPLE

```
.
.
.
     CALL UIS$SET_LINE_STYLE(VD_ID,0,1,'FFFFFFF0'x)
     CALL UIS$PLOT(VD_ID,1,0.0,0.0,5.0,20.0)

     CALL UIS$SET_LINE_STYLE(VD_ID,0,2,'FFF0FFF0'x)
     CALL UIS$PLOT(VD_ID,2,0.0,0.0,10.0,20.0)
.
.
.
```

The preceding example produces the first two dashed lines shown in the next section.

screen output



ZK-5285-86

UIS$SET_LINE_WIDTH

Sets the width of lines drawn on the screen.

FORMAT

UIS$SET_LINE_WIDTH *vd_id, iatb, oatb, width [,mode]*

RETURNS

UIS$SET_LINE_WIDTH signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

iatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.5 for a description of this argument.

oatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.6 for a description of this argument.

Controls line width.

width

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Width of the line. The **width** argument is the address of an f_floating point number that defines the line width. See the Description section for more information about specifying the line width with UIS$C_WIDTH_WORLD. The default value is 1.

mode

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Mode. The **mode** argument is the address of a longword that indicates whether the line width should be interpreted as an absolute number of pixels or as an x world coordinate width. Specify the mode using one of the following constants:

- UIS$C_WIDTH_PIXELS
- UIS$C_WIDTH_WORLD

If **mode** is not specified, line width is interpreted as an absolute number of pixels (UIS$C_WIDTH_PIXELS).

See the Description section for more information about the constant UIS$C_WIDTH_WORLD.

DESCRIPTION

The line width is specified as a floating point number that is multiplied by the normal line width to produce line width actually drawn.

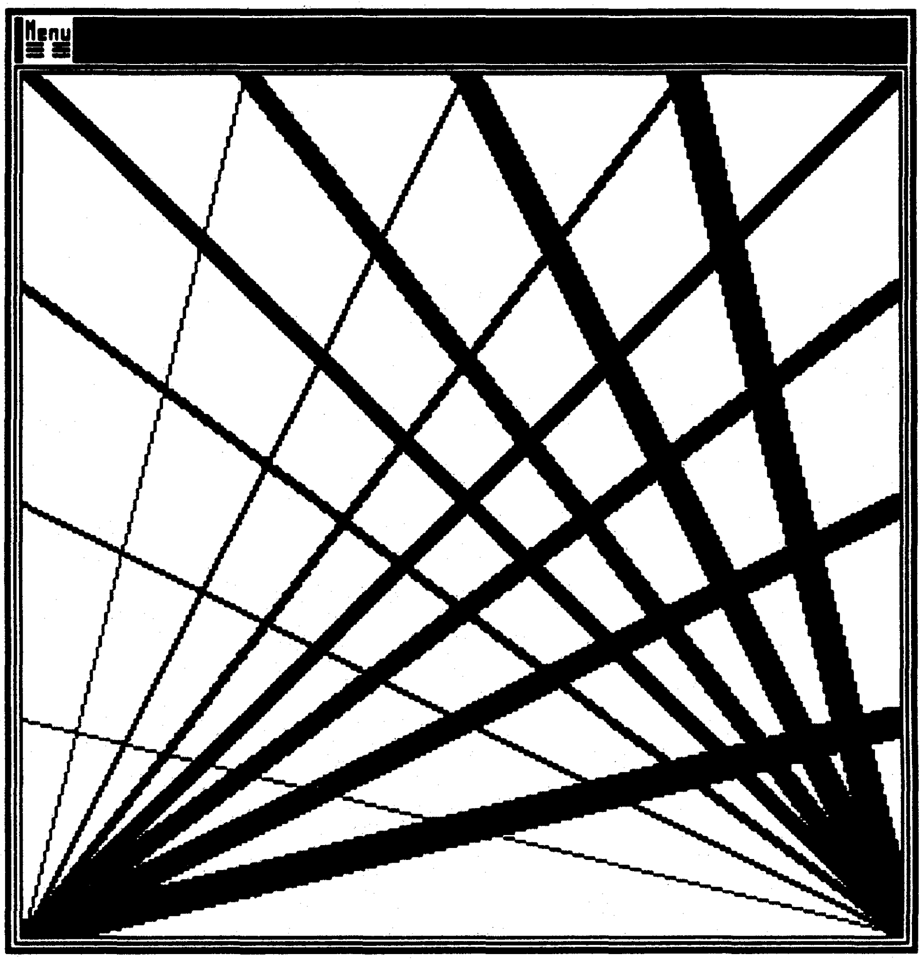
If you specify *0.0* in the **width** argument when the **mode** argument is UIS$C_WIDTH_WORLD, the minimum line width is generated.

EXAMPLE

```
   .
   .
   .
      CALL UIS$SET_LINE_WIDTH(VD_ID,0,1,2.0,WDPL$C_WIDTH_WORLD)
      CALL UIS$PLOT(VD_ID,1,0.0,0.0,10.0,20.0)

      CALL UIS$SET_LINE_WIDTH(VD_ID,0,2,4.0,WDPL$C_WIDTH_WORLD)
      CALL UIS$PLOT(VD_ID,2,0.0,0.0,15.0,20.0)
   .
   .
   .
```

The preceding example describes how to specify line width as x world coordinate width.

screen output



ZK-5286-86

UIS$SET_LOSE_KB_AST

Enables an AST routine that is executed when the specified virtual keyboard is detached from the physical keyboard.

FORMAT

UIS$SET_LOSE_KB_AST *kb_id [,astadr [,astprm]]*

RETURNS

UIS$SET_LOSE_KB_AST signals all errors; no condition values are returned.

ARGUMENTS

kb_id

See Section 18.3.8 for more information about the **kb_id** argument.

astadr

VMS Usage:	**ast_procedure**
type:	**procedure entry mask**
access:	**read only**
mechanism:	**by reference**

AST routine. The **astadr** argument is the entry mask address to a procedure that is called at AST level whenever the virtual keyboard is disconnected from the physical keyboard.

astprm

See Section 18.3.7 for more information on this argument.

DESCRIPTION

To cancel the AST-enabling request of UIS$SET_LOSE_KB_AST, specify 0 in the **astadr** argument or omit the **astadr** and **astprm** arguments.

UIS$SET_MOVE_INFO_AST

Enables an AST routine execution whenever the specified display viewport has been moved.

FORMAT

UIS$SET_MOVE_INFO_AST *wd_id, [,astadr [,astprm]]*

RETURNS

UIS$SET_MOVE_INFO_AST signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for a description of this argument.

astadr

VMS Usage:	**ast_procedure**
type:	**procedure entry mask**
access:	**read only**
mechanism:	**by reference**

AST routine. The **astadr** argument is the entry mask address to a procedure that is called at AST level whenever the specified display viewport is moved.

astprm

See Section 18.3.7 for more information on this argument.

DESCRIPTION

A MOVE notification AST can be used when an image needs to keep several display viewports in a particular arrangement. If one is moved, the AST routine can recreate the other display viewports in the correct positions around the moved viewport.

To cancel the AST-enabling request of UIS$SET_MOVE_INFO_AST, perform any of the following actions:

- Specify the **wd_id** argument only.
- Specify *0* in the optional **astadr** argument.
- Omit the **astadr** and **astprm** arguments.

UIS$SET_POINTER_AST

Allows an application to find out when the pointer is moved within, into, and out of a specified rectangle in the display window.

FORMAT

UIS$SET_POINTER_AST *vd_id, wd_id [,astadr [,astprm]] [,x_1, y_1, x_2, y_2] [,exitastadr [,exitastprm]]*

RETURNS

UIS$SET_POINTER_AST signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

wd_id
See Section 18.3.2 for a description of this argument.

astadr

VMS Usage:	**ast_procedure**
type:	**procedure entry mask**
access:	**read only**
mechanism:	**by reference**

AST routine. The **astadr** argument is the entry mask address to a procedure that is called at AST level whenever you move the pointer within a rectangle in the virtual display.

To cancel the AST-enabling request of UIS$SET_POINTER_AST for this argument only, specify *0* in the **astadr** argument and the coordinates of the rectangle.

astprm
See Section 18.3.7 for more information on this argument.

x_1, y_1
x_2, y_2

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

World coordinates of the rectangle. The x_1 and y_1 arguments are the addresses of f_floating point numbers that define the lower-left corner of the rectangle of the display window. The x_2 and y_2 arguments are the addresses of f_floating point numbers that define the upper-right corner of the rectangle of the display window.

If no rectangle is specified, the entire display window is assumed.

To cancel an AST-enabling request, specify *0* in either the **astadr** or the **exitastadr** arguments or both and the coordinates of the rectangle.

exitastadr

VMS Usage:	**ast_procedure**
type:	**procedure entry mask**
access:	**read only**
mechanism:	**by reference**

Exit AST routine. The **exitastadr** argument is the address of the entry mask to a procedure that is called at AST level whenever the pointer leaves the rectangle.

To cancel the AST-enabling request of UIS$SET_POINTER_AST for the EXIT AST routine only, specify *0* in the **exitastadr** argument and the coordinates of the rectangle.

exitastprm

VMS Usage:	**user_arg**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Exit AST parameter. The **exitastprm** argument is the address of a single argument or data structure, such as an array or record, to be passed to the AST routine. Calls to UIS$SET_POINTER_AST in FORTRAN application programs should be coded as follows: %REF(%LOC(exitastprm)).

DESCRIPTION

The Set Pointer AST routine allows an application to keep track of the pointer. This routine can be called any number of times for different rectangles.

Note that an application can specify one AST routine or the other.

The application can use UIS$SET_POINTER_AST to highlight the display or some other application-specific function as you move the pointer over specific areas of the display window. Use this to define a number of regions within a menu and execute an AST routine when the pointer enters or leaves any of these regions.

If both AST routines are enabled and the value *0* is specified in the **astadr** argument, the first AST routine is canceled.

To disable AST-enabling behavior for pointers entering a region, omit the **astadr** and **astprm** arguments.

To disable AST-enabling behavior for pointers leaving a region, omit the **exitastadr** and **exitastprm** arguments.

Pointer Region Priorities

UIS pointer regions appear on the VAXstation screen in the order they are created. Therefore, if you create two overlapping viewports, then use UIS$SET_POINTER_PATTERN, UIS$SET_BUTTON_AST, or UIS$SET_POINTER_AST to define different pointer patterns for each viewport, the *correctness* of the result will depend on the order in which you created the viewports and defined the cursor regions. For example, the viewport 1 cursor pattern will have a higher priority than the viewport 2 cursor pattern in the overlapping region if you create the viewports and define the cursor patterns as follows:

1. Create viewport 1
2. Create overlapping viewport 2
3. Define viewport 2 cursor pattern
4. Define viewport 1 cursor pattern

In preceding example, the viewport 1 cursor pattern takes priority over the viewport 2 cursor pattern in the overlapping region. This is an undesired result, and you can correct it by creating the viewports and defining the cursor patterns in the same order, as follows:

1. Create viewport 1
2. Define viewport 1 cursor pattern
3. Create overlapping viewport 2
4. Define viewport 2 cursor pattern

To avoid this problem, have UIS or your application pop the viewport before you define the cursor region for it.

UIS$SET_POINTER_PATTERN

Allows an application to specify a special pointer cursor pattern for a specified rectangle in the virtual display.

FORMAT

UIS$SET_POINTER_PATTERN *vd_id, wd_id [,pattern_array, pattern_count, activex, activey] [, x_1, y_1, x_2, y_2][,flags]*

RETURNS

UIS$SET_POINTER_PATTERN signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

wd_id
See Section 18.3.2 for a description of this argument.

pattern_array

VMS Usage:	**vector_word_unsigned**
type:	**word_unsigned**
access:	**read only**
mechanism:	**by reference**

16- x 16-bit cursor pattern. The **pattern_array** argument is the address of one or more 16-bit arrays of 16 words that represents a bitmap image of the cursor pattern.

You can define two patterns that are executable on color and intensity systems using two arrays—a color plane and a mask plane. However, monochrome systems use a single array to specify the cursor pattern.

If two arrays are specified in an application running on a single-plane system, the first array is used.

NOTE: **The bitmap image of the new pointer pattern is mapped in reverse order to the display screen.**

pattern_count

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Number of 16- x 16-bit cursor patterns defined. The **pattern_count** argument is the address of a longword that contains the number of cursor pattern arrays defined in the **pattern_array** argument.

activex
activey

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

The **activex** and **activey** arguments are used to specify the actual bit in the cursor pattern that should be used to calculate the current pointer position. The arguments are expressed as bit offsets from the lower-left corner of the cursor pattern.

x_1, y_1
x_2, y_2

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

World coordinates of the rectangle in the virtual display. The x_1 and y_1 arguments are the addresses of f_floating point numbers that define the lower-left corner of the rectangle in the display window. The x_2 and y_2 arguments are the addresses of f_floating point numbers that define the upper-right corner of the rectangle in the display window.

flags

VMS Usage:	**longword_mask**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Flags. The **flags** argument is the address of a longword mask whose bits determine whether or not the cursor is confined to the display window rectangle.

When specified, UIS$M_BIND_POINTER sets the appropriate bit in the mask.

DESCRIPTION

UIS$SET_POINTER_PATTERN allows an application to specify a special pointer pattern to be used when the pointer is within the display window region specified by the optional rectangle. If no rectangle is given, then the entire display window is assumed. This function can be called any number of times for different rectangles.

To disable UIS$SET_POINTER_PATTERN, omit the **pattern_array, pattern_count, activex, activey,** and **flags** arguments.

UIS$SET_POINTER_POSITION

Specifies a new current pointer position in world coordinates. It is only effective if the new pointer position is within the specified display window and visible.

FORMAT

status = **UIS$SET_POINTER_POSITION** *vd_id, wd_id, x, y*

RETURNS

VMS Usage:	**Boolean**
type:	**longword**
access:	**write only**
mechanism:	**by value**

Boolean value returned in a status variable or R0 (VAX MACRO). A status of 1 is returned, if the operation is successful, otherwise a 0 is returned.

UIS$SET_POINTER_POSITION signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

wd_id

See Section 18.3.2 for a description of this argument.

x

y

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

World coordinates of the new pointer position. The x and y arguments are the addresses of f_floating point numbers that define the new pointer position.

UIS$SET_POSITION

Sets the current position for text output. The current position is the point of alignment on the baseline of the next character to be output.

FORMAT

UIS$SET_POSITION *vd_id, x,y*

RETURNS

UIS$SET_POSITION signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

x
y

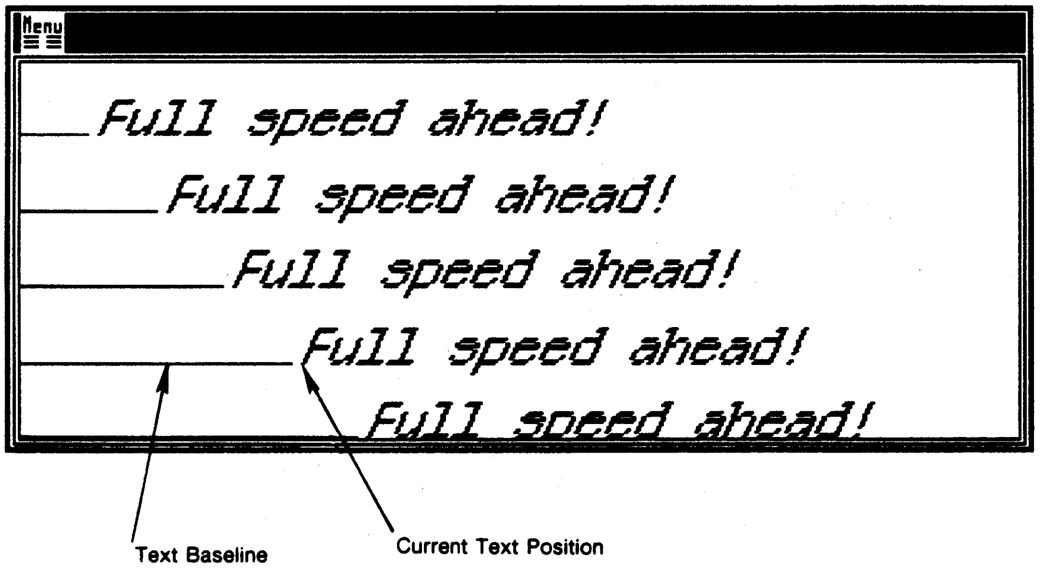
VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

X and y world coordinate position. The x and y arguments are the addresses of f_floating point numbers that define the current position for text output.

EXAMPLE

```
.
.
.
REAL*4 Y
DATA Y/4.0/
.
.
.
DO I=1,5
CALL UIS$SET_POSITION(VD_ID,FLOAT(I),Y)
CALL UIS$PLOT(VD_ID,1,0.0,Y,FLOAT(I),Y)
Y=Y-1.0
CALL UIS$SET_FONT(VD_ID,1,1,'MY_FONT_11')
CALL UIS$TEXT(VD_ID,1,'Full speed ahead!')
ENDDO
.
.
.
```

screen output



ZK-5386-86

UIS$SET_RESIZE_AST

Specifies a user-requested AST routine to be executed when a display window has been resized using the user interface.

FORMAT

UIS$SET_RESIZE_AST *vd_id, wd_id [,astadr [,astprm]] [,new_abs_x, new_abs_y] [,new_width, new_height] [,new_wc_x*$_1$*, new_wc_y*$_1$*, new_wc_x*$_2$*, new_wc_y*$_2$*]*

RETURNS

UIS$SET_RESIZE_AST signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

wd_id
See Section 18.3.2 for a description of this argument.

astadr

VMS Usage:	**ast_procedure**
type:	**procedure entry mask**
access:	**read only**
mechanism:	**by reference**

AST routine. The **astadr** argument is the entry mask address of a procedure that is called at AST level whenever you select the Change the Size item in the Window Options Menu and a display window has been resized.

See the Description section for information about disabling UIS$SET_RESIZE_AST.

astprm
See Section 18.3.7 for more information on this argument.

new_abs_x
new_abs_y

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Absolute device coordinate pair. The **new_abs_x** and **new_abs_y** arguments are the addresses of f_floating point longwords that receive the exact location of the newly resized display window in centimeters.

new_width
new_height

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

Width and height of the resized window. The **new_width** and **new_height** arguments are the addresses of f_floating point longwords that receive the dimensions of the newly resized display window in centimeters.

new_wc_x_1, new_wc_y_1
new_wc_x_2, new_wc_y_2

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**write only**
mechanism:	**by reference**

World coordinates of the resized window. The **new_wc_x_1** and **new_wc_y_1** arguments are the addresses of f_floating point longwords that receive the world coordinates of the lower-left corner of the newly resized display window. The **new_wc_x_2** and **new_wc_y_2** arguments are the addresses of f_floating point longwords that receive the world coordinates of the upper-right corner of the newly resized display window.

DESCRIPTION

Typically, a call to UIS$SET_RESIZE_AST in an application program indicates that the default resizing behavior is to be overridden.

By default, if a resize AST has not been enabled in an application program, UIS calls UIS$RESIZE_WINDOW. If this behavior is not sufficient, the application program can call UIS$SET_RESIZE_AST with its own resize routine.

To reenable the default behavior, specify UIS$C_DEFAULT_RESIZE as the **astadr** argument in a subsequent call to UIS$SET_RESIZE_AST.

You can completely disable the ability to resize a window in the following ways:

- Specify the required **wd_id** argument and a value of 0 in the **astadr** argument
- Specify only the required **wd_id** argument
- Omit the **astadr** and **astprm** arguments.

When window resizing is disabled, the option, ''Change the size'' displayed in the Window Options Menu changes from boldface to halftone.

The parameters for the resized window's new location, dimensions, and world coordinate range will not be overwritten with subsequent values until the AST has completed.

EXAMPLE

```
    .
    .
    .
VD_ID=UIS$CREATE_DISPLAY(1.0,1.0,40.0,40.0,15.0,15.0)    1
WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION','RESIZE',
2        5.0,5.0,25.0,25.0) 2
    .
    .
    .
CALL UIS$SET_RESIZE_AST(VD_ID,WD_ID,RESIZE_ME,0,NEW_ABS_X,NEW_ABS_Y,
2        NEW_WIDTH,NEW_HEIGHT,NEW_WC_X1,NEW_WC_Y1,
2        NEW_WC_X2,NEW_WC_Y2)  3

CALL SYS$HIBER()
    .
    .
    .
END   !end of main program
    .
    .
    .
SUBROUTINE RESIZE_ME   4
    .
    .
    .
CALL UIS$RESIZE_WINDOW(VD_ID,WD_ID,NEW_ABS_X,NEW_ABS_Y,,,
2        1.0,1.0,40.0,40.0)  5
    .
    .
    .
RETURN
END
    .
    .
    .
```

In the preceding example, the call to UIS$CREATE_DISPLAY 1 establishes the initial viewport size as a square.

The coordinate space of the initial display window is defined to be a subset of the virtual display 2. When the original window is displayed it will show only a portion of the virtual display.

The call to UIS$SET_RESIZE_AST 3 indicates that the program will override the default window resizing operation by enabling a user-written AST routine RESIZE_ME 4.

The parameter list of UIS$RESIZE_WINDOW 5 indicates how the resize operation is redefined. The absolute position and size of all viewports will default as usual to the final position and dimensions of the stretchy box.

However, the world coordinate range of the newly resized window is defined explicitly as the coordinate range of the virtual display. All newly resized windows will show the entire virtual display. If you tried to resize a previously resized window, you would still see the contents of the entire virtual display.

Distortion of objects displayed in the viewport will occur whenever the aspect ratio of the newly resized viewport does not equal the aspect ratio of the newly resized display window.

UIS$SET_SHRINK_TO_ICON_AST

Specifies a user-requested AST routine to be executed whenever a display viewport is shrunk using the human interface.

FORMAT

UIS$SET_SHRINK_TO_ICON_AST *wd_id [,astadr [,astprm]]*

RETURNS

UIS$SET_SHRINK_TO_ICON_AST signals all errors; no condition values are returned.

ARGUMENTS

wd_id
See Section 18.3.2 for a description of this argument.

astadr

VMS Usage:	**ast_procedure**
type:	**procedure entry mask**
access:	**read only**
mechanism:	**by reference**

AST routine. The **astadr** argument is the entry mask address of a procedure called at AST level whenever you select the Shrink to Icon item in the Window Options Menu.

astprm
See Section 18.3.7 for more information on this argument.

DESCRIPTION

The user interface for replacing a display viewport with an icon can be disabled by calling UIS$SET_SHRINK_TO_ICON_AST with the **wd_id** only.

To reenable the default behavior of UIS$SET_SHRINK_TO_ICON_AST, specify the constant UIS$C_DEFAULT_SHRINK_TO_ICON in the **astadr** argument.

UIS$SET_TB_AST

Specifies a user-requested AST routine to be executed whenever the digitizer lies within a specified rectangle on the tablet.

FORMAT

UIS$SET_TB_AST *tb_id, [,data_astadr, [data_astprm]], [,x_pos ,y_pos] [data_x_1,data_y_1,data_x_2,data_y_2] [,button_astadr [,button_astprm],button_keybuf]*

RETURNS

UIS$SET_TB_AST signals all errors; no condition values are returned.

ARGUMENTS

tb_id

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Tablet identifier. The **tb_id** argument is the address of a longword that uniquely identifies the tablet. See UIS$CREATE_TB for more information about the **tb_id** argument.

data_astadr

VMS Usage:	**ast_procedure**
type:	**procedure entry mask**
access:	**read only**
mechanism:	**by reference**

AST routine. The **data_astadr** argument is the address of an entry mask of a procedure that is called at AST level for each data point whenever the digitizer is moved within the specified active data region defined on the tablet.

See the Description section for information about disabling the digitizing region.

data_astprm

VMS Usage:	**user_arg**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**reference**

AST parameter. The **data_astprm** is the address of a single argument or data structure, such as an array or record, to be passed to the AST routine. Calls to UIS$SET_TB_AST in VAX FORTRAN application programs should be coded as follows: %REF(%LOC(astprm)).

x_pos
y_pos

VMS Usage: **floating_point**
type: **f_floating**
access: **write only**
mechanism: **by reference**

Absolute device coordinate pair. The **x_pos, y_pos** arguments are the addresses of f_floating longwords that receive the current x and y tablet positions in centimeters relative to the lower-left corner of the tablet, when a data AST occurs.

data_x$_1$*, data_y*$_1$
data_x$_2$*, data_y*$_2$

VMS Usage: **floating_point**
type: **f_floating**
access: **read only**
mechanism: **by reference**

Absolute device coordinate pair. The **data_x**$_1$**,data_y**$_1$ arguments are the addresses of f_floating point numbers that define the lower-left corner of the data or digitizer region specified on the tablet. The data rectangle defines an area on the tablet in which data should be collected.

button_astadr

VMS Usage: **ast_procedure**
type: **procedure entry mask**
access: **read only**
mechanism: **by reference**

AST routine. The **button_astadr** argument is the address of an entry mask of a procedure that is called at AST level whenever a button is depressed or released within the specified active data region defined on the tablet.

See the Description section for information about disabling the digitizing region.

button_astprm

VMS Usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **reference**

AST parameter. The **button_astprm** is the address of a single argument or data structure, such as an array or record, to be passed to the AST routine. Calls to UIS$SET_TB_AST in VAX FORTRAN application programs should be coded as follows: %REF(%LOC(astprm)).

button_keybuf

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Button information. The **button_keybuf** argument is the address of a longword that receives button information.

DESCRIPTION

The data rectangle specifies the active data region on the tablet. Only points within this rectangle are returned to the application. The data rectangle is specified using a centimeter coordinate system that is based at the lower-left corner of the tablet.

If no data rectangle is specified, the entire tablet is assumed.

Button AST Routines

To disable button AST routines, specify *0* in the **button_ast_rtn** argument.

UIS$SET_TEXT_FORMATTING

Sets the text formatting justification mode.

FORMAT

UIS$SET_TEXT_FORMATTING *vd_id, iatb, oatb, mode*

RETURNS

UIS$SET_TEXT_FORMATTING signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

iatb

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.5 for a description of this argument.

oatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.6 for a description of this argument.

mode

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Text formatting mode. The **mode** argument is the address of a longword mask that sets the text formatting mode. The following table lists valid text formatting modes.

Formatting Mode	Function
UIS$C_TEXT_FORMAT_LEFT	Left justified, ragged right
UIS$C_TEXT_FORMAT_RIGHT	Right justified, left ragged
UIS$C_TEXT_FORMAT_CENTER	Centered line between left and right margin
UIS$C_TEXT_FORMAT_JUSTIFY	Justified lines, space filled to right margin
UIS$C_TEXT_FORMAT_NOJUSTIFY	No text justification (default)

All other values are reserved to DIGITAL for future use.

DESCRIPTION

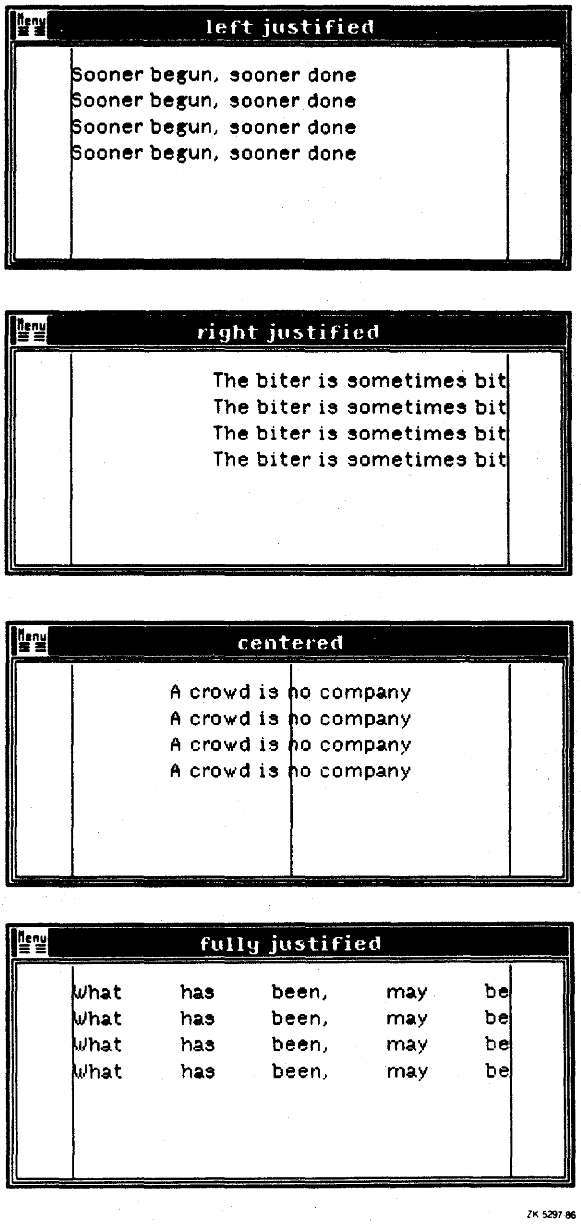
Text justification occurs at the end of every UIS$TEXT or UIS$MEASURE_TEXT call. Text justification also occurs when a UIS$C_TEXT_NEW_LINE item is encountered in a UIS$TEXT or UIS$MEASURE_TEXT control list. The formatting mode and margins that are used are based on either the attribute block specified in the routine call or the last attribute block specified before the UIS$C_TEXT_NEW_LINE item code is encountered.

NOTE: **Lines of text that do not fit completely within the margins will extend beyond the margin.**

EXAMPLE

```
    .
    .
    .
CALL UIS$SET_TEXT_MARGINS(VD_ID,0,1,3.0,27.0,24.0)
CALL UIS$PLOT(VD_ID,0,3.0,30.0,3.0,0.0)
CALL UIS$PLOT(VD_ID,0,27.0,30.0,27.0,0.0)

CALL UIS$SET_TEXT_FORMATTING(VD_ID,1,1,UIS$C_TEXT_FORMAT_JUSTIFY)
CALL UIS$SET_ALIGNED_POSITION(VD_ID,1,3.0,28.0)

CALL UIS$SET_FONT(VD_ID,1,2,'MY_FONT_8')

DO I= 1,4
CALL UIS$TEXT(VD_ID,2,'What has been, may be')
CALL UIS$NEW_TEXT_LINE(VD_ID,2)
ENDDO
    .
    .
    .
```

screen output



```
Sooner begun, sooner done
Sooner begun, sooner done
Sooner begun, sooner done
Sooner begun, sooner done
```

```
The biter is sometimes bit
The biter is sometimes bit
The biter is sometimes bit
The biter is sometimes bit
```

```
A crowd is no company
A crowd is no company
A crowd is no company
A crowd is no company
```

```
What     has     been,     may     be
What     has     been,     may     be
What     has     been,     may     be
What     has     been,     may     be
```

UIS$SET_TEXT_MARGINS

Sets the text margins for a line of text.

FORMAT **UIS$SET_TEXT_MARGINS** *vd_id ,iatb ,oatb ,x ,y ,margin_length*

RETURNS UIS$SET_TEXT_MARGINS signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

iatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.5 for a description of this argument.

oatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.6 for a description of this argument.

x
y

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Starting margin position. The **x,y** arguments are the addresses of f_floating numbers that define a point on the margin. The margin is the minor text path when slope equals zero.

margin_length

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Ending margin position. The **margin_length** is the address of an f_floating number that defines the distance in world coordinates from the starting margin to the end margin.

DESCRIPTION

Lines of text do not automatically wrap to the next line.

UIS$SET_TEXT_PATH

Sets the direction of text drawing and the direction of new text lines.

FORMAT

UIS$SET_TEXT_PATH *vd_id, iatb, oatb, major [,minor]*

RETURNS

UIS$SET_TEXT_PATH signal all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

iatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.5 for a description of this argument.

oatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.6 for a description of this argument.

major

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Major text path. The **major** argument is the address of a symbol that identifies the major text path type. The major path of text drawing is the direction of text drawing along a line. See the Description section for more information.

minor

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Minor text path. The **minor** argument is the address of a symbol that identifies the minor text path type. The minor path of text drawing refers to the direction of new text line creation. See the Description section for more information.

DESCRIPTION

The following table contains symbols for valid character drawing directions.

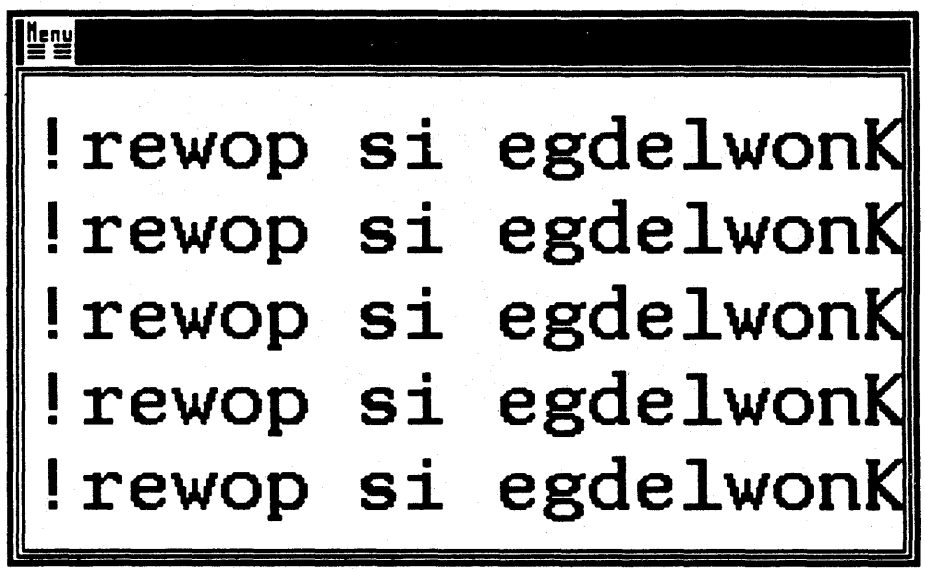
Path	Direction
UIS$C_TEXT_PATH_RIGHT	Left to right (default major text path)
UIS$C_TEXT_PATH_LEFT	Right to left
UIS$C_TEXT_PATH_UP	Bottom to top
UIS$C_TEXT_PATH_DOWN	Top to bottom (default minor text path)

EXAMPLE

```
.
.
.
      CALL UIS$SET_TEXT_PATH(VD_ID,0,1,UIS$C_TEXT_PATH_LEFT,
      2      UIS$C_TEXT_PATH_DOWN)

      CALL UIS$SET_FONT(VD_ID,1,1,'MY_FONT_5')

      CALL UIS$SET_ALIGNED_POSITION(VD_ID,1,38.0,38.0)

      CALL UIS$TEXT(VD_ID,1,'Knowledge is power!')
      CALL UIS$NEW_TEXT_LINE(VD_ID,1)
.
.
.
```

The preceding example illustrates how to alter the default major text drawing path to produce the output shown in the next section.

screen output



```
!rewop si egdelwonK
!rewop si egdelwonK
!rewop si egdelwonK
!rewop si egdelwonK
!rewop si egdelwonK
```

ZK-5287-86

UIS$SET_TEXT_SLOPE

Sets the angle of the actual path of text drawing relative to the major path.

FORMAT

UIS$SET_TEXT_SLOPE *vd_id ,iatb ,oatb ,angle*

RETURNS

UIS$SET_TEXT_SLOPE signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

iatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.5 for a description of this argument.

oatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.6 for a description of this argument.

angle

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

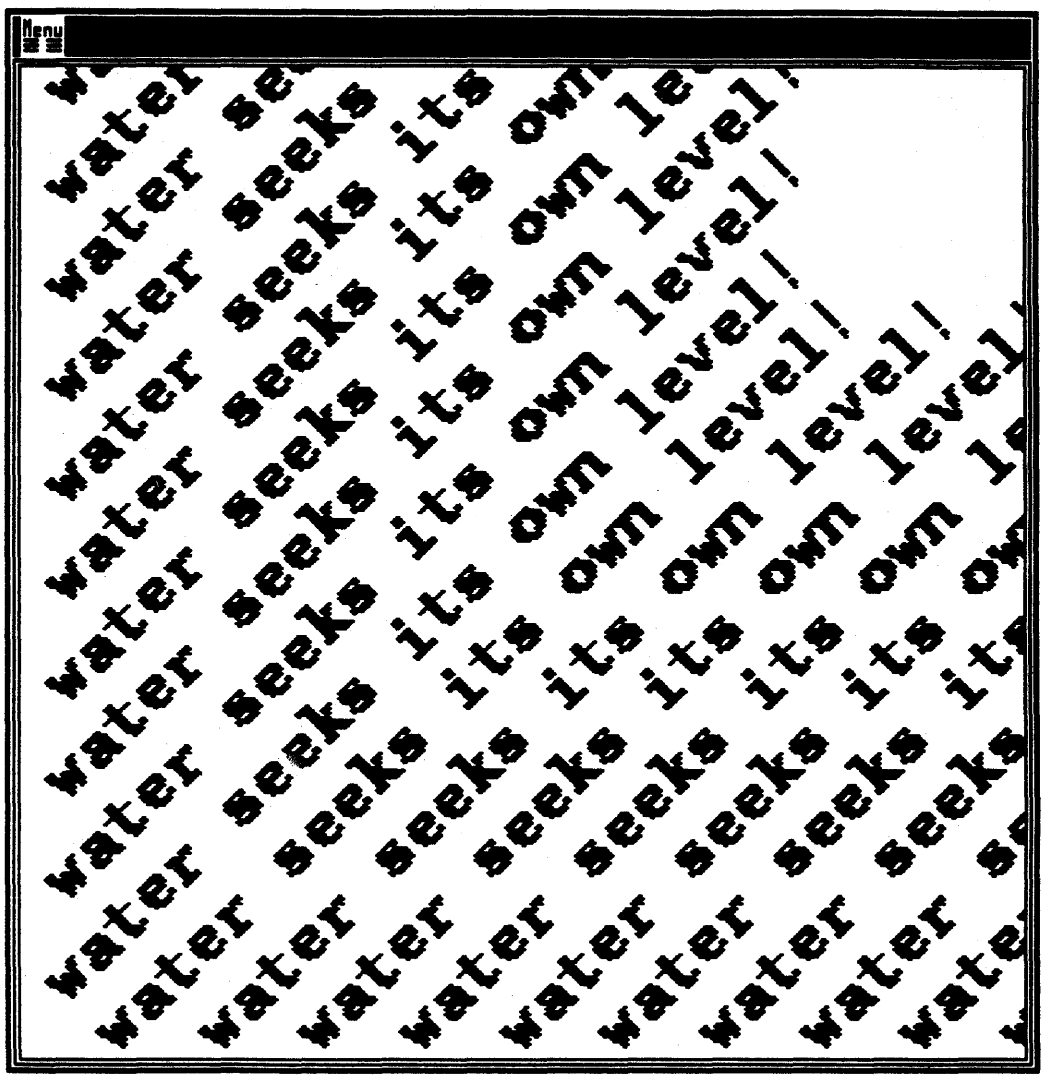
Angle of text slope. The **angle** argument is the address of an f_floating point number that defines the angle of the actual path of text drawing relative to the major path measured counterclockwise in degrees. The default angle of text slope is 0 degrees.

EXAMPLE

```
      .
      .
      .
          CALL UIS$SET_FONT(VD_ID,0,1,'MY_FONT_13')
          CALL UIS$SET_TEXT_SLOPE(VD_ID,1,2,45.0)

          DO I=1,10
          CALL UIS$SET_ALIGNED_POSITION(VD_ID,2,0.0,Y)
          CALL UIS$TEXT(VD_ID,2,'water seeks its own level!')
          Y=Y-2.0
          ENDDO
```

```
        PAUSE

        DO I=1,10
        CALL UIS$SET_ALIGNED_POSITION(VD_ID,2,X,1.0)
        CALL UIS$TEXT(VD_ID,2,'water seeks its own level!')
        X=X+2.0
        ENDDO
  .
  .
  .
```

screen output



ZK-5288-86

UIS$SET_WRITING_INDEX

Sets the writing color index for text and graphics output.

FORMAT

UIS$SET_WRITING_INDEX *vd_id, iatb, oatb, index*

RETURNS

UIS$SET_WRITING_INDEX signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

iatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.5 for a description of this argument.

oatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.6 for a description of this argument.

index

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Color map index. The **index** argument is the address of a longword integer that specifies a color map index. If the index exceeds the maximum index for the associated color map, an error is signaled.

UIS$SET_WRITING_MODE

Sets the text and graphics mode.

FORMAT

UIS$SET_WRITING_MODE *vd_id, iatb, oatb, mode*

RETURNS

UIS$SET_WRITING_MODE signals all errors; no condition values are returned.

ARGUMENTS

vd_id

See Section 18.3.1 for a description of this argument.

iatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.5 for a description of this argument.

oatb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

See Section 18.3.6 for a description of this argument.

Controls the writing mode.

mode

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Writing mode. The **mode** argument is the address of a longword that specifies the writing mode (UIS$C_MODE_xxxx). The default writing mode is overlay.

DESCRIPTION

Section 9.4 lists and describes all UIS writing modes.

UIS$SHRINK_TO_ICON

Replaces a display viewport with its associated icon.

FORMAT

UIS$SHRINK_TO_ICON *wd_id [,icon_wd_id] [,icon_flags] [,icon_name] [,attributes]*

RETURNS

UIS$SHRINK_TO_ICON signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for a description of this argument.

icon_wd_id

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Icon window identifier. The **icon_wd_id** argument is the address of a longword that uniquely identifies an icon.

icon_flags

VMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Icon flags. The **icon_flags** is the address of a longword mask of flags that can be used to specify whether default icon behavior should be extended to an application-supplied icon. By default, no modifications are made to the application-supplied icon. The following table lists valid icon flags.

Flag	Function
UIS$M_ICON_DEF_KB	UIS manages keyboard ownership. If the display window is enabled for keyboard ownership, UIS$DISABLE_VIEWPORT_KB is called during window shrinking and UIS$ENABLE_KB is called during icon expansion.
UIS$M_ICON_DEF_BODY	UIS places a button AST region over the body of the icon window and uses that AST to trigger icon expansion.
All other bits	The remaining bits are set to *0* and are reserved to DIGITAL.

icon_name

VMS Usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

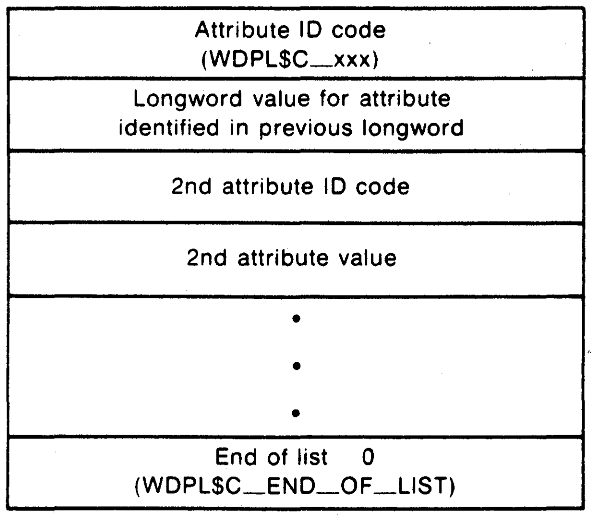
Icon name. The **icon_name** argument is the address of a descriptor of the text to be used as the icon name.

attributes

VMS Usage: **item_list_pair**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Window attributes list. The **attributes** argument is the address of data structure, such as an array or record. You can use the **attributes** argument to specify exact placement of the icon on the display screen.

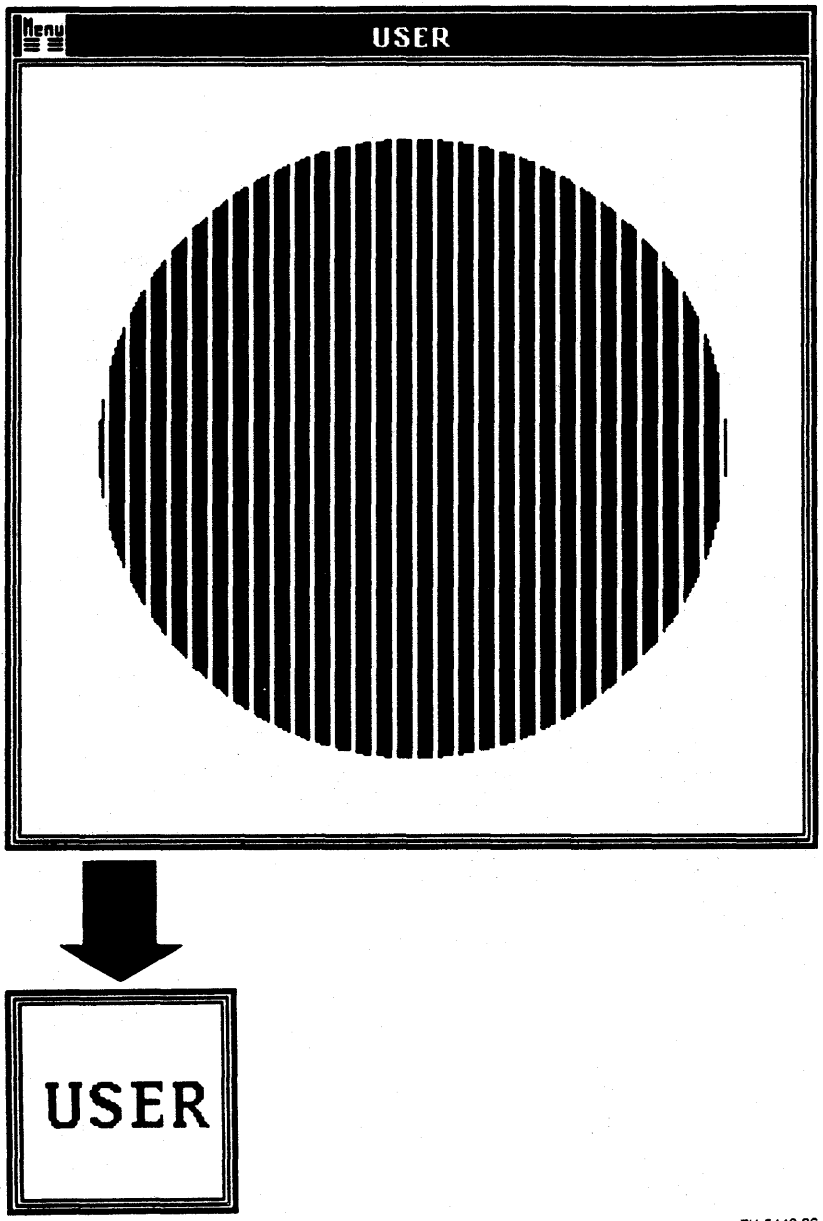
The following figure describes the structure of the window attributes list.



ZK-4581-85

See UIS$CREATE_WINDOW for more information.

screen output



ZK-5448-86

UIS$SOUND_BELL

Actuates the keyboard bell to ring once.

FORMAT **UIS$SOUND_BELL** *devnam [,bell_volume]*

RETURNS UIS$SOUND_BELL signals all errors; no condition values are returned.

ARGUMENTS

devnam

See Section 18.3.9 for more information about this argument.

bell_volume

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Bell volume level. The **bell_volume** argument is the address of a longword that specifies the bell volume. The **bell_volume** argument can be supplied explicitly as a number from *0* to *8*, where *0* is the most quiet; and *8* is the loudest. If the **bell_volume** argument is not specified, the default volume specified in the workstation setup menu is used.

DESCRIPTION On the LK201 keyboard, the bell sound differs from a key click sound in the frequency and tone.

UIS$SOUND_CLICK

Actuates the keyboard click sound once.

FORMAT

UIS$SOUND_CLICK *devnam [,click_volume]*

RETURNS

UIS$SOUND_CLICK signals all errors; no condition values are returned.

ARGUMENTS

devnam

See Section 18.3.9 for more information about this argument.

click_volume

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Key click volume level. The **click_volume** argument is the address of a longword that specifies the key click volume level. The **click_volume** argument is specified explicitly as a number from *0* to *8*, where *0* is the most quiet and *8* is the loudest. If the **click_volume** argument is not specified, the default volume is used from the workstation setup menu mechanism.

DESCRIPTION

On the LK201 keyboard, the key click sound differs from a bell sound in the frequency and tone.

UIS$TEST_KB

Returns a Boolean value indicating whether the physical keyboard is currently bound to the specified virtual keyboard.

FORMAT

status = **UIS$TEST_KB** *kb_id*

RETURNS

VMS Usage:	**Boolean**
type:	**longword**
access:	**write only**
mechanism:	**by value**

Boolean value returned in a status variable or R0 (VAX MACRO). The Boolean value TRUE is returned if the physical keyboard is bound to the virtual keyboard, otherwise a Boolean value FALSE is returned.

UIS$TEST_KB signals all errors; no condition values are returned.

ARGUMENTS

kb_id

See Section 18.3.8 for more information about the **kb_id** argument.

UIS$TEXT

Draws a series of characters. Supports 16-bit text (for example, 2-byte Kanji fonts). To enable 16-bit functions, set the DTYPE field (DSC$B_DTYPE) in the descriptor of the text string to DSC$K_DYPTE_T2.

FORMAT

UIS$TEXT *vd_id, atb, text_string [,x,y] [,ctllist ,ctllen]*

RETURNS

UIS$TEXT signals all errors; no condition values are returned.

ARGUMENTS

vd_id
See Section 18.3.1 for a description of this argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword integer that specifies an attribute block that modifies text output. When a control list is specified, the **atb** argument defines the initial attribute settings of the text string.

text_string

VMS Usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

Text string. The **text_string** argument is the address of a character string descriptor of a text string.

x
y

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Starting point of text output. The x and y arguments are the addresses of f_floating point numbers that define in world coordinates of the starting point of text output. The starting point is the upper-left corner of the character cell of the next character to be drawn.

If this argument is not specified, the current text position is used. (See the UIS$SET_ALIGNED_POSITION routine for more information.)

When a control list is specified, the **x,y** arguments specify the starting coordinate for the first character of the character string.

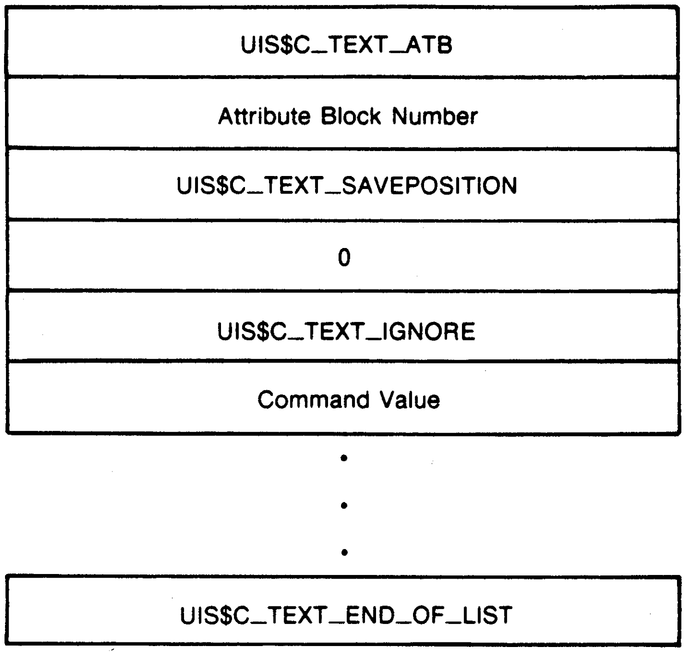
ctllist

VMS Usage:	**vector_longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Text formatting control list. The **ctllist** argument is the address of an array of longwords that define the font, text rendition, text formatting, and positioning of fragments of the text string. When a control list is specified, the **atb** argument defines the initial attribute settings of the text string

If **ctllist** is not specified, text rendition and position are the values specified in the arguments **atb** and **x,y**.

The control list consists of a sequence of data elements, each two longwords in length. The first longword of each element is a tag. The second longword is either a value particular to the type of element specified or zero. Following is a diagram showing the structure of a text control list.



The following table describes valid formatting commands.

Formatting Command	Function
Commands Without Values[1]	
UIS$C_TEXT_NOP	Nil operation
UIS$C_TEXT_RESTORE_POSITION	Restores the current writing position
UIS$C_TEXT_SAVE_POSITION	Saves the current writing position

[1]Second longword must be zero

Formatting Command	Function
Commands Requiring Values	
UIS$C_TEXT_ATB	Specifies an attribute block number
UIS$C_TEXT_HPOS_ ABSOLUTE	Specifies a new current x position
UIS$C_TEXT_HPOS_RELATIVE	Modifies the current x position by a delta
UIS$C_TEXT_IGNORE	Skips *n* characters
UIS$C_TEXT_NEW_LINE	Skips *n* new lines and positions at the left margin
UIS$C_TEXT_TAB_ABSOLUTE	Writes white space to the new absolute position
UIS$C_TEXT_TAB_RELATIVE	Writes white space to the new relative position
UIS$C_TEXT_VPOS_ ABSOLUTE	Writes a new current y position
UIS$C_TEXT_VPOS_RELATIVE	Modifies the current y position by a delta
UIS$C_TEXT_WRITE	Writes *n* characters
Commands Not Requiring a Second Longword	
UIS$C_TEXT_END_OF_LIST	Terminates the control list

When UIS encounters illegal commands and values within the control list, it skips the invalid item and signals an error.

ctllen

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Length of formatting control list. The **ctllen** argument is the address of a longword that specifies the length of the formatting control list in longwords.

DESCRIPTION

Nonprinting characters such as tab and line feed are not handled in any special way. The character is obtained from the font and is displayed like any other character.

UIS$TRANSFORM_OBJECT

Transforms the coordinates or attributes or both of the specified object within the display list.

FORMAT

UIS$TRANSFORM_OBJECT $\left\{ \begin{array}{l} \textit{obj\_id} \\ \textit{seg\_id} \end{array} \right\}$ *[,matrix]* *[,atb]*

RETURNS

UIS$TRANSFORM_OBJECT signals all errors; no condition values are returned.

ARGUMENTS

obj_id

See Section 18.3.3 for a description of this argument.

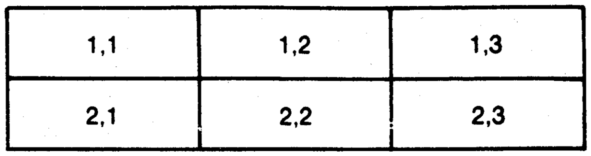
seg_id

See Section 18.3.4 for a description of this argument.

matrix

VMS Usage:	**vector_longword_signed**
type:	**F_floating**
access:	**read only**
mechanism:	**by reference**

Transformation matrix. The **matrix** argument is the address of an array of f_floating point numbers that define the values to be used for scaling, rotation, and/or translation. A two-dimensional array declared as ARRAY(2,3) has the following structure.



ZK-5492-86

VAX FORTRAN allocates memory for the array elements. Memory addresses of array elements range from lowest to highest in the following order: (1,1),(2,1), (1,2),(2,2),(1,3), and (2,3). UIS assigns values to array elements in the order shown in the following illustration.

NOTE: **For the purposes of assigning values to array elements, UIS treats all transformation matrices as VAX FORTRAN arrays regardless of the programming language of the application.**

1	3	5
2	4	6

ZK-5493-86

Pairs of array elements govern how displayed objects are scaled, rotated, and translated. UIS computes the transformed coordinates in the following manner.

$$x_1 = A(1,1)*x + A(1,2)*y + A(1,3)$$
$$y_1 = A(2,1)*x + A(2,2)*y + A(2,3)$$

Translation

When translation alone is performed, the following array elements are assigned values. D_x and D_y represent distances between the original coordinates and the new coordinates.

1	0	Dx
0	1	Dy

ZK-5494-86

Scaling

When scaling alone is performed, the following array elements are assigned values.

Sx	0	0
0	Sy	0

ZK-5495-86

Rotation

When rotation alone is performed, the following array elements are assigned values, where "@" is the desired angle of rotation. The values returned from the FORTRAN SIN and COS functions are stored in the appropriate array elements. (The following example matrix for rotation causes a clockwise rotation of the object.)

cos (@)	sin (@)	0
-sin (@)	cos (@)	0

ZK-5496-86

An unlimited number of transformations can be performed at one time by simply multiplying the matrices together into a single matrix using matrix multiplication.

In order to multiply two matrices together, you must add a row to the bottom of each matrix.

0	0	1

ZK-5461-86

After the multiplication is performed, remove the last row of the result.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword that identifies an attribute block to override current attribute settings.

DESCRIPTION

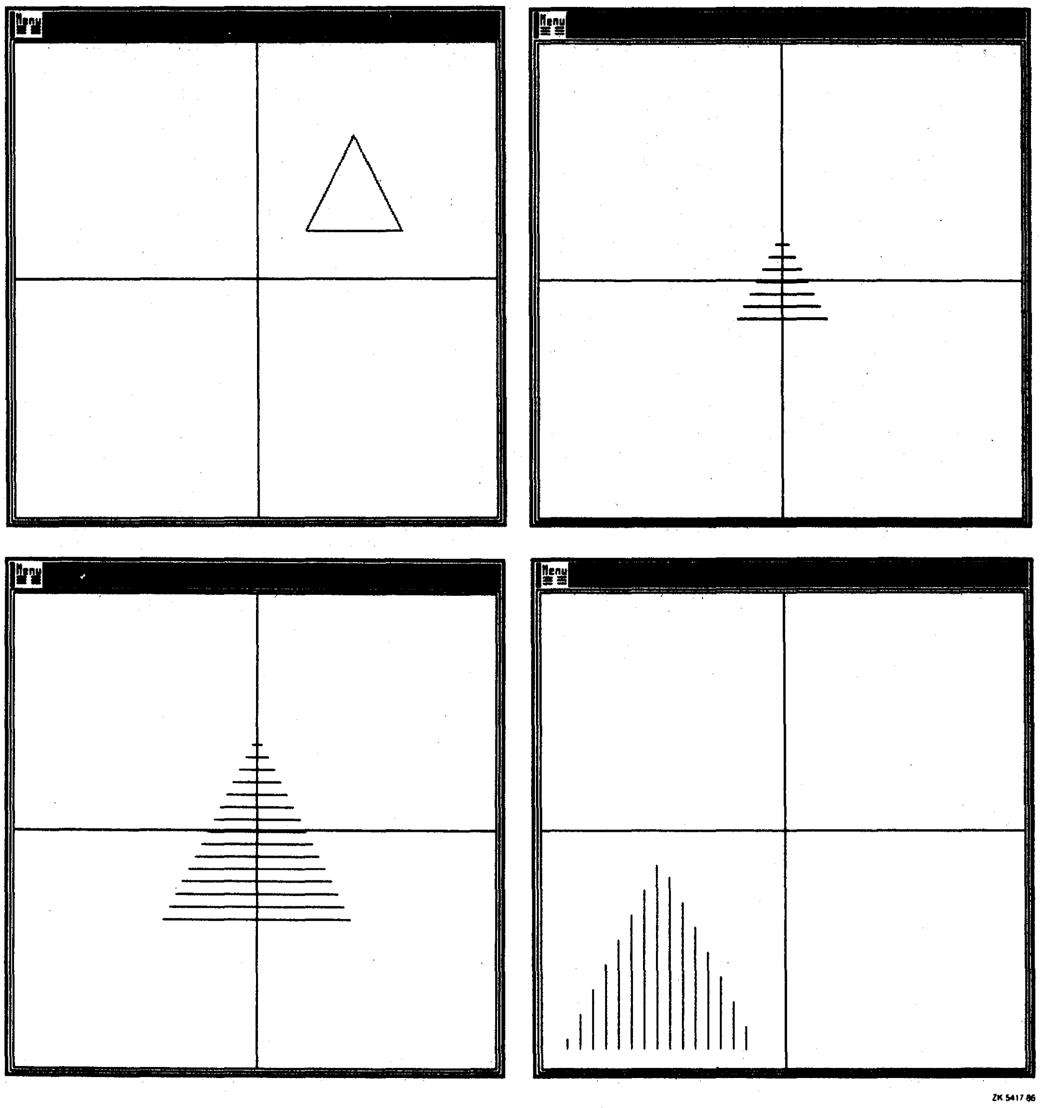
Either the coordinates can be transformed, or the attributes can be overridden or both.

After a transformation, occluded objects might not appear correctly. This can be corrected by calling UIS$EXECUTE to refresh the display screen.

EXAMPLE

```
    .
    .
    .
        REAL*4 MATRIX(2,3)
    .
    .
    .
        CALL UIS$PLOT(VD_ID,0,5.0,5.0,15.0,5.0,10.0,15.0,5.0,5.0)

        CURRENT_ID=UIS$GET_CURRENT_OBJECT(VD_ID)
        OBJ_ID=CURRENT_ID

        CALL UIS$SET_FONT(VD_ID,0,1,'UIS$FILL_PATTERNS')
        CALL UIS$SET_FILL_PATTERN(VD_ID,1,1,PATT$C_HORIZ1_7)
        PAUSE
        MATRIX(1,1)=1.0
        MATRIX(2,1)=0.0
        MATRIX(1,2)=0.0
        MATRIX(2,2)=1.0
        MATRIX(1,3)=-10.0
        MATRIX(2,3)=-10.0
        CALL UIS$TRANSFORM_OBJECT(OBJ_ID,MATRIX,1)

        PAUSE

        MATRIX(1,1)=2.0
        MATRIX(2,1)=0.0
        MATRIX(1,2)=0.0
        MATRIX(2,2)=2.0
        MATRIX(1,3)=0.0
        MATRIX(2,3)=0.0
        CALL UIS$TRANSFORM_OBJECT(OBJ_ID,MATRIX,1)

        PAUSE

        CALL UIS$SET_FONT(VD_ID,0,2,'UIS$FILL_PATTERNS')
        CALL UIS$SET_FILL_PATTERN(VD_ID,2,2,PATT$C_VERT1_7)

        MATRIX(1,1)=1.0
        MATRIX(2,1)=0.0
        MATRIX(1,2)=0.0
        MATRIX(2,2)=1.0
        MATRIX(1,3)=-13.0
        MATRIX(2,3)=-13.0
        CALL UIS$TRANSFORM_OBJECT(OBJ_ID,MATRIX,2)
   .
   .
   .
```

screen output



Part IV UIS Device Coordinate (UISDC) Routines

19 UIS Device Coordinate Graphics Routines

19.1 Overview

This chapter introduces the VMS Workstation Software UISDC (device coordinate) graphics system services. It describes all UISDC routines and how they are used.

19.2 UISDC Routines—How to Use Them

VMS workstation software provides, in addition to the world coordinate interface (UIS), a device-coordinate, or pixel-level, interface (UISDC) to the graphics system services. UISDC gives applications the ability to create UIS windows and to manipulate the contents of those windows at the pixel level.

When an application programs in device coordinates, it must make mixed use of UIS and UISDC routines. Only UIS routines that use or modify world coordinate positions are duplicated as UISDC routines. Most informational, attribute, windowing, and display routines exist only in UIS format and are shared by the two programming levels.

The principal differences between UISDC and UIS follow.

- The UISDC drawing surface is a display window; the UIS drawing surface is a virtual display. Therefore, UISDC output routines use display window identifiers rather than virtual display identifiers.
- Most UISDC positions are expressed in viewport-relative device coordinates.
  - The lower-left corner of the display viewport is pixel (0,0).
  - The upper-right corner is: width multiplied by *x* resolution, height multiplied by *y* resolution, where width and height are expressed in centimeters and resolution is expressed in pixels per centimeter.
- UISDC does not maintain or manage a display list and thus does not support automatic zooming, panning, or display playback.

You can mix use of UIS and UISDC output routines. Therefore, you can perform the following UIS and UISDC operations simultaneously:

- Use virtual coordinates to draw to a virtual display that contains a window.
- Use viewport-relative device coordinates to draw directly to the same window.

Each coordinate system has separate current text positions, character size, text margins, and clipping rectangles.

19.3 Routine Arguments Quick Reference

The UISDC routine descriptions in this chapter refer to the Quick Reference, Section 18.3 in Chapter 18, for more detailed explanations of arguments common to many different routines.

19.4 UISDC Routines and Arguments

Table 19–1 lists each routine and the arguments it uses.

Table 19–1 Routine Arguments

Routine	Arguments
UISDC$ALLOCATE_DOP	wd_id, size, atb
UISDC$CIRCLE	wd_id, atb, center_x, center_y, xradius, start_deg, end_deg
UISDC$ELLIPSE	wd_id, atb, center_x, center_y, xradius, yradius, start_deg, end_deg
UISDC$ERASE	wd_id, x_1, y_1, x_2, y_2
UISDC$EXECUTE_DOP_ASYNCH	wd_id, dop, iosb
UISDC$EXECUTE_DOP_SYNCH	wd_id, dop
UISDC$GET_ALIGNED_POSITION	wd_id, atb, retx, rety
UISDC$GET_CHAR_SIZE	wd_id, atb, char, width, height
UISDC$GET_CLIP	wd_id, atb, x_1, y_1, x_2, y_2
UISDC$GET_POINTER_POSITION	wd_id, retx, rety
UISDC$GET_POSITION	wd_id, retx, rety
UISDC$GET_TEXT_MARGINS	wd_id, atb, x, y, margin_length
UISDC$GET_VISIBILITY	wd_id, x_1, y_1, x_2, y_2
UISDC$IMAGE	wd_id, atb, x_1, y_1, x_2, y_2, rasterwidth, rasterheight, bitsperpixel, rasteraddr
UISDC$LINE	wd_id, atb, x, y
UISDC$LINE_ARRAY	wd_id, atb, count, x_vector, y_vector
UISDC$LOAD_BITMAP	wd_id, bitmap_adr, bitmap_len, bitmap_width, bits_per_pixel
UISDC$MEASURE_TEXT	wd_id, atb, text_string, retwidth, retheight, ctllist, ctllen, posarray
UISDC$MOVE_AREA	wd_id, x_1, y_1, x_2, y_2, new_x, new_y
UISDC$NEW_TEXT_LINE	wd_id, atb
UISDC$PLOT	wd_id, atb, x, y
UISDC$PLOT_ARRAY	wd_id, atb, count, x_vector, y_vector
UISDC$QUEUE_DOP	wd_id, dop

Table 19-1 (Cont.) Routine Arguments

Routine	Arguments
UISDC$READ_IMAGE	wd_id, x_1, y_1, x_2, y_2, rasterwidth, rasterheight, bitsperpixel, rasteraddr, rasterlen
UISDC$SET_ALIGNED_POSITION	wd_id, atb, x, y
UISDC$SET_BUTTON_AST	wd_id, astadr, astprm, keybuf, x_1, y_1, x_2, y_2
UISDC$SET_CHAR_SIZE	wd_id, iatb, oatb, char, width, height
UISDC$SET_CLIP	wd_id, iatb, oatb, x_1, y_1, x_2, y_2
UISDC$SET_POINTER_AST	wd_id, astadr, astprm, x_1, y_1, x_2, y_2, exitastadr, exitastprm
UISDC$SET_POINTER_PATTERN	wd_id, pattern_array, pattern_count, activex, activey, x_1, y_1, x_2, y_2, flags
UISDC$SET_POINTER_POSITION	wd_id, x, y
UISDC$SET_POSITION	wd_id, x, y
UISDC$SET_TEXT_MARGINS	wd_id, iatb, oatb, x, y, margin_length
UISDC$TEXT	wd_id, atb, text_string, x, y, ctllist, ctllen

The following section contains the UISDC routines with their arguments and descriptions.

UISDC$ALLOCATE_DOP

Allocates a DOP (drawing operation primitive) for a particular display window in VAXstation color and intensity systems.

FORMAT

dop = **UISDC$ALLOCATE_DOP** *wd_id ,size ,atb*

RETURNS

VMS Usage:	**address**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the address of the drawing operation primitive in the variable *dop* or R0 (VAX MACRO).

UISDC$ALLOCATE_DOP signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

size

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**modify**
mechanism:	**by reference**

Size of the variable portion of the DOP. The **size** argument is the address of a number that defines the size of the variable portion of the DOP to be allocated.

The size of the variable portion of the allocated DOP is returned in the size field. The size of the allocated DOP may be smaller than the requested size.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of an attribute block.

DESCRIPTION

UISDC$ALLOCATE_DOP writes the following information from the specified attribute block into portions of the DOP data structure and returns the DOP address.

- Clipping rectangle
- Writing mode

- Writing mask

See the *VMS Workstation Software Video Device Driver Manual* for more information.

UISDC$CIRCLE

Draws an arc along the circumference of a circle.

FORMAT

UISDC$CIRCLE *wd_id, atb, center_x, center_y, xradius [,start_deg, end_deg]*

RETURNS

UISDC$CIRCLE signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword integer that specifies an attribute block that controls the appearance of the circle or arc.

center_x
center_y

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Center position x and y viewport-relative device coordinates. The **center_x** and **center_y** arguments are the integer addresses defining a point in the virtual display that is the center of the arc or circle.

xradius

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Radius of the circle specified as an x viewport-relative device coordinate width. The **xradius** argument is the integer address that defines the distance from the center to the circumference of the circle.

start_deg
end_deg

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Degree where the arc starts. The **start_deg** and **end_deg** arguments are the addresses of f_floating numbers that define the start- and end-point on the circumference of the circle where the arc or circle is drawn. Degrees are measured clockwise from the top of the circle. If these arguments are not specified, 0.0 degrees and 360.0 degrees are assumed.

DESCRIPTION

UISDC$CIRCLE draws an arc specified by a center position and a radius for the range of the degrees specified.

The arc is closed by drawing one or more lines between the endpoints. The arc type associated with the attribute block specifies how the arc is closed. The arc is not closed by default. See UISDC$SET_ARC_TYPE for more information.

The points are drawn with the current line pattern and width, and filled with the current fill pattern, if enabled.

UISDC$CIRCLE does not support the following combination of attributes:

- Line width not equal to *1* and line style not equal to $FFFFFFFF_{16}$
- Line width not equal to *1* and complement writing mode

Circles are distorted by virtual display/display window aspect ratio distortion.

UISDC$ELLIPSE

Draws an arc at a starting position along the circumference of an ellipse.

FORMAT **UISDC$ELLIPSE** *wd_id, atb, center_x, center_y, xradius, yradius [,start_deg ,end_deg]*

RETURNS UISDC$ELLIPSE signals all errors; no condition values are returned.

ARGUMENTS

wd_id
See Section 18.3.2 for more information about the **wd_id** argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword that identifies the attribute block that modifies the ellipse. If you specify 0 in the **atb** argument, the default settings of attribute block 0 are used.

center_x
center_y

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Center position x and y viewport-relative device coordinates. The **center_x** and **center_y** arguments are the addresses of integers that define a point in the display window that is the center of the ellipse or arc.

xradius
yradius

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Radius of the ellipse specified as an x and y device coordinate width. The **xradius** argument is the integer address that defines the distance from the ellipse center to the circumference of the ellipse or arc. The **yradius** argument is the integer address that defines the distance from the center of the ellipse to the circumference of the ellipse or arc.

start_deg
end_deg

VMS Usage:	**floating_point**
type:	**f_floating**
access:	**read only**
mechanism:	**by reference**

Degree where the arc starts and ends. The **start_deg** and **end_deg** arguments are the addresses of f_floating numbers that define the starting point and ending point in degrees on the circumference of the ellipse where the arc or ellipse is drawn. Degrees are measured clockwise from the top of the ellipse. If these arguments are not specified, *0.0* and *360.0* degrees are assumed. If neither argument is specified, a complete ellipse is drawn.

DESCRIPTION

UISDC$ELLIPSE uses center position coordinates and x and y radii to construct an ellipse. Along the circumference of this ellipse, UISDC$ELLIPSE draws an arc for a specified range of degrees. To close the arc, draw one or more lines between the endpoints. The type of arc associated with the attribute block specifies the way the arc is closed. See the UISDC$SET_ARC_TYPE routine for more information. The points are drawn with the current line pattern and width and filled with the current fill pattern if enabled. UISDC$ELLIPSE does not create thick patterned ellipses and thick ellipses that are undefined in complement mode.

UISDC$ELLIPSE does not support the following combination of attributes:

- Line width not equal to *1* and line style not equal to $FFFFFFFF_{16}$
- Line width not equal to *1* and complement writing mode

UISDC$ERASE

Erases the specified rectangle in the display window.

FORMAT **UISDC$ERASE** *wd_id [,$x_1$, y_1, x_2, y_2]*

RETURNS UISDC$ERASE signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

x_1, y_1
x_2, y_2

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Viewport-relative device coordinate pairs. The x_1 and y_1 arguments are the integer addresses that define the lower-left corner of the rectangle in the display window. The x_2 and y_2 arguments are the integer addresses that define the upper-right corner of the rectangle in the display window. If no rectangle is specified, the entire display window is erased.

DESCRIPTION Areas within display windows affected by this call are filled with the color specified by entry 0 in the virtual display color map.

UISDC$EXECUTE_DOP_ASYNCH

Starts execution of the specified drawing operation primitive (DOP) in the specified display window of VAXstation color and intensity systems and immediately returns control to the application.

FORMAT

UISDC$EXECUTE_DOP_ASYNCH *wd_id ,dop ,iosb*

RETURNS

UISDC$EXECUTE_DOP_ASYNCH signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

dop

VMS Usage:	**vector_byte_unsigned**
type:	**byte_unsigned**
access:	**read only**
mechanism:	**by reference**

Drawing operation primitive. The **dop** argument is the address of an array of bytes that compose the drawing operation primitive.

iosb

VMS Usage:	**io_status_block**
type:	**quadword (unsigned)**
access:	**write only**
mechanism:	**by reference**

I/O status block. The **iosb** argument is the I/O status block address that receives a value indicating that the DOP is queued for execution.

DESCRIPTION

UISDC$EXECUTE_DOP_ASYNCH queues the specified DOP for execution in the specified window.

You can later use the SYS$SYNCH system service to determine when the DOP has been drawn. See the *VMS Workstation Software Video Device Driver Manual* for more information.

UISDC$EXECUTE_DOP_SYNCH

Queues the drawing operation primitive (DOP), waits for the specified DOP to complete execution in the specified display window, then returns control to the application.

FORMAT

UISDC$EXECUTE_DOP_SYNCH *wd_id ,dop*

RETURNS

UISDC$EXECUTE_DOP_SYNCH signals all errors; no condition values are returned.

ARGUMENTS

wd_id
See Section 18.3.2 for more information about the **wd_id** argument.

dop

VMS Usage:	**vector_byte_unsigned**
type:	**byte_unsigned**
access:	**read only**
mechanism:	**by reference**

Drawing operation primitive. The **dop** argument is the address of an array of bytes that compose the drawing operation primitive.

DESCRIPTION

UISDC$EXECUTE_DOP_SYNCH queues the specified drawing operation primitive for execution in the specified window and returns when the drawing operation is complete.

See the *VMS Workstation Software Video Device Driver Manual* for more information.

UISDC$GET_ALIGNED_POSITION

Returns the current position for text output—the upper-left corner of the next character cell.

FORMAT

UISDC$GET_ALIGNED_POSITION *wd_id, atb, retx, rety*

RETURNS

UISDC$GET_ALIGNED_POSITION signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Attribute block. The **atb** argument is the address of a longword that identifies an attribute block that contains a modified font attribute.

retx
rety

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Viewport-relative device coordinate pair. The **retx** and **rety** arguments are the addresses of longwords that receive the current position as *x* and *y* viewport-relative device coordinate positions.

DESCRIPTION

UISDC$GET_ALIGNED_POSITION differs from UISDC$GET_POSITION in that the current position refers to the upper-left corner of the next character to be output by using the specified attribute block. This feature is useful for applications that require the position of the upper-left corner but do not have enough information about the font baseline to determine the proper alignment point. Applications use the font specified in the given attribute block to convert the position into the proper alignment point. See UISDC$SET_ALIGNED_POSITION.

UISDC$GET_CHAR_SIZE

Returns both a value indicating whether or not character scaling is enabled and the character size used.

FORMAT

Boolean = **UISDC$GET_CHAR_SIZE** *wd_id, atb ,,[,height]*

RETURNS

VMS Usage:	**boolean**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword value returned as a Boolean to indicate the status of character scaling in the variable *Boolean* or R0 (VAX MACRO).

UISDC$GET_CHAR_SIZE signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword that identifies an attribute block that contains the character size attribute setting.

char

VMS Usage:	**char_string**
type:	**character_string**
access:	**write only**
mechanism:	**by descriptor**

Single character. The **char** argument is the address of a character string descriptor of a single char.

width
height

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Character width and height. The **width** argument is the address of a longword that receives the character width in viewport-relative device coordinates. The **height** argument is the address of a longword that receives the character height in viewport-relative device coordinates.

UISDC$GET_CLIP

Returns the clipping mode.

FORMAT

status = **UISDC$GET_CLIP** *wd_id, atb [,x_1, y_1, x_2, y_2]*

RETURNS

VMS Usage:	**boolean**
type:	**(unsigned)**
access:	**write only**
mechanism:	**by value**

Boolean value returned as the clipping mode in a status variable or R0 (VAX MACRO). If clipping is enabled, a Boolean TRUE is returned. If clipping is disabled, a Boolean FALSE is returned.

UISDC$GET_CLIP signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword that identifies the attribute block that modifies the clipping mode.

x_1, y_1
x_2, y_2

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Viewport-relative device coordinate pairs. The **x_1** and **y_1** arguments are the longword addresses that receive the viewport-relative device coordinates of the lower-left corner of the clipping rectangle. The **x_2** and **y_2** arguments are the longword addresses that receive the viewport-relative device coordinates of the upper-right corner of the clipping rectangle.

UISDC$GET_POINTER_POSITION

Returns the current pointer position in viewport-relative device coordinates.

FORMAT

status = **UISDC$GET_POINTER_POSITION** *wd_id, retx, rety*

RETURNS

VMS Usage:	**boolean**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Boolean value returned as the current position of the pointer in a status variable. UISDC$GET_POINTER_POSITION returns the Boolean TRUE value *1* if the pointer is within the visible portion of the viewport; *0* is returned if the pointer is outside the visible portion of the viewport and the *x* and *y* values are returned as 0,0.

UISDC$GET_POINTER_POSITION signals all errors; no condition values are returned.

ARGUMENTS

wd_id
See Section 18.3.2 for more information about the **wd_id** argument.

retx
rety

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Viewport-relative device coordinate pair. The **retx** and **rety** arguments are the addresses of longwords that receive the current position as *x* and *y* viewport-relative device coordinate positions.

DESCRIPTION

Always test the returned status value when you use this routine, since the pointer could be outside the window when the service is called and the x,y values would be meaningless.

UISDC$GET_POSITION

Returns the current baseline position for text output.

FORMAT

UISDC$GET_POSITION *wd_id, retx, rety*

RETURNS

UISDC$GET_POSITION signals all errors; no condition values are returned.

ARGUMENTS

wd_id
See Section 18.3.2 for more information about the **wd_id** argument.

retx
rety

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Viewport-relative device coordinate pair. The **retx** and **rety** arguments are the addresses of longwords that receive the current position of text output in viewport-relative device coordinate positions.

DESCRIPTION

UISDC$NEW_TEXT_LINE and UISDC$TEXT recognize the concept of current position, which refers to the alignment point on the baseline of the next output character. (See the UISDC$SET_POSITION routine.)

UISDC$GET_TEXT_MARGINS

Returns text margins for a line of text. See UISDC$SET_TEXT_MARGINS for more information.

FORMAT

UISDC$GET_TEXT_MARGINS *wd_id ,atb ,x ,y [,margin_length]*

RETURNS

UISDC$GET_TEXT_MARGINS signals all errors; no condition values are returned.

ARGUMENTS

wd_id
See Section 18.3.2 for more information about the **wd_id** argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword that identifies an attribute block.

x
y

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Starting margin position. The **x,y** arguments are the longword addresses that receive the starting margin relative to the direction of text drawing in viewport-relative device coordinates.

margin_length

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Ending margin position. The **margin_length** is the longword address that receives the distance to the end margin in viewport-relative device coordinates. The margin is measured along the actual path of text drawing.

UISDC$GET_VISIBILITY

Returns a Boolean value that indicates whether the specified rectangle in the display window is visible.

FORMAT

Boolean = **UISDC$GET_VISIBILITY** *wd_id [,x_1, y_1 [,x_2, y_2]]*

RETURNS

VMS Usage:	**boolean**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Boolean value returned in a status variable or R0 (VAX MACRO). The returned value, the visibility status, is a Boolean TRUE only if the entire area is visible and a Boolean FALSE if even a portion of the area is occluded or clipped.

UISDC$GET_VISIBILITY signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

x_1, y_1
x_2, y_2

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Viewport-relative device coordinates of a rectangle in the display window. The x_1 and y_1 arguments are integer addresses that define the lower-left corner of a rectangle in the display window. The x_2 and y_2 arguments are integer addresses that define the upper-right corner of a rectangle in the display window.

If you do not specify the coordinates of the rectangle, the dimensions of the entire display window are used by default.

DESCRIPTION

UISDC$GET_VISIBILITY determines if a single position is visible by specifying the same coordinate for both minimum and maximum values.

UISDC$IMAGE

Draws a raster image into a specified display rectangle.

FORMAT

UISDC$IMAGE *wd_id, atb, x_1, y_1, x_2, y_2, rasterwidth, rasterheight, bitsperpixel, rasteraddr*

RETURNS

UISDC$IMAGE signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword that identifies an attribute block that defines the writing mode.

x_1, y_1
x_2, y_2

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Viewport-relative device coordinates of the rectangle in the display window. The **x_1** and **y_1** arguments are the addresses of integers that define the lower-left corner of the rectangle in the display window. The **x_2** and **y_2** arguments are the addresses of integer pixels that define the upper-right corner of the rectangle in the display window.

rasterwidth
rasterheight

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Width and height of the raster image. The **rasterwidth** argument is the address of the longword that defines the width of the raster image in pixels. The **rasterheight** is the address of the longword that defines the height of the raster image in pixels.

bitsperpixel

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Number of bits per pixel in the raster image. The **bitsperpixel** argument is the address of the longword that defines the number of bits per pixel in the raster image. Currently, the **bitsperpixel** argument must be either *1* or *8*.

If **bitsperpixel** is specified as 8 on a single-plane system, the results are unpredictable.

rasteraddr

VMS Usage:	**vector_longword_unsigned**
type:	**longword_unsigned**
access:	**read only**
mechanism:	**by reference**

Raster image. The **rasteraddr** argument is the array address that defines a raster image.

DESCRIPTION

Raster dimensions are described by the width, height, and bits per pixel parameters. Width and height give the number of pixels in each dimension, and bits per pixel represents the number of bits that make up each pixel. The raster is read from memory as *height* bit vectors; each vector is *width* pixels long; each pixel is *bits/pixel* bits long.

UISDC$IMAGE never scales. If the size of the destination rectangle is larger than the size of the raster, the remaining space on the right and top is not written.

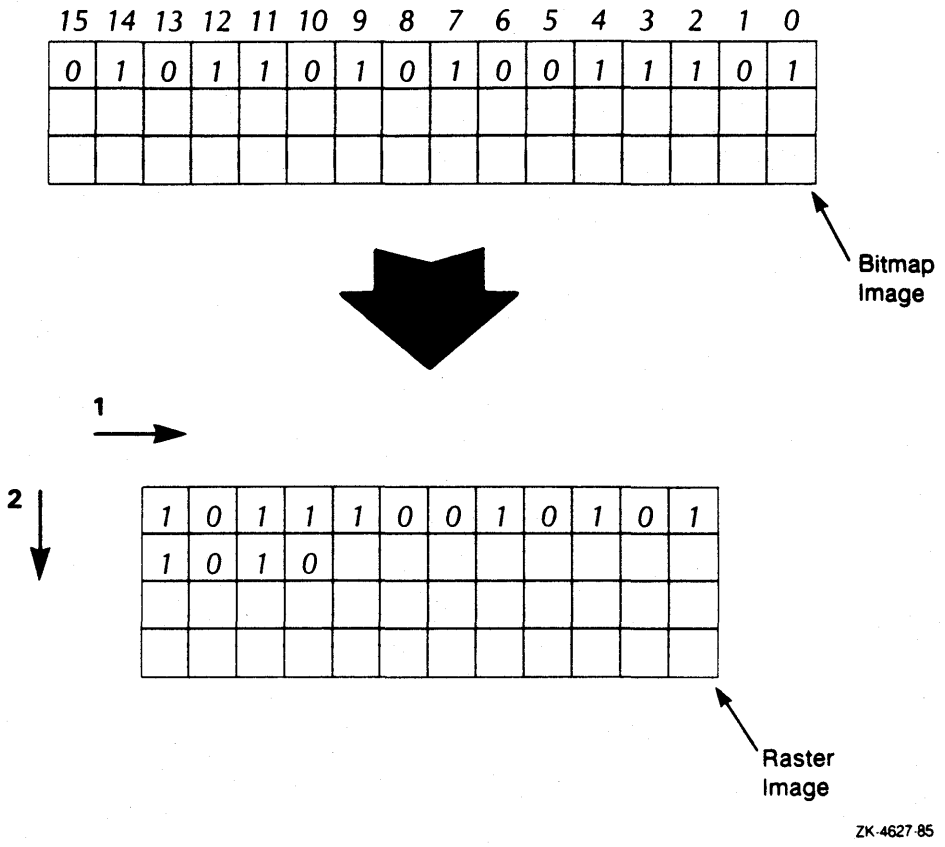
UISDC$IMAGE assigns bits in the bitmap is as follows:

1. Each bit in the array is set from left-most bit to right-most bit.
2. Each row is filled from the top to the bottom row.

NOTE: **The bitmap is not byte- or word-aligned.**

Figure 19–1 illustrates bit setting in the bitmap.

Figure 19–1 Bit Setting in the Bitmap



UISDC$LINE

Draws a line or series of unconnected lines.

FORMAT

UISDC$LINE *wd_id, atb,* x_1, y_1 $[,x_2,y_2$ $[,...x_n,y_n]]$

RETURNS

UISDC$LINE signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword that identifies an attribute block that modifies line style or line width or both.

x

y

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Viewport-relative device coordinate pair. The x and y arguments are integer addresses that define a point in the display window.

If the arguments are repeated to specify a second position, a line is created.

If one coordinate pair is specified, a point is drawn. If any other odd number of coordinate pairs is specified, the final coordinate pair is ignored.

You can specify up to 126 world coordinate pairs as arguments.

DESCRIPTION

If you specify one position, a point is drawn. If you specify two positions, a single vector is drawn.

You can specify up to 252 arguments; that is, it is possible to draw 63 unconnected lines. Use UISDC$LINE_ARRAY to specify a greater number of points in a single call.

Draw points or lines with the line pattern and width for the attribute block. Fill pattern attribute settings are ignored.

UISDC$LINE_ARRAY

Draws an unfilled point, line, or a series of unconnected lines depending on the number of positions specified.

FORMAT

UISDC$LINE_ARRAY *wd_id, atb, count, x_vector, y_vector*

RETURNS

UISDC$LINE_ARRAY signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword that identifies an attribute block that modifies line style or line width or both.

count

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Number of points. The **count** argument is the longword integer address that denotes the number of viewport-relative device coordinate pairs defined in the arguments **x_vector** and **y_vector**.

x_vector
y_vector

VMS Usage:	**vector_longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Array of x and y viewport-relative device coordinates. The **x_vector** and **y_vector** arguments are the integer array addresses whose elements define, in viewport-relative device coordinates, the start- and end-points of lines drawn in the display window.

DESCRIPTION

UISDC$LINE_ARRAY performs the same functions as UISDC$LINE except that in UISDC$LINE_ARRAY, *x* and *y* coordinates are stored in arrays.

You can plot up to 32,767 points in a single call. UISDC$LINE_ARRAY is the same as UISDC$LINE except that in UISDC$LINE_ARRAY, *x* and *y* coordinates are specified using two arrays, each *count* points long.

UISDC$LOAD_BITMAP

Loads a bitmap into offscreen memory on VAXstation color and intensity systems.

FORMAT

bitmap_id = **UISDC$LOAD_BITMAP** *wd_id ,bitmap_adr ,bitmap_len ,bitmap_width ,bits_per_pixel*

RETURNS

VMS Usage:	**identifier**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword value returned as the bitmap identifier in the variable *bitmap_id* or R0 (VAX MACRO) for use in DOP$L_BITMAP_ID field of a drawing operation primitive (DOP).

UISDC$LOAD_BITMAP signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

bitmap_adr

VMS Usage:	**address**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Bitmap address. The **bitmap_adr** argument is the address of a bitmap.

bitmap_len

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Bitmap length. The **bitmap_len** argument is the address of the number that defines the length of the bitmap in bytes. The length must be a multiple of 2.

bitmap_width

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Width of the bitmap. The **bitmap_width** argument is the address of a number that defines the width of the bitmap in pixels. If the number of bits per pixel is 1, the specified width must be a multiple of 16.

If the width of the bitmap should not exceed 1024.

bits_per_pixel

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

The **bits_per_pixel** argument is the address of a number that defines the number of bits per pixel. Currently, the values *1* and *8* are supported.

DESCRIPTION

See the *VMS Workstation Software Video Device Driver Manual* for more information.

UISDC$MEASURE_TEXT

Measures a text string as if it were output in a display window.

FORMAT

UISDC$MEASURE_TEXT *wd_id, atb, text_string, retwidth, retheight, [,ctllist, ctllen] [,posarray]*

RETURNS

UISDC$MEASURE_TEXT signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword that identifies an attribute block that modifies text output.

text_string

VMS Usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

Text string. The **text_string** argument is the address of a character string descriptor of a text string.

retwidth
retheight

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Width and height of the text string. The **retwidth** and **retheight** arguments are longword addresses that receive the width and height of the text in pixels.

ctllist

VMS Usage:	**vector_longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Text formatting list. The **ctllist** argument is the address of a longword array that describes the font, text rendition, format, and positioning of text string fragments. See UISDC$TEXT for a complete description of the formatting control list.

ctllen

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Length of the text formatting control list. The **ctllen** argument is the address of a longword that defines the length of the text formatting control list.

posarray

VMS Usage:	**vector_longword_unsigned**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by reference**

Character position array. The **posarray** argument is the longword array address that receives character positions in pixels that are relative offsets where each character is displayed. See UISDC$MEASURE_TEXT for a complete description of the character position array.

DESCRIPTION

Use UISDC$MEASURE_TEXT in justification and text-positioning applications. UISDC$MEASURE_TEXT returns the height and width of the text string in viewport-relative device coordinates.

UISDC$MOVE_AREA

Shifts a portion of a display window to another position in the window.

FORMAT

UISDC$MOVE_AREA *wd_id, x_1, y_1, x_2, y_2, new_x, new_y*

RETURNS

UISDC$MOVE_AREA signals all errors; no condition values are returned.

ARGUMENTS

wd_id
See Section 18.3.2 for more information about the **wd_id** argument.

x_1, y_1
x_2, y_2

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Viewport-relative device coordinates of the source rectangle. The x_1 and y_1 arguments are the addresses of integers that define the lower-left corner of the source rectangle. The x_2 and y_2 are the addresses of integers that define the upper-right corner of the source rectangle.

new_x
new_y

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Viewport-relative device coordinate pair. The **new_x** and **new_y** arguments are the addresses of integers that define the lower-left corner of the destination rectangle. The height and width of the destination rectangle is implied from the height and width of the source rectangle.

DESCRIPTION

Some display objects are only partially contained within the specified source rectangle but are partially moved within existing display windows. These display objects will be moved completely within the display list.

The nonoccluding portion of the source rectangle (if any) is erased after the operation.

UISDC$NEW_TEXT_LINE

Moves the current text position along the actual text path, drawing to the starting margin, then along the margin in the direction of the minor text path. Depending on the minor text path, either the width or height of the character cell is used for spacing between characters and lines.

FORMAT

UISDC$NEW_TEXT_LINE *wd_id, atb*

RETURNS

UISDC$NEW_TEXT_LINE signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword that identifies an attribute block.

UISDC$PLOT

Draws a filled or unfilled point, line, or polygon depending on the number of positions specified.

FORMAT

UISDC$PLOT *wd_id, atb,* x_1, y_1 *[,*x_2,y_2 *[,...*x_n,y_n*]]*

RETURNS

UISDC$PLOT signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword that identifies an attribute block that modifies line style and line width.

x

y

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Viewport-relative device coordinate pair. The **x** and **y** arguments are the addresses of integers that define a point in the display window. If the argument is used to specify a second position, a line is created. Up to 126 viewport-relative device coordinate pairs can be specified as arguments. See the Description section for more information about this argument.

DESCRIPTION

If you specify one position, a point is drawn. If you specify two positions, a single vector is drawn. If you specify more than two positions, a connected polygon is drawn. You can specify up to 252 arguments; this routine gives a maximum 126-point polygon. If you must specify a larger number of points in a single call, use UISDC$PLOT_ARRAY.

You draw points or lines with the line pattern and width for the attribute block. If FILL is enabled for the attribute block, the enclosed area is filled with the current fill pattern.

NOTE: **For VAX PASCAL application programs that draw lines and polygons, use UISDC$PLOT_ARRAY.**

UISDC$PLOT_ARRAY

Draws an unfilled or filled point, line, or polygon depending on the number of positions specified. This routine performs the same functions as UISDC$PLOT.

FORMAT

UISDC$PLOT_ARRAY *wd_id, atb, count, x_vector, y_vector*

RETURNS

UISDC$PLOT_ARRAY signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword that identifies an attribute block that modifies line style or line width or both.

count

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Number of points. The **count** argument is the address of the longword that denotes the number of viewport-relative device coordinate pairs defined in the **x_vector** and **y_vector** arguments.

x_vector
y_vector

VMS Usage:	**vector_longword_signed**
type:	**longword_signed**
access:	**read only**
mechanism:	**by reference**

Array of x and y viewport-relative device coordinates. The **x_vector** argument is the integer array address whose elements are the x viewport-relative device coordinate values of points defined in the window display. The **y_vector** argument is the integer array address whose elements are the y viewport-relative device coordinate values of points defined in the display window.

DESCRIPTION

You can plot a maximum of 65,535 points in a single call. UISDC$PLOT_ARRAY is the same as UISDC$PLOT, except that you specify the *x* and *y* viewport-relative device coordinates using two arrays, each of length *n* points.

UISDC$QUEUE_DOP

Queues the specified drawing operation primitive (DOP) for execution in the specified window and then returns control to the application.

FORMAT

UISDC$QUEUE_DOP *wd_id ,dop*

RETURNS

UISDC$QUEUE_DOP signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

dop

VMS Usage:	**vector_byte_unsigned**
type:	**byte_unsigned**
access:	**read only**
mechanism:	**by reference**

Drawing operation primitive. The **dop** argument is the address of an array of bytes that contains the drawing operation primitive.

DESCRIPTION

UISDC$EXECUTE_DOP_ASYNCH queues the specified DOP for execution in the specified window. To obtain notification that the DOP has completed execution, see UISDC$EXECUTE_DOP_ASYNCH and UISDC$EXECUTE_DOP_SYNCH. See the *VMS Workstation Software Video Device Driver Manual* for more information about DOPs.

UISDC$READ_IMAGE

Reads a raster image from within a specified rectangle contained by a display window.

FORMAT **UISDC$READ_IMAGE** *wd_id, x_1, y_1, x_2, y_2, rasterwidth, rasterheight, bitsperpixel,[rasteraddr], rasterlen*

RETURNS UISDC$READ_IMAGE signals all errors; no condition values are returned.

ARGUMENTS

wd_id
See Section 18.3.2 for more information about the **wd_id** argument.

x_1
y_1

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Viewport-relative device coordinates of lower-left and upper-right corners of the specified rectangle. The x_1,y_1 arguments are the addresses of integers that define the lower-left corner of the rectangle in the display window. The x_2,y_2 arguments are the addresses of integers that define the upper-right corner of the specified rectangle in the display window.

rasterwidth
rasterheight

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Width and height in pixels of the raster image. The **rasterwidth** argument is the address of a longword that receives the width of the raster image in pixels. The **rasterheight** argument is the address of a longword that receives the height of the raster image in pixels.

bitsperpixel

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**write only**
mechanism:	**by reference**

Number of bits per pixel in the raster image. The **bitsperpixel** argument is the address of a longword that receives the number of bits per pixel in the raster image.

rasteraddr

VMS Usage:	**vector_byte_unsigned**
type:	**byte (unsigned)**
access:	**write only**
mechanism:	**by reference**

Address of buffer in which to return the raster image. The **rasteraddr** argument is the address of an array of bytes that receives the raster image.

rasterlen

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Size in bytes of the buffer. The **rasterlen** argument is the address of a longword that specifies the size in bytes of the buffer.

DESCRIPTION

The raster image contained within the rectangle described by x_1, y_1 and x_2, y_2 is returned in the specified buffer. The actual dimensions, in pixels, of the returned buffer is written to rasterwidth and rasterheight. The number of bits per pixel is written to bitsperpixel. If the size of the buffer specified by rasterlen is not large enough to accept the entire bitmap raster, then rasterwidth, rasterheight, and bitsperpixel are returned as 0 and no data is written to the buffer.

If you specify the buffer length as *0*, values are returned in **rasterwidth**, **rasterheight**, and **bitsperpixel**. Use these values to calculate the size of the buffer you need to contain the raster image. Specify a buffer length of *0* to obtain the width, height, and bits per pixel. Use these returned values to do the following:

1. Calculate the correct buffer size
2. Reissue the call with the correct data

UISDC$SET_ALIGNED_POSITION

Sets the current position for text output. This routine differs from UISDC$SET_POSITION in that the position refers to the upper-left corner of the next character to the output.

FORMAT

UISDC$SET_ALIGNED_POSITION *wd_id, atb, x, y*

RETURNS

UISDC$SET_ALIGNED_POSITION signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword that identifies an attribute block that contains the appropriate font attribute text attribute setting.

x
y

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Viewport-relative device coordinate pair. The **x** and **y** arguments are the addresses of integers that define the current position for text output.

DESCRIPTION

UISDC$SET_ALIGNED_POSITION is useful in applications that know the position of the upper-left corner but do not know enough about the font baseline to determine the proper alignment point. The position is converted into the proper alignment point using the font specified in the given attribute block. The alignment point is stored internally.

UISDC$SET_BUTTON_AST

Allows an application to determine when a button on the pointing device is depressed or released in a given rectangle of the display window.

FORMAT

UISDC$SET_BUTTON_AST *wd_id [,astadr, [astprm] ,keybuf] [,x_1, y_1, x_2, y_2]*

RETURNS

UISDC$SET_BUTTON_AST signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

astadr

VMS Usage:	**ast_procedure**
type:	**procedure entry mask**
access:	**read only**
mechanism:	**by reference**

AST routine. The **astadr** argument is the entry mask address to a procedure called at AST level whenever you release or depress a pointer button. To cancel the AST-enabling request of UISDC$SET_BUTTON_AST, specify *0* in the **astadr** argument. To disable UIS$SET_BUTTON_AST, omit the **astadr** argument.

astprm

See Section 18.3.7 for more information about the **astprm** argument.

keybuf

VMS Usage:	**address**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by reference**

Key buffer. The **keybuf** argument is the address of a longword buffer that receives button information whenever a pointer button is depressed or released. The low two bytes are the key code. The buttons are located on the left, center, and right of the pointing device and are defined as UISDC$C_POINTER_BUTTON_1, UISDC$C_POINTER_BUTTON_2, UISDC$C_POINTER_BUTTON_3, and UISDC$C_POINTER_BUTTON_4, respectively. Bit <31> is set to *1* if the button has been pressed, and to *0* if the button has been released. The buffer is not overwritten with subsequent button transitions until the AST routine completes.

x_1, y_1
x_2, y_2

VMS Usage:	**longword_signed**
type:	**longword (signed)**

access: **read only**
mechanism: **by reference**

Viewport-relative device coordinates of a rectangle in the display window. The x_1 and y_1 arguments are the addresses of integers that define the lower-left corner of a rectangle in the display window. The x_2 and y_2 arguments are the addresses of integer pixels that define the upper-right corner of a rectangle in the display window.

If no rectangle is specified, the entire display window is assumed.

DESCRIPTION

This function can be called any number of times for different rectangles within the same display window or many display windows.

See the Description section of UISDC$SET_BUTTON_AST for information about pointer region priorities.

UISDC$SET_CHAR_SIZE

Sets the viewport-relative device coordinate size of the specified character.

FORMAT

UISDC$SET_CHAR_SIZE *wd_id, iatb, oatb,,[,width][,height]*

RETURNS

UISDC$SET_CHAR_SIZE signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

iatb

See Section 18.3.5 for more information on the **iatb** argument.

oatb

See Section 18.3.6 for more information on the **oatb** argument.

char

VMS Usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

Single character. The **char** argument is a single character descriptor address. You can specify any character in the font. Choose this argument when you use proportionally spaced fonts to establish spacing and scaling factors among characters within the font. The **char** argument has no effect on monospaced fonts.

If you do not specify **char** or if the specified character is invalid, the widest character in the font is chosen.

width
height

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Character width and height. The **width** argument is the address of an integer that defines the character width in viewport-relative device coordinates. The **height** argument is the address of an integer that defines the character height in viewport-relative device coordinates.

DESCRIPTION

To disable character scaling, omit the **char**, **width**, and **height** arguments.

To scale characters to their nominal size as specified in the font, do not specify **width** or **height**. Scaling is visible only when you use a window that does not have the same proportions as the virtual display.

If you specify only **width** or **height**, characters are scaled to the size you specify and in the direction you specify. In the unspecified direction, characters are scaled to maintain the same ratio of height and width as the unscaled character.

Note that this routine does not change the size of only one character. Rather, all characters in the font are scaled to the direct proportion.

UISDC$SET_CLIP

Sets a clipping rectangle within the display window.

FORMAT

UISDC$SET_CLIP *wd_id, iatb, oatb [,x_1, y_1, x_2, y_2]*

RETURNS

UISDC$SET_CLIP signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

iatb

See Section 18.3.5 for more information on the **iatb** argument.

oatb

See Section 18.3.6 for more information on the **oatb** argument.

x_1, y_1
x_2, y_2

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Viewport-relative device coordinates of the clipping rectangle. The x_1 and y_1 arguments are the addresses of integers that define the lower-left corner of the clipping rectangle in viewport-relative device coordinates. The x_2 and y_2 arguments are the addresses of integers that define the upper-right corner of the clipping rectangle in viewport-relative device coordinates. Only graphic objects and portions of graphic objects drawn **within** the clipping rectangle are seen.

If you do not specify the device coordinates of the clipping rectangle corners, clipping is disabled for this attribute block.

UISDC$SET_POINTER_AST

Allows an application to determine when the pointer is moved in a given rectangle of the display window.

FORMAT **UISDC$SET_POINTER_AST** *wd_id [,astadr [,astprm]] [,x_1, y_1, x_2, y_2] [,exitastadr [,exitastprm]]*

RETURNS UISDC$SET_POINTER_AST signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

astadr

VMS Usage:	**ast_procedure**
type:	**procedure entry mask**
access:	**read only**
mechanism:	**by reference**

AST routine. The **astadr** argument is the entry mask address to a procedure called at AST level whenever you move the pointer within a rectangle in the display window.

To cancel the AST-enabling request of UISDC$SET_POINTER_AST for this argument only, specify 0 in the **astadr** argument and the coordinates of the rectangle.

astprm

See Section 18.3.7 for more information on the **astprm** argument.

x_1, y_1
x_2, y_2

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Viewport-relative device coordinates of the rectangle of the display window. The x_1 and y_1 arguments are the addresses of integers that define the lower-left corner of the rectangle of the display window. The x_2 and y_2 arguments are the addresses of integer pixels that define the upper-right corner of the rectangle of the display window.

If no rectangle is specified, the entire display window is assumed.

To cancel an AST-enabling request, specify *0* in either the **astadr** or the **exitastadr** arguments, or both, and the coordinates of the rectangle.

exitastadr

VMS Usage:	**ast_procedure**
type:	**procedure entry mask**
access:	**read only**
mechanism:	**by reference**

Exit AST routine. The **exitastadr** argument is the address of the entry mask to a procedure called at AST level whenever the pointer leaves the rectangle.

To cancel the AST-enabling request of UISDC$SET_POINTER_AST for the EXIT AST routine only, specify *0* in the **exitastadr** argument and the coordinates of the rectangle.

exitastprm

VMS Usage:	**user_arg**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Exit AST parameter. The **exitastprm** argument is the address of a single argument or data structure, such as an array or record, to be passed to the AST routine. Calls to UISDC$SET_POINTER_AST in VAX FORTRAN application programs should be coded as follows: %REF(%LOC(exitastprm)).

DESCRIPTION

UISDC$SET_POINTER_AST also allows an application to track the pointer in its own way. This routine can be called any number of times for different rectangles.

Note that an application need not enable both AST routines. It can specify one or the other.

The application can use UISDC$SET_POINTER_AST to highlight the display or some other application-specific function, when you move the pointer over specific areas of the display window. You might use this feature to define a number of regions within a menu and execute an AST when the pointer enters or leaves any of these regions.

If both AST routines are enabled and the value *0* is specified in the **astadr** argument, the first AST routine is canceled.

See the Description section of UISDC$SET_BUTTON_AST for information about pointer region priorities.

UISDC$SET_POINTER_PATTERN

Allows an application to specify a special pointer cursor pattern.

FORMAT **UISDC$SET_POINTER_PATTERN** *wd_id [,pattern_array, pattern_count, activex, activey] [x_1, y_1, x_2, y_2] [flags]*

RETURNS UISDC$SET_POINTER_PATTERN signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

pattern_array

VMS Usage:	**vector_word_unsigned**
type:	**word (unsigned)**
access:	**read only**
mechanism:	**by reference**

The 16- x 16-bit cursor pattern. The **pattern_array** argument is the address of one or more arrays of 16 words that represent a bitmap image of the cursor.

Color and intensity applications can define two patterns that are also executable on monochrome systems.

If two arrays are specified in an application that runs on a single-plane system, the first array is used.

NOTE: **The bitmap image of the new pointer pattern is mapped in reverse order to the display screen.**

pattern_count

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Number of 16- x 16-bit cursor pattern. The **pattern_count** argument is the address of a longword that contains the number of cursor pattern arrays defined in the **pattern_array** argument.

activex
activey

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

The **activex** and **activey** arguments specify the actual bit in the cursor pattern that should be used to calculate the current pointer position. The arguments are expressed as bit offsets from the lower-left corner of the cursor pattern.

x_1, y_1
x_2, y_2

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Viewport-relative device coordinates of the rectangle in the display window. The x_1 and y_1 arguments are the addresses of integers that define the lower-left corner of the rectangle in the display window. The x_2 and y_2 arguments are the addresses of integer pixels that define the upper-right corner of the rectangle in the display window.

flags

VMS Usage:	**longword_mask**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Flags. The **flags** argument is the address of a longword mask whose bits determine whether or not the cursor is confined to the display window rectangle.

UISDC$M_BIND_POINTER sets the appropriate bit in the mask.

DESCRIPTION

UISDC$SET_POINTER_PATTERN allows an application to specify a special pointer pattern to be used when the pointer is within the display window region specified by the optional rectangle. If no rectangle is given, the entire display window is assumed. You can call this function any number of times for different rectangles.

To disable UISDC$SET_POINTER_PATTERN, omit the **pattern_array, pattern_count**, **activex**, and **activey** arguments.

See the Description section of UISDC$SET_BUTTON_AST for information about pointer region priorities.

UISDC$SET_POINTER_POSITION

Specifies a new current pointer position in device coordinates. It is only effective if the new pointer position is visible within the specified display window.

FORMAT

status = **UISDC$SET_POINTER_POSITION** *wd_id, x, y*

RETURNS

Longword value returned as Boolean in the variable *status* or R0 (VAX MACRO) to indicate that the position is set.

UISDC$SET_POINTER_POSITION signals all errors; no condition values are returned.

ARGUMENTS

wd_id
See Section 18.3.2 for more information about the **wd_id** argument.

x
y

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Viewport-relative device coordinates of the new pointer position. The **x** and **y** arguments are the addresses of integers that define the new pointer position.

UISDC$SET_POSITION

Sets the current position for text output. The current position is the point of alignment on the baseline of the next output character.

FORMAT **UISDC$SET_POSITION** *wd_id, x,y*

RETURNS UISDC$SET_POSITION signals all errors; no condition values are returned.

ARGUMENTS

wd_id
See Section 18.3.2 for more information about the **wd_id** argument.

x
y

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Viewport-relative device coordinate pair. The **x** and **y** arguments are the addresses of integers that define the current position for text output.

UISDC$SET_TEXT_MARGINS

Sets the text margins for a line of text.

FORMAT

UISDC$SET_TEXT_MARGINS *wd_id ,iatb ,oatb ,x ,y ,margin_length*

RETURNS

UISDC$SET_TEXT_MARGINS signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

iatb

See Section 18.3.5 for more information on the **iatb** argument.

oatb

See Section 18.3.6 for more information on the **oatb** argument.

x
y

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Starting margin position. The **x,y** arguments are the addresses of integers that define a point on the starting margin in viewport-relative device coordinates. The starting margin is the minor text path when the angle of text slope equals 0 degrees.

margin_length

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Ending margin position. The **margin_length** is the address of a number that defines the distance from the starting margin to the end margin in viewport-relative device coordinates.

UISDC$TEXT

Draws a series of encoded characters. Supports 16-bit text (for example, 2-byte Kanji fonts). To enable 16-bit text functions, set the DTYPE field (DSC$B_DTYPE) in the descriptor of the text string to DSC$K_DTYPE_T2.

FORMAT

UISDC$TEXT *wd_id, atb, text_string [,x,y] [,ctllist ,ctllen]*

RETURNS

UISDC$TEXT signals all errors; no condition values are returned.

ARGUMENTS

wd_id

See Section 18.3.2 for more information about the **wd_id** argument.

atb

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Attribute block number. The **atb** argument is the address of a longword that identifies an attribute block that modifies text output.

text_string

VMS Usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

Text string. The **text_string** argument is the address of a text string character string descriptor.

x
y

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Viewport-relative device coordinates pair. The **x** and **y** arguments are the addresses of integers that define the viewport-relative device coordinates of the starting point of text output at the upper-left corner of the character cell.

If this argument is not specified, the current text position is used. (See the UISDC$SET_ALIGNED_POSITION routine for more information.)

ctllist

VMS Usage:	**vector_longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Text control formatting list. The **ctllist** argument is the address of an array of longwords that describe the font, text rendition, format, and positioning of text string fragments. See UISDC$TEXT for a complete description of the text formatting control list.

ctllen

VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by reference**

Length of the text formatting control list. The **ctllen** argument is the address of an integer that defines the length of the text formatting control list in longwords.

DESCRIPTION

Nonprinting characters such as tab and linefeed are not handled in any special way. The character is obtained from the font and displayed like any other character.

A UIS CALLING SEQUENCES SUMMARY

Table A–1 lists return values, entry point names, and parameter lists of all UIS routines.

Table A–1 UIS Calling Sequences

Return Value	Routine	Argument List
seg_id	UIS$BEGIN_SEGMENT	vd_id
	UIS$CIRCLE	vd_id, atb, center_x, center_y, xradius, [,start_deg] [,end_deg]
	UIS$CLOSE_WINDOW	wd_id
copy_id	UIS$COPY_OBJECT[1]	$\left\{ \begin{array}{c} \text{obj\_id} \\ \text{seg\_id} \end{array} \right\}$ [,matrix] [,atb]
vcm_id	UIS$CREATE_COLOR_MAP	vcm_size [,vcm_name] [,vcm_attributes]
cms_id	UIS$CREATE_COLOR_MAP_SEG	vcm_id, [,devnam] [,place_mode] [,place_data]
vd_id	UIS$CREATE_DISPLAY	x_1, y_1, x_2, y_2, width, height [,vcm_id]
kb_id	UIS$CREATE_KB	devnam
	UIS$CREATE_TERMINAL	termtype [,title] [,attributes] [,devnam] [,devlen]
tb_id	UIS$CREATE_TB	devname
tr_id	UIS$CREATE_TRANSFORMATION	vd_id, x_1, y_1, x_2, y_2 [vdx_1, vdy_1, vdx_2, vdy_2]
wd_id	UIS$CREATE_WINDOW	vd_id, devnam [,title] [,x_1, y_1, x_2, y_2] [, width,height] [,attributes]
	UIS$DELETE_COLOR_MAP	vcm_id
	UIS$DELETE_COLOR_MAP_SEG	cms_id
	UIS$DELETE_DISPLAY	vd_id
	UIS$DELETE_KB	kb_id
	UIS$DELETE_OBJECT[1]	$\left\{ \begin{array}{c} \text{obj\_id} \\ \text{seg\_id} \end{array} \right\}$
	UIS$DELETE_PRIVATE	obj_id
	UIS$DELETE_TB	tb_id
	UIS$DELETE_TRANSFORMATION	tr_id
	UIS$DELETE_WINDOW	wd_id

[1]VAX PASCAL and VAX PL/I applications must specify the **obj_id** argument.

Table A-1 (Cont.) UIS Calling Sequences

Return Value	Routine	Argument List
	UIS$DISABLE_DISPLAY_LIST	vd_id [,display_flags]
	UIS$DISABLE_KB	kb_id
	UIS$DISABLE_TB	tb_id
	UIS$DISABLE_VIEWPORT_KB	wd_id
	UIS$ELLIPSE	vd_id, atb, center_x, center_y, xradius, yradius, [,start_deg] [,end_deg]
	UIS$ENABLE_DISPLAY_LIST	vd_id [,display_flags]
	UIS$ENABLE_KB	kb_id [,wd_id]
	UIS$ENABLE_TB	tb_id
	UIS$ENABLE_VIEWPORT_KB	kb_id, wd_id
	UIS$END_SEGMENT	vd_id
	UIS$ERASE	vd_id [,x_1 y_1, x_2, y_2]
	UIS$EXECUTE	vd_id [,buflen] [,bufaddr]
vd_id	UIS$EXECUTE_DISPLAY	buflen, bufaddr
	UIS$EXPAND_ICON	wd_id [,icon_wd_id] [,attributes]
	UIS$EXTRACT_HEADER	vd_id [,buflen, bufaddr] [,retlen]
	UIS$EXTRACT_OBJECT[1]	{ obj_id / seg_id } [,buflen ,bufaddr] [,retlen]
	UIS$EXTRACT_PRIVATE[1]	{ obj_id / seg_id } [,buflen ,bufaddr] [,retlen]
	UIS$EXTRACT_REGION	vd_id [,x_1,y_1,x_2,y_2] [,buflen ,bufaddr] [,retlen]
	UIS$EXTRACT_TRAILER	vd_id [,buflen, bufaddr] [,retlen]
obj_id	UIS$FIND_PRIMITIVE	vd_id, x_1, y_1,x_2,y_2 [,context] [,extent]
seg_id	UIS$FIND_SEGMENT	vd_id, x_1, y_1, x_2, y_2, [,context] [,extent]
	UIS$GET_ABS_POINTER_POS	devnam, retx, rety
	UIS$GET_ALIGNED_POSITION	vd_id, atb, retx, rety
arc_type	UIS$GET_ARC_TYPE	vd_id, atb
index	UIS$GET_BACKGROUND_INDEX	vd_id, atb
status	UIS$GET_BUTTONS	wd_id, retstate
angle	UIS$GET_CHAR_ROTATION	vd_id, atb
boolean	UIS$GET_CHAR_SIZE	vd_id, atb [,char], [width, height]
angle	UIS$GET_CHAR_SLANT	vd_id, atb

[1]VAX PASCAL and VAX PL/I applications must specify the **obj_id** argument.

Table A-1 (Cont.) UIS Calling Sequences

Return Value	Routine	Argument List
	UIS$GET_CHAR_SPACING	vd_id, atb, dx, dy
status	UIS$GET_CLIP	vd_id, atb [,x_1, y_1, x_2, y_2]
	UIS$GET_COLOR	vd_id, index, retr, retg, retb [,wd_id]
	UIS$GET_COLORS	vd_id, index, count, retr_vector, retg_vector, retb_vector [,wd_id]
current_id	UIS$GET_CURRENT_OBJECT	vd_id
	UIS$GET_DISPLAY_SIZE	devnam, retwidth, retheight [,retresolx, retresoly] [,retpwidth retpheight]
status	UIS$GET_FILL_PATTERN	vd_id, atb [,index]
	UIS$GET_FONT	vd_id, atb, bufferdesc [,length]
	UIS$GET_FONT_ATTRIBUTES	vd_id, ascender, descender, height, [,maximum_width] [,item_list]
	UIS$GET_FONT_SIZE	fontid, text_string, retwidth, retheight
	UIS$GET_HW_COLOR_INFO	devnam [,type] [,indices] [,colors] [,maps] [,rbits] [,gbits] [,bbits] [,ibits] [,res_indices] [,regen]
	UIS$GET_INTENSITIES	vd_id, index, count, reti_vector [,wd_id]
	UIS$GET_INTENSITY	vd_id, index, reti [,wd_id]
	UIS$GET_KB_ATTRIBUTES	kb_id [,enable_items] [,disable_items] [,click_volume]
style	UIS$GET_LINE_STYLE	vd_id, atb
width	UIS$GET_LINE_WIDTH	vd_id, atb [,mode]
next_id	UIS$GET_NEXT_OBJECT[1]	$\left\{ \begin{array}{l} \text{obj\_id} \\ \text{seg\_id} \end{array} \right\}$ [,flags]
type	UIS$GET_OBJECT_ATTRIBUTES[1]	$\left\{ \begin{array}{l} \text{obj\_id} \\ \text{seg\_id} \end{array} \right\}$ [,extent]
parent_id	UIS$GET_PARENT_SEGMENT[1]	$\left\{ \begin{array}{l} \text{obj\_id} \\ \text{seg\_id} \end{array} \right\}$
status	UIS$GET_POINTER_POSITION	vd_id, wd_id, retx, rety
	UIS$GET_POSITION	vd_id, retx, rety
prev_id	UIS$GET_PREVIOUS_OBJECT[1]	$\left\{ \begin{array}{l} \text{obj\_id} \\ \text{seg\_id} \end{array} \right\}$ [,flags]
root_id	UIS$GET_ROOT_SEGMENT	vd_id
	UIS$GET_TB_INFO	devnam, retwidth, retheight,retresolx, retresoly [,retpwidth, retpheight]
	UIS$GET_TB_POSITION	wd_id, retx, rety
formatting	UIS$GET_TEXT_FORMATTING	vd_id, atb
	UIS$GET_TEXT_MARGINS	vd_id, atb, x, y [,margin_length]

[1]VAX PASCAL and VAX PL/I applications must specify the **obj_id** argument.

Table A-1 (Cont.) UIS Calling Sequences

Return Value	Routine	Argument List
	UIS$GET_TEXT_PATH	vd_id, atb [,major][,minor]
angle	UIS$GET_TEXT_SLOPE	vd_id, atb
vcm_id	UIS$GET_VCM_ID	vd_id
boolean	UIS$GET_VIEWPORT_ICON	wd_id [icon_wd_id]
	UIS$GET_VIEWPORT_ POSITION	wd_id, retx, rety
	UIS$GET_VIEWPORT_SIZE	wd_id, retwidth, retheight
status	UIS$GET_VISIBILITY	vd_id, wd_id [,x_1, y_1 [,x_2, y_2]]
attributes	UIS$GET_WINDOW_ ATTRIBUTES	wd_id
	UIS$GET_WINDOW_SIZE	vd_id, wd_id, x_1, y_1, x_2, y_2
index	UIS$GET_WRITING_INDEX	vd_id, atb
mode	UIS$GET_WRITING_MODE	vd_id, atb
	UIS$GET_WS_COLOR	vd_id, color_id, retr, retg, retb [,wd_id]
	UIS$GET_WS_INTENSITY	vd_id, color_id, reti [,wd_id]
	UIS$HLS_TO_RGB	H, L, S, retr, retg, retb
	UIS$HSV_TO_RGB	H, S, V, retr, retg, retb
	UIS$IMAGE	vd_id, atb, x_1, y_1, x_2, y_2, rasterwidth, rasterheight, bitsperpixel, rasteraddr
	UIS$INSERT_OBJECT[1]	$\left\{ \begin{array}{l} \text{obj\_id} \\ \text{seg\_id} \end{array} \right\}$
	UIS$LINE	vd_id, atb, x_1, y_1 [,x_2, y_2 [,...x_n, y_n]]
	UIS$LINE_ARRAY	vd_id, atb, count, x_vector, y_vector
	UIS$MEASURE_TEXT	vd_id, atb, text_string, retwidth, retheight ,[ctllist, ctllen] [,posarray]
	UIS$MOVE_AREA	vd_id, x_1, y_1, x_2, y_2, new_x, new_y
	UIS$MOVE_VIEWPORT	wd_id, attributes
	UIS$MOVE_WINDOW	vd_id, wd_id, x_1, y_1, x_2, y_2
	UIS$NEW_TEXT_LINE	vd_id, atb
	UIS$PLOT	vd_id, atb, x_1, y_1 [,x_2, y_2 [,...x_n, y_n]]
	UIS$PLOT_ARRAY	vd_id, atb, count, x_vector, y_vector
	UIS$POP_VIEWPORT	wd_id
status	UIS$PRESENT	[major_version], [minor_version]
	UIS$PRIVATE[1]	$\left\{ \begin{array}{l} \text{obj\_id} \\ \text{vd\_id} \end{array} \right\}$, facnum, buffer
	UIS$PUSH_VIEWPORT	wd_id
keybuf	UIS$READ_CHAR	kb_id [,flags]

[1]VAX PASCAL and VAX PL/I applications must specify the **obj_id** argument.

Table A–1 (Cont.) UIS Calling Sequences

Return Value	Routine	Argument List
	UIS$RESIZE_WINDOW	vd_id, wd_id [,new_abs_x, new_abs_y] [,new_width new_height] [,new_wc_x_1, new_wc_y_1, new_wc_x_2, new_wc_y_2]
	UIS$RESTORE_CMS_ COLORS	cms_id
	UIS$RGB_TO_HLS	R, G, B, reth, retl, rets
	UIS$RGB_TO_HSV	R, G, B, reth, rets, retv
	UIS$SET_ADDOPT_AST	vd_id [,astadr [,astprm]]
	UIS$SET_ALIGNED_ POSITION	vd_id, atb, x, y
	UIS$SET_ARC_TYPE	vd_id, iatb, oatb, arc_type
	UIS$SET_BACKGROUND_ INDEX	vd_id, iatb, oatb, index
	UIS$SET_BUTTON_AST	vd_id, wd_id [,astadr [,astprm] ,keybuf] [,x_1, y_1, x_2, y_2]
	UIS$SET_CHAR_ROTATION	vd_id, iatb, oatb, angle
	UIS$SET_CHAR_SIZE	vd_id, iatb, oatb [,char] [,width][,height]
	UIS$SET_CHAR_SLANT	vd_id, iatb, oatb, angle
	UIS$SET_CHAR_SPACING	vd_id, iatb, oatb, dx, dy
	UIS$SET_CLIP	vd_id, iatb, oatb [,x_1, y_1, x_2, y_2]
	UIS$SET_CLOSE_AST	wd_id [,astadr [,astprm]]
	UIS$SET_COLOR	vd_id, index, R, G, B
	UIS$SET_COLORS	vd_id, index, count, r_vector, g_vector, b_vector
	UIS$SET_EXPAND_ICON_ AST	wd_id [,astadr [,astprm]]
	UIS$SET_FILL_PATTERN	vd_id, iatb, oatb [,index]
	UIS$SET_FONT	vd_id, iatb, oatb, font_id
	UIS$SET_GAIN_KB_AST	kb_id [,astadr [,astprm]]
	UIS$SET_INSERTION_ POSITION[1]	$\left\{\begin{array}{l}\text{obj\_id}\\\text{seg\_id}\\\text{vd\_id}\end{array}\right\}$ [,flags]
	UIS$SET_INTENSITIES	vd_id, index, count, i_vector
	UIS$SET_INTENSITY	vd_id, index, I
	UIS$SET_KB_AST	kb_id [,astadr [,astprm], keybuf]
	UIS$SET_KB_ATTRIBUTES	kb_id [,enable_items] [,disable_items] [click_volume]
	UIS$SET_KB_COMPOSE2	kb_id [,table, tablelen]
	UIS$SET_KB_COMPOSE3	kb_id [,table, tablelen]
	UIS$SET_KB_KEYTABLE	kb_id [,table, tablelen]
	UIS$SET_LINE_STYLE	vd_id, iatb, oatb, style

[1]VAX PASCAL and VAX PL/I applications must specify the **obj_id** argument.

Table A–1 (Cont.) UIS Calling Sequences

Return Value	Routine	Argument List
	UIS$SET_LINE_WIDTH	vd_id, itab, oatb, width [,mode]
	UIS$SET_LOSE_KB_AST	kb_id [,astadr [,astprm]]
	UIS$SET_MOVE_INFO_AST	wd_id [,astadr [,astprm]]
	UIS$SET_POINTER_AST	vd_id, wd_id [,astadr [,astprm]] [,x_1, y_1, x_2, y_2] [exitastadr [,exitastprm]]
	UIS$SET_POINTER_ PATTERN	vd_id, wd_id [,pattern_array, pattern_count, activex, activey] [,x_1, y_1, x_2, y_2] [,flags]
status	UIS$SET_POINTER_ POSITION	vd_id, wd_id, x, y
	UIS$SET_POSITION	vd_id, x, y
	UIS$SET_RESIZE_AST	vd_id, wd_id [,astadr [,astprm]] [,new_abs_x, new_abs_y] [,new_width, new_height] [,new_wc_x_1, new_wc_y_1, new_wc_ x_2, new_wc_y_2]
	UIS$SET_SHRINK_TO_ ICON_AST	wd_id [,astadr [,astprm]]
	UIS$SET_TB_AST	tb_id, [,data_astadr, [data_astprm]], [,x_pos,y_pos] [,data_ x_1, data_y_1, data_x_2, data_y_2] [,button_astadr [,button_ astprm],button_keybuf]
	UIS$SET_TEXT_ FORMATTING	vd_id, iatb, oatb,mode
	UIS$SET_TEXT_MARGINS	vd_id, iatb, oatb, x, y, margin_length
	UIS$SET_TEXT_PATH	vd_id, iatb, oatb, major[,minor]
	UIS$SET_TEXT_SLOPE	vd_id, iatb, oatb, angle
	UIS$SET_WRITING_INDEX	vd_id, iatb, oatb, index
	UIS$SET_WRITING_MODE	vd_id, iatb, oatb, mode
	UIS$SHRINK_TO_ICON	wd_id [,icon_wd_id] [,icon_flags] [,icon_name] [,attributes]
	UIS$SOUND_BELL	devnam [,bell_volume]
	UIS$SOUND_CLICK	devnam [,click_volume]
status	UIS$TEST_KB	kb_id
	UIS$TEXT	vd_id, atb, text_string [,x, y], [ctllist, ctllen]
	UIS$TRANSFORM_ OBJECT[1]	$\left\{ \begin{array}{l} \text{obj\_id} \\ \text{seg\_id} \end{array} \right\}$ [,matrix] [,atb]

[1]VAX PASCAL and VAX PL/I applications must specify the **obj_id** argument.

B UISDC CALLING SEQUENCES SUMMARY

Table B-1 summarizes UISDC calling sequences.

Table B-1 UISDC Calling Sequences

Return Value	Routine	Argument List
dop	UISDC$ALLOCATE_DOP	wd_id, size, atb
	UISDC$CIRCLE	wd_id, atb, center_x, center_y, xradius [,start_deg] [,end_deg]
	UISDC$ELLIPSE	wd_id, atb, center_x, center_y, xradius, yradius, [,start_deg] [,end_deg]
	UISDC$ERASE	wd_id [,x_1,y_1,x_2, y_2]
	UISDC$EXECUTE_DOP_ASYNCH	wd_id, dop, iosb
	UISDC$EXECUTE_DOP_SYNCH	wd_id, dop
	UISDC$GET_ALIGNED_POSITION	wd_id, atb, retx, rety
boolean	UISDC$GET_CHAR_SIZE	wd_id, atb [,char],[width][,height]
status	UISDC$GET_CLIP	wd_id, atb [,x_1,y_1, x_2,y_2]
status	UISDC$GET_POINTER_POSITION	wd_id, retx, rety
	UISDC$GET_POSITION	wd_id, retx, rety
	UISDC$GET_TEXT_MARGINS	wd_id, atb, x, y [,margin_length]
status	UISDC$GET_VISIBILITY	wd_id [,x_1,y_1 [,x_2,y_2]]
	UISDC$IMAGE	wd_id, atb, x_1, y_1, x_2, y_2, rasterwidth, rasterheight, bitsperpixel, rasteraddr
	UISDC$LINE	wd_id, atb, x_1,y_1, [,x_2,y_2 [,...x_n, y_n]]
	UISDC$LINE_ARRAY	wd_id, atb, count, x_vector, y_vector
bitmap_id	UISDC$LOAD_BITMAP	wd_id, bitmap_adr, bitmap_len, bitmap_width, bits_per_pixel
	UISDC$MEASURE_TEXT	wd_id, atb, text_string, retwidth, retheight [,ctllist ,ctllen] [,posarray]

Table B-1 (Cont.) UISDC Calling Sequences

Return Value	Routine	Argument List
	UISDC$MOVE_AREA	wd_id, x_1,y_1,x_2, y_2, new_x, new_y
	UISDC$NEW_TEXT_LINE	wd_id, atb
	UISDC$PLOT	wd_id, atb, x_1,y_1, [,x_2,y_2 [,...x_n, y_n]]
	UISDC$PLOT_ARRAY	wd_id, atb, count, x_ vector, y_vector
	UISDC$QUEUE_DOP	wd_id, dop
	UISDC$READ_IMAGE	wd_id, x_1, y_1, x_2, y_2, rasterwidht, rasterheight, bitsperpixel, rasteraddr, rasterlen
	UISDC$SET_ALIGNED_POSITION	wd_id, atb, x, y
	UISDC$SET_BUTTON_AST	wd_id [,astadr, [astprm], keybuf] [,x_1, y_1, x_2, y_2]
	UISDC$SET_CHAR_SIZE	wd_id, iatb, oatb [,char][,width][,height]
	UISDC$SET_CLIP	wd_id, iatb, oatb [,x_1, y_1, x_2, y_2]
	UISDC$SET_POINTER_AST	wd_id [,astadr [astprm]] [,x_1, y_1, x_2, y_2] [,exitastadr [,exitastprm]]
	UISDC$SET_POINTER_PATTERN	wd_id [,pattern_array, pattern_count, activex, activey] [,x_1, y_1, x_2, y_2][,flags]
status	UISDC$SET_POINTER_POSITION	wd_id, x, y
	UISDC$SET_POSITION	wd_id, x, y
	UISDC$SET_TEXT_MARGINS	wd_id, iatb, oatb, x, y, margin_length
	UISDC$TEXT	wd_id atb, text_string [,x, y] [,ctllist, ctllen]

C UIS Multinational Character and Technical Fonts

C.1 Overview

This appendix contains figures and tables that illustrate the UIS multinational character and technical fonts and font names in the directory SYS$FONT.

C.2 UIS Multinational Character Set Fonts and Font Specifications

The SYS$FONT directory has 14 multinational character set font files. The following figure captions identify each UIS font with an arbitrarily assigned font number. Each figure caption has an accompanying table with additional typographical information. The tables analyze the first 16 characters of the font file name.

Figure C–1 Font 1

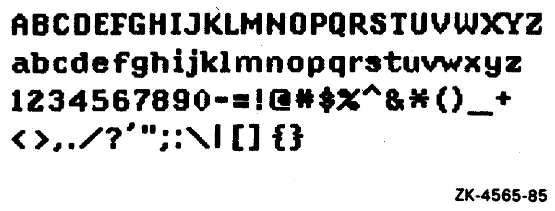
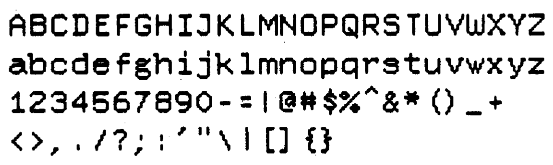


Table C–1 Font 1

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by DIGITAL
2-7	Type Family ID	TABER0	Taber
8	Spacing	0	Proportionally spaced
9-11	Type size	$03W_{36}$	14 points (140 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	P	Bold
16	Proportion	G	Regular

Figure C–2 Font 2



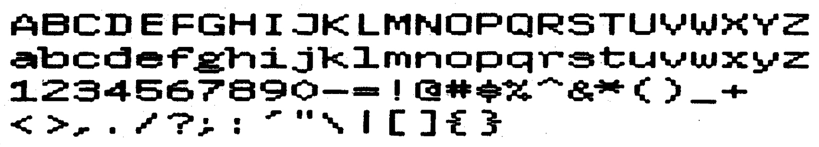
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
1234567890-=|@#$%^&*()_+
<>,./?;:'"\|[]{}

ZK-4566-85

Table C–2 Font 2

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by DIGITAL
2-7	Type Family ID	TABER0	Taber
8	Spacing	I	9 pitch (monospaced)
9-11	Type size	$03W_{36}$	14 points (140 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	G	Regular
16	Proportion	G	Regular

Figure C–3 Font 3

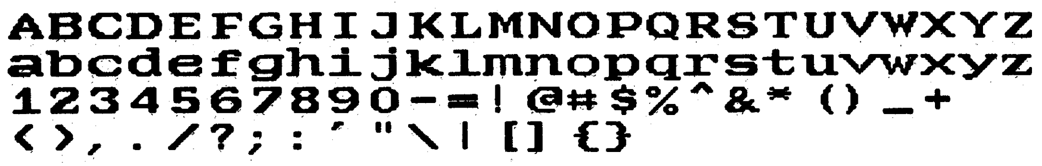


ZK-4567-85

Table C–3 Font 3

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by DIGITAL
2-7	Type Family ID	TABER0	Taber
8	Spacing	M	13 pitch (monospaced)
9-11	Type size	$03C_{36}$	12 points (120 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	G	Regular
16	Proportion	G	Regular

Figure C–4 Font 4

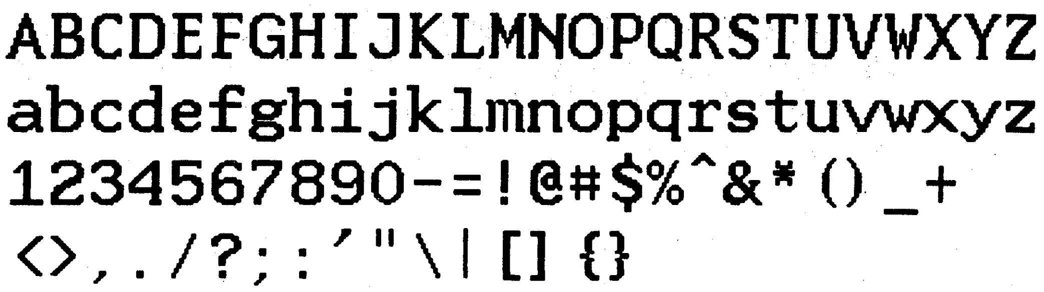


ZK-4568-85

Table C–4 Font 4

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by DIGITAL
2-7	Type Family ID	TABER0	Taber
8	Spacing	R	18 pitch (monospaced)
9-11	Type size	$03W_{36}$	14 points (140 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	G	Regular
16	Proportion	G	Regular

Figure C–5 Font 5

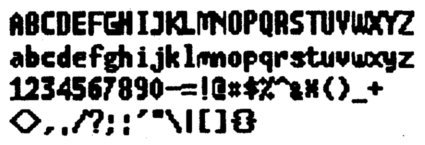


ZK-4569-85

Table C–5 Font 5

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by DIGITAL
2-7	Type Family ID	TABER0	Taber
8	Spacing	R	18 pitch (monospaced)
9-11	Type size	07S$_{36}$	28 points (280 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00$_{36}$	Roman
15	Weight	G	Regular
16	Proportion	G	Regular

Figure C–6 Font 6



ZK-4570-85

Table C–6 Font 6

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by DIGITAL
2-7	Type Family ID	TERMIN	Terminal
8	Spacing	G	7 pitch (monospaced)
9-11	Type size	$03C_{36}$	12 points (120 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	P	Bold
16	Proportion	G	Regular

Figure C-7 Font 7

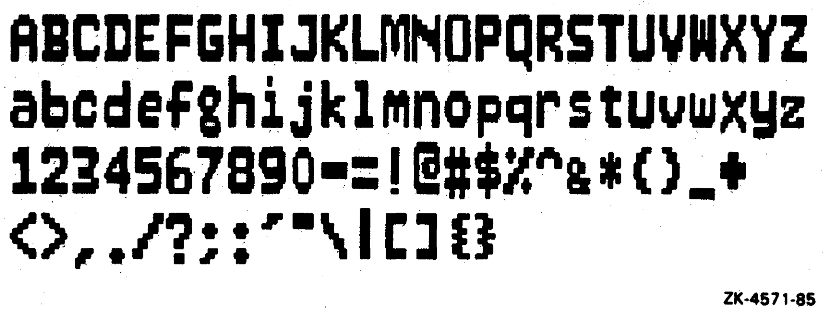


Table C-7 Font 7

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by DIGITAL
2-7	Type Family ID	TERMIN	Terminal
8	Spacing	M	13 pitch (monospaced)
9-11	Type size	$06O_{36}$	24 points (240 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	P	Bold
16	Proportion	G	Regular

Figure C–8 Font 8

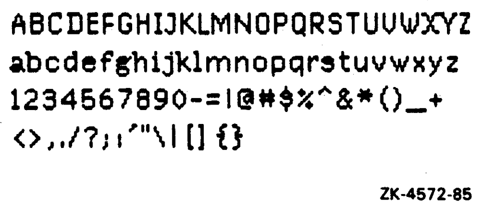
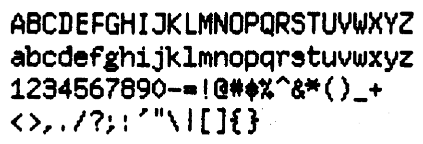


Table C–8 Font 8

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by DIGITAL
2-7	Type Family ID	TABER0	Taber
8	Spacing	0	proportionally spaced
9-11	Type size	$03W_{36}$	14 points (140 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	G	Regular
16	Proportion	G	Regular

Figure C–9 Font 9



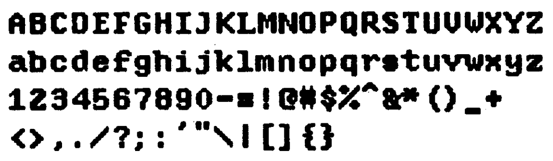
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
1234567890-=!@#$%^&*()_+
<>,./?;:'"\|[]{}

ZK-4573-85

Table C–9 Font 9

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by DIGITAL
2-7	Type Family ID	TABER0	Taber
8	Spacing	G	7 pitch (monospaced)
9-11	Type size	$03C_{36}$	12 points (120 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	G	Regular
16	Proportion	G	Regular

Figure C–10 Font 10

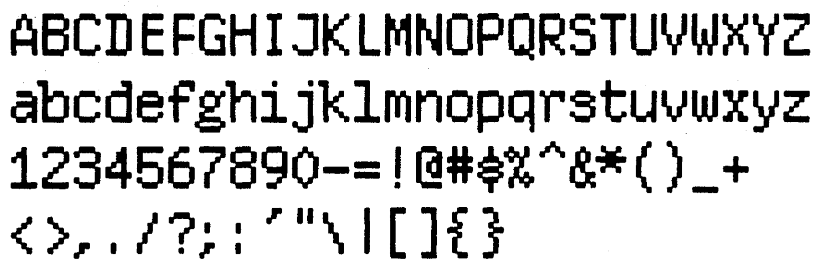


ZK-4574-85

Table C–10 Font 10

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by DIGITAL
2-7	Type Family ID	TABER0	Taber
8	Spacing	I	9 (monospaced)
9-11	Type size	$03W_{36}$	14 points (140 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	P	Bold
16	Proportion	G	Regular

Figure C–11 Font 11

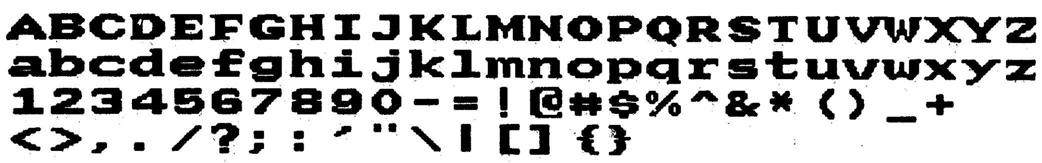


ZK-4575-85

Table C–11 Font 11

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by DIGITAL
2-7	Type Family ID	TABER0	Taber
8	Spacing	M	13 pitch (monospaced)
9-11	Type size	$06O_{36}$	24 points (240 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	G	Regular
16	Proportion	G	Regular

Figure C–12 Font 12

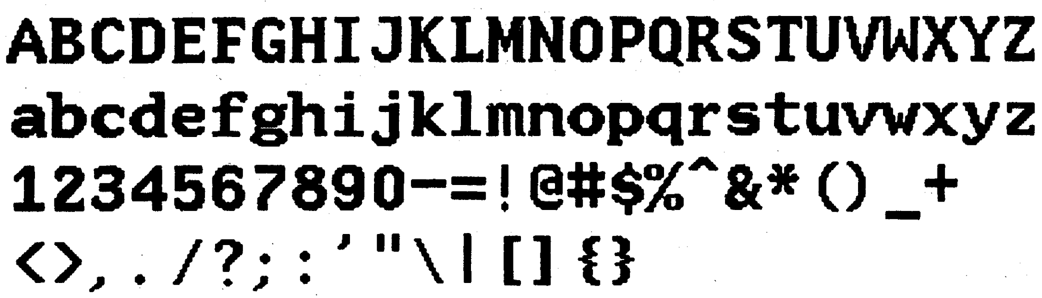


ZK-4576-85

Table C–12 Font 12

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by DIGITAL
2-7	Type Family ID	TABER0	Taber
8	Spacing	R	18 pitch (monospaced)
9-11	Type size	$03W_{36}$	14 points (140 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	P	Bold
16	Proportion	G	Regular

Figure C–13 Font 13

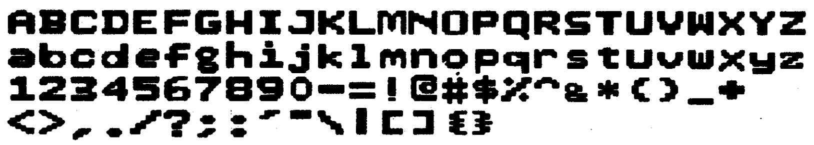


ZK-4577-85

Table C–13 Font 13

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by DIGITAL
2-7	Type Family ID	TABER0	Taber
8	Spacing	R	18 pitch (monospaced)
9-11	Type size	$07S_{36}$	28 points (280 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	P	Bold
16	Proportion	G	Regular

Figure C-14 Font 14



ZK-4578-85

Table C-14 Font 14

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by DIGITAL
2-7	Type Family ID	TERMIN	Terminal
8	Spacing	M	13 pitch (monospaced)
9-11	Type size	$03C_{36}$	12 points (120 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	P	Bold
16	Proportion	G	Regular

C.3 UIS Technical Character Set Fonts

The SYS$FONT directory contains 12 technical character set font files. The following figure captions identify each UIS font with an arbitrarily assigned font number. Each figure caption has an accompanying table with additional typographical information. The tables analyze the first 16 characters of the font file name.

Figure C–15 Font 15

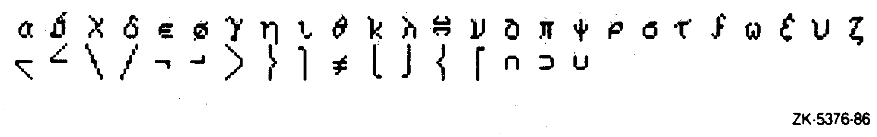


Table C–15 Font 15

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by Digital
2-7	Type Family ID	VWSVT0	VAXstation Technical Character Set
8	Spacing	G	7 pitch (monospaced)
9-11	Type size	$03C_{36}$	12 points (120 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	G	Regular
16	Proportion	G	Regular

Figure C–16 Font 16

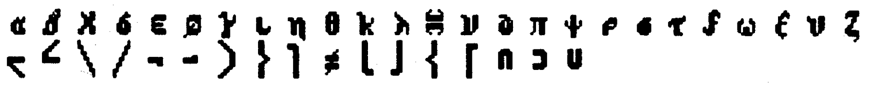

ZK-5375-86

Table C–16 Font 16

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by Digital
2-7	Type Family ID	VWSVT0	VAXstation Technical Character Set
8	Spacing	G	7 pitch (monospaced)
9-11	Type size	$03C_{36}$	12 points (120 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	P	Bold
16	Proportion	G	Regular

Figure C–17 Font 17

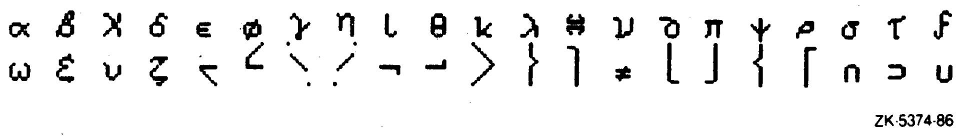


Table C–17 Font 17

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by Digital
2-7	Type Family ID	VWSVT0	VAXstation Technical Character Set
8	Spacing	I	9 pitch (monospaced)
9-11	Type size	$03W_{36}$	14 points (140 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	G	Regular
16	Proportion	G	Regular

Figure C–18 Font 18

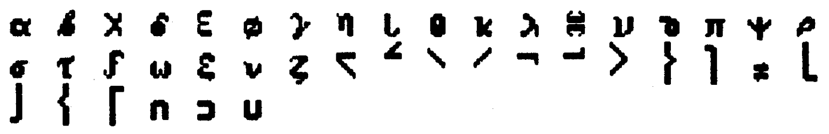


Table C–18 Font 18

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by Digital
2-7	Type Family ID	VWSVT0	VAXstation Technical Character Set
8	Spacing	I	9 pitch (monospaced)
9-11	Type size	$03W_{36}$	14 points (140 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	P	Bold
16	Proportion	G	Regular

Figure C–19 Font 19

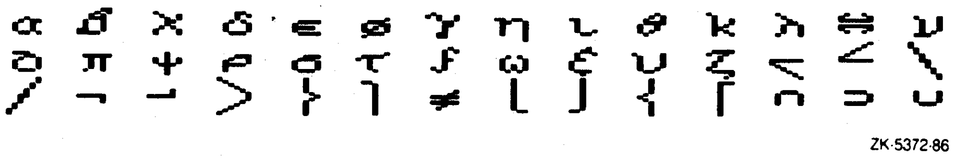


Table C–19 Font 19

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by Digital
2-7	Type Family ID	VWSVT0	VAXstation Technical Character Set
8	Spacing	N	14 pitch (monospaced)
9-11	Type size	$03C_{36}$	120 points (120 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	G	Regular
16	Proportion	G	Regular

Figure C–20 Font 20

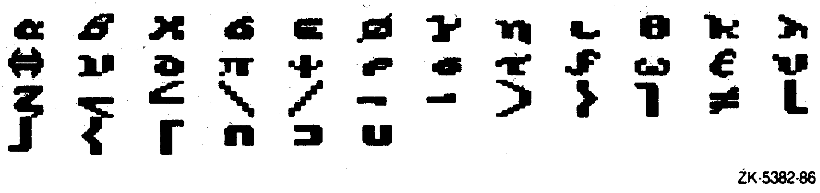


Table C–20 Font 20

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by Digital
2-7	Type Family ID	VWSVT0	VAXstation Technical Character Set
8	Spacing	N	14 pitch (monospaced)
9-11	Type size	$03C_{36}$	12 points (120 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	P	Bold
16	Proportion	G	Regular

Figure C–21 Font 21

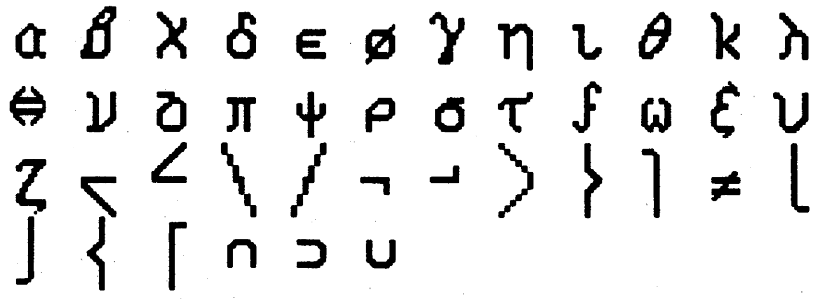


Table C–21 Font 21

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by Digital
2-7	Type Family ID	VWSVT0	VAXstation Technical Character Set
8	Spacing	N	14 pitch (monospaced)
9-11	Type size	$06O_{36}$	24 points (240 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	G	Regular
16	Proportion	G	Regular

Figure C–22 Font 22

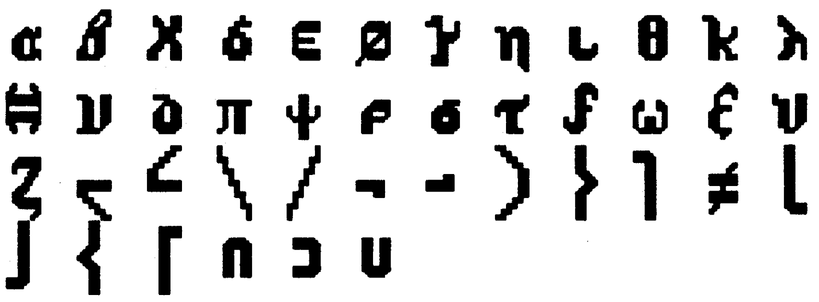


Table C–22 Font 22

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by Digital
2-7	Type Family ID	VWSVT0	VAXstation Technical Character Set
8	Spacing	N	14 pitch (monospaced)
9-11	Type size	$06O_{36}$	24 points (240 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	P	Bold
16	Proportion	G	Regular

Figure C–23 Font 23

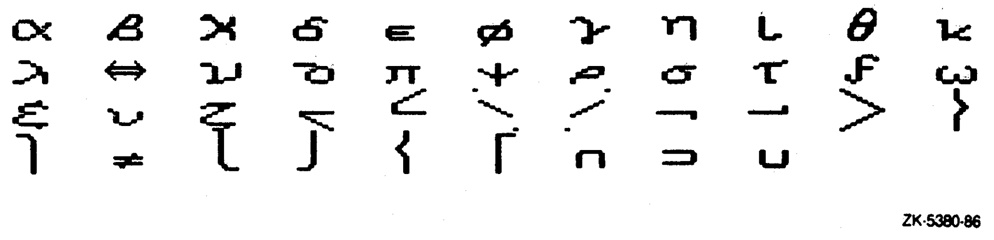


Table C–23 Font 23

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by Digital
2-7	Type Family ID	VWSVT0	VAXstation Technical Character Set
8	Spacing	R	18 pitch (monospaced)
9-11	Type size	$03W_{36}$	14 points (140 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	G	Regular
16	Proportion	G	Regular

Figure C–24 Font 24

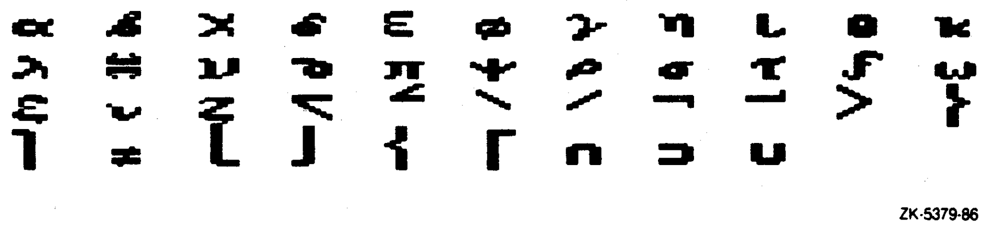


Table C–24 Font 24

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by Digital
2-7	Type Family ID	VWSVT0	VAXstation Technical Character Set
8	Spacing	R	18 pitch (monospaced)
9-11	Type size	$03W_{36}$	14 points (140 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	P	Bold
16	Proportion	G	Regular

Figure C–25 Font 25

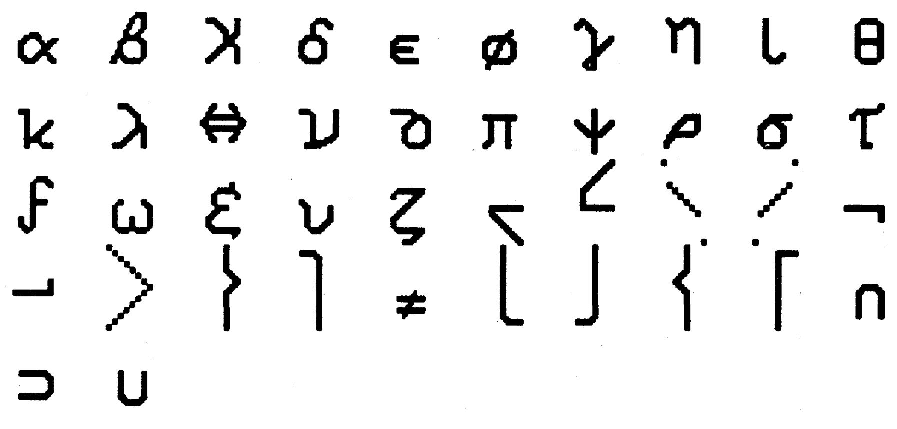


Table C–25 Font 25

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by Digital
2-7	Type Family ID	VWSVT0	VAXstation Technical Character Set
8	Spacing	R	18 pitch (monospaced)
9-11	Type size	07S$_{36}$	28 points (280 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00$_{36}$	Roman
15	Weight	G	Regular
16	Proportion	G	Regular

Figure C–26 Font 26

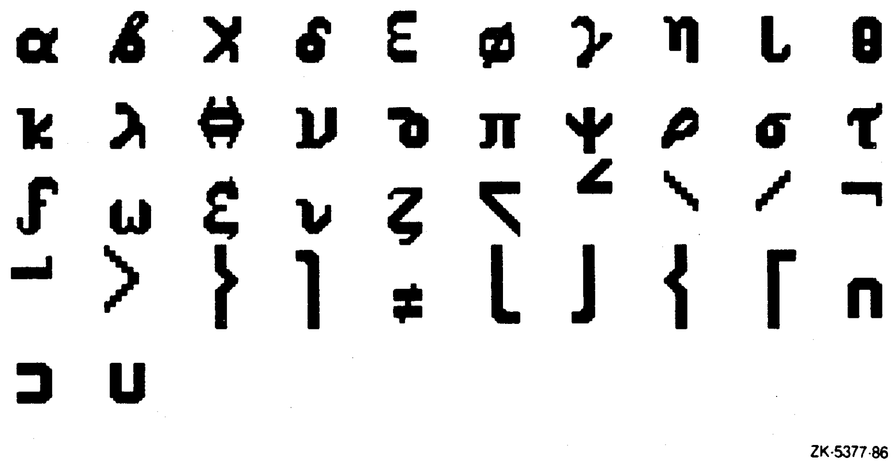


Table C–26 Font 26

Field	Field Name	Value	Meaning
1	Registration code	D	Registered by Digital
2-7	Type Family ID	VWSVT0	VAXstation Technical Character Set
8	Spacing	R	18 pitch (monospaced)
9-11	Type size	$07S_{36}$	28 points (280 decipoints)
12	Scale factor	K	1 (normal)
13-14	Style	00_{36}	Roman
15	Weight	G	Regular
16	Proportion	G	Regular

D FILL PATTERNS

All fill patterns are located in directory SYS$FONT, in a separate file named DEUISPATAAAAAAF000000000DA.VWS$FONT. Access this file with the logical name UIS$FILL_PATTERNS.

The pairs of fill patterns shown in the following figures are drawn in overlay writing mode on a white background. The figure caption contains the symbol name for each fill pattern. The symbol name represents an index to the appropriate fill pattern.

Symbol names are located in language-specific symbol definition files in SYS$LIBRARY. 6.2 lists symbol definition files.

Figure D–1 PATT$C_VERT1_1 and PATT$C_VERT1_3

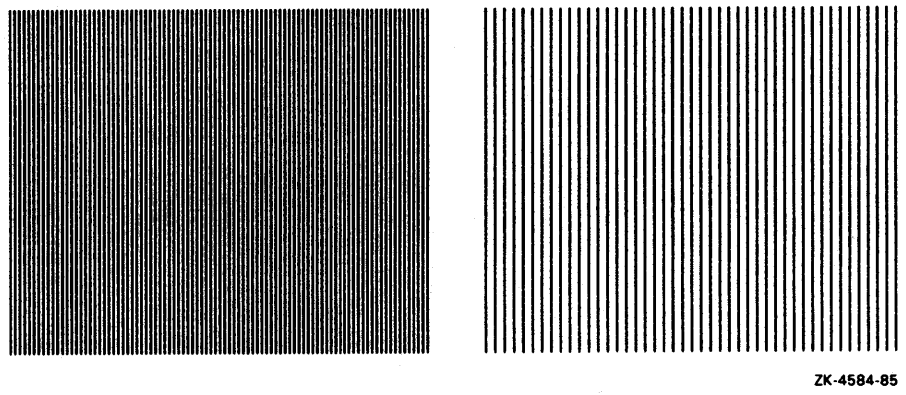


Figure D–2 PATT$C_VERT2_2 and PATT$C_VERT3_1

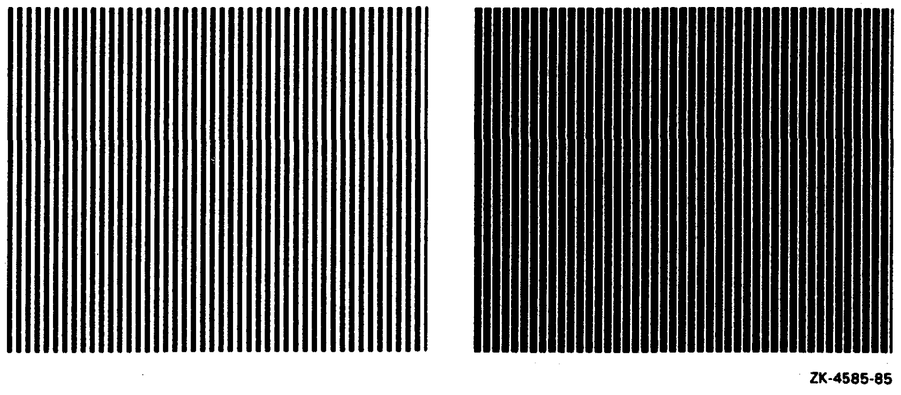


Figure D–3 PATT$C_VERT1_7 and PATT$C_VERT2_6

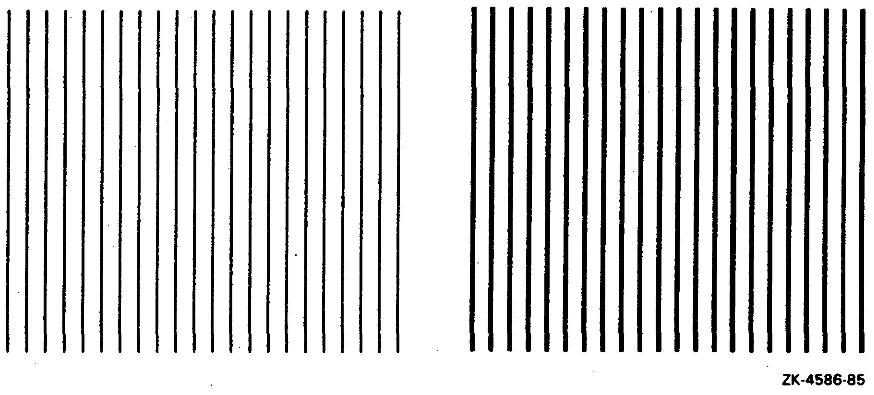


Figure D–4 PATTC_VERT_4 and PATT$C_VERT6_2

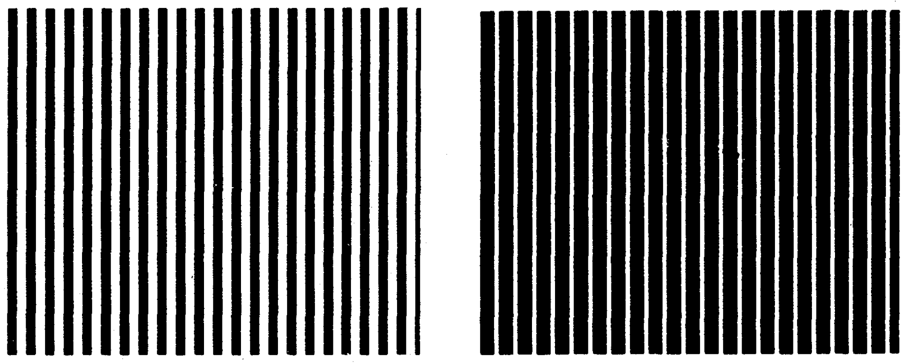


Figure D–5 PATT$C_HORIZ1_1 and PATT$C_HORIZ1_3

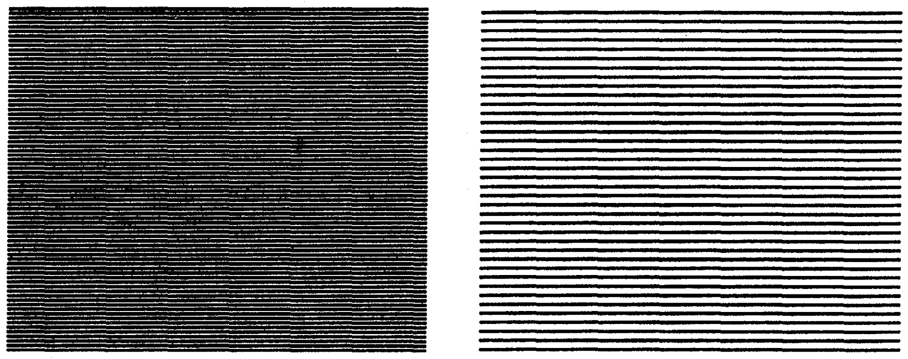


Figure D-6   PATT$C_HORIZ2_2 and PATT$C_HORIZ3_1

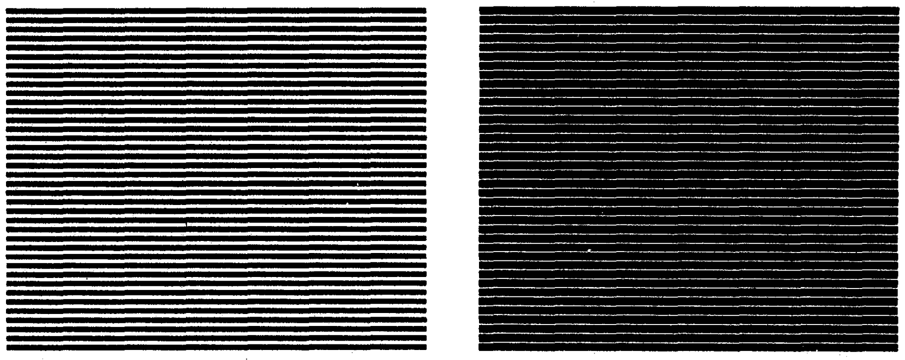


Figure D-7   PATT$C_HORIZ1_7 and PATT$C_HORIZ2_6

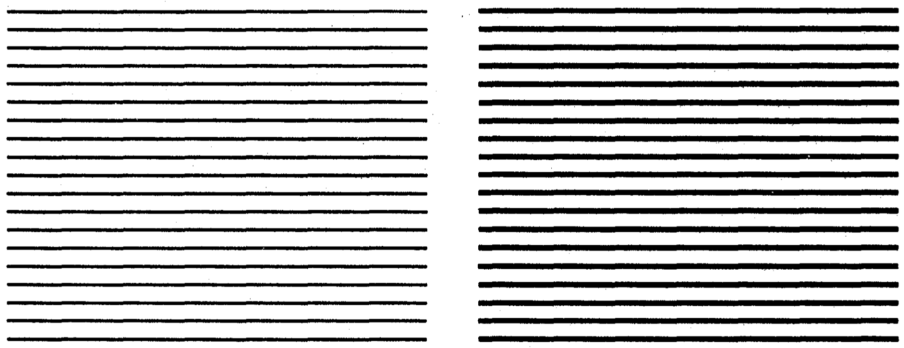


Figure D-8 PATT$C_HORIZ4_4 and PATT$C_HORIZ6_2

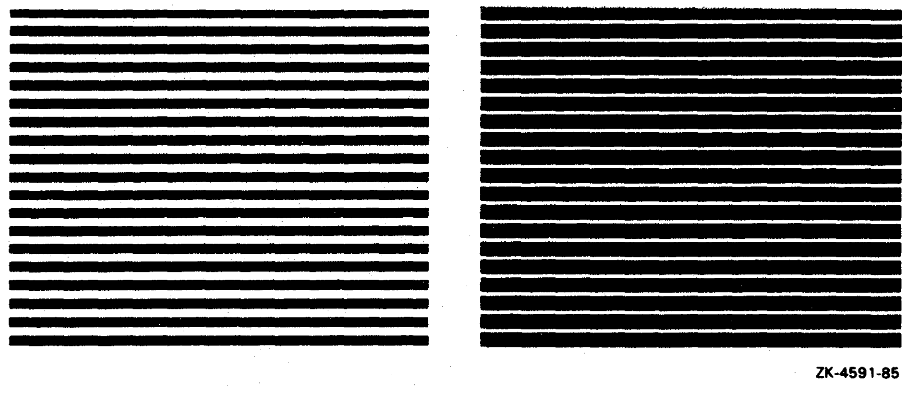


Figure D-9 PATT$C_GRID4 and PATT$C_GRID8

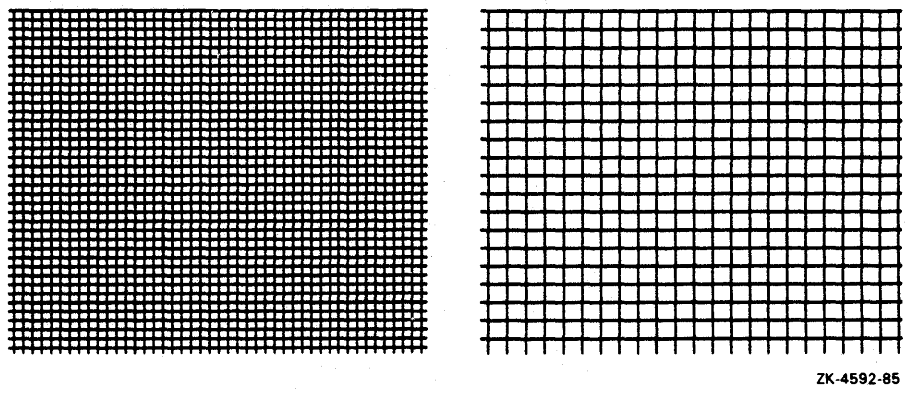


Figure D–10 PATT$C_UPDIAG1_3 and PATT$C_UPDIAG2_2

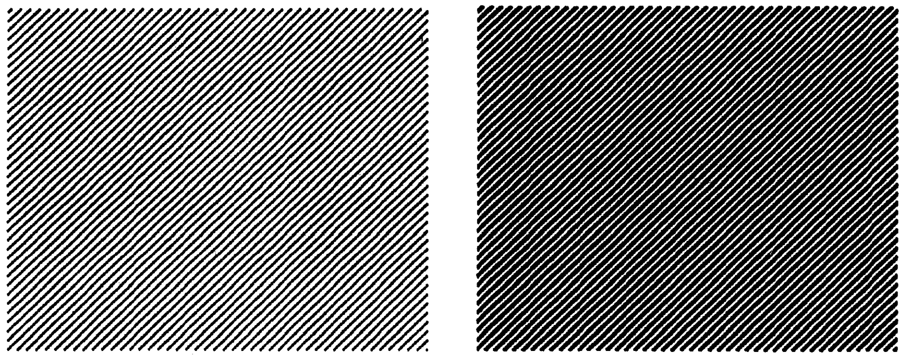


Figure D–11 PATT$C_UPDIAG3_1 and PATT$C_UPDIAG1_7

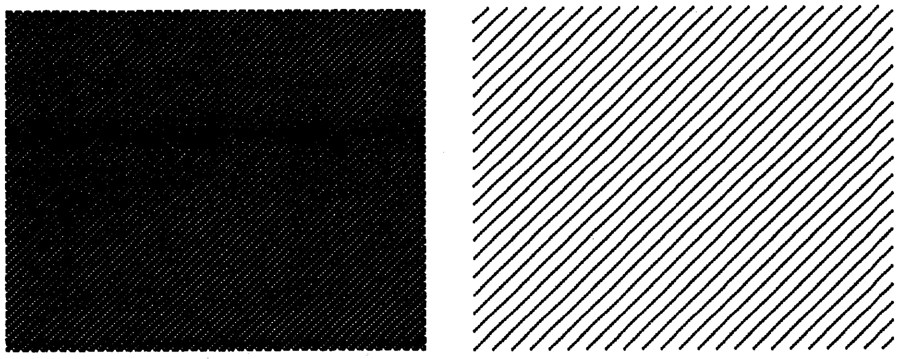


Figure D–12 PATT$C_UPDIAG2_6 and PATT$C_UPDIAG4_4

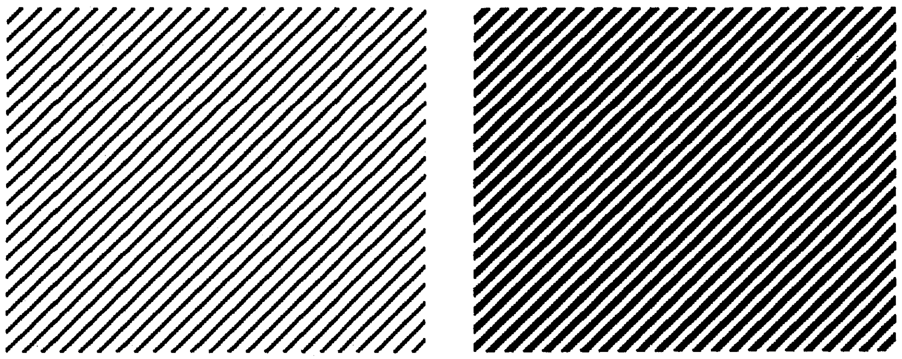


Figure D–13 PATT$C_UPDIAG6_2 and PATT$C_DOWNDIAG1_3

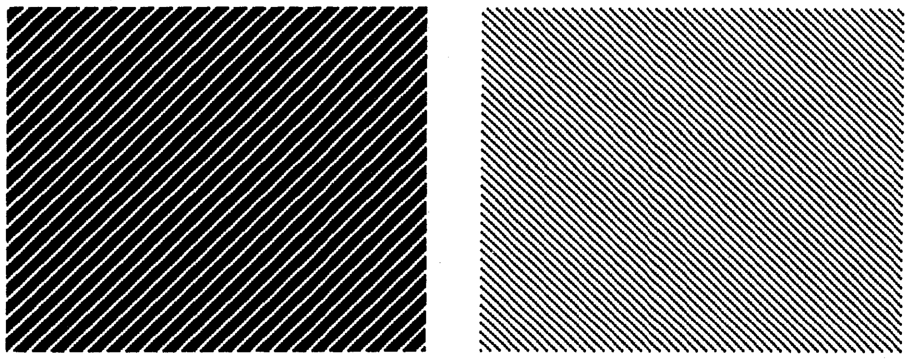
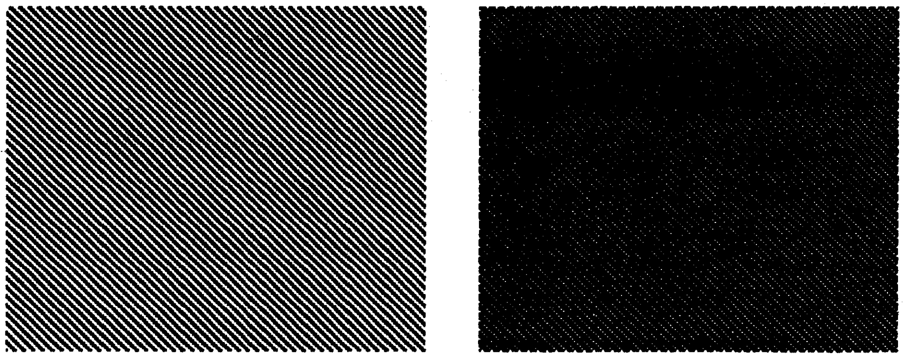
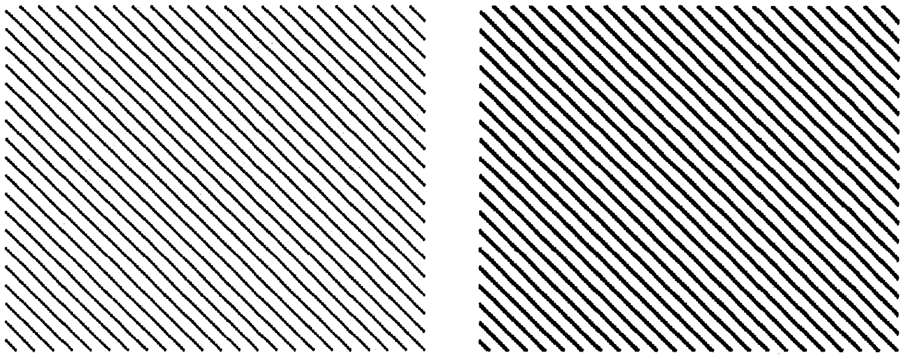


Figure D–14 PATT$C_DOWNDIAG2_2 and PATT$C_DOWNDIAG3_1



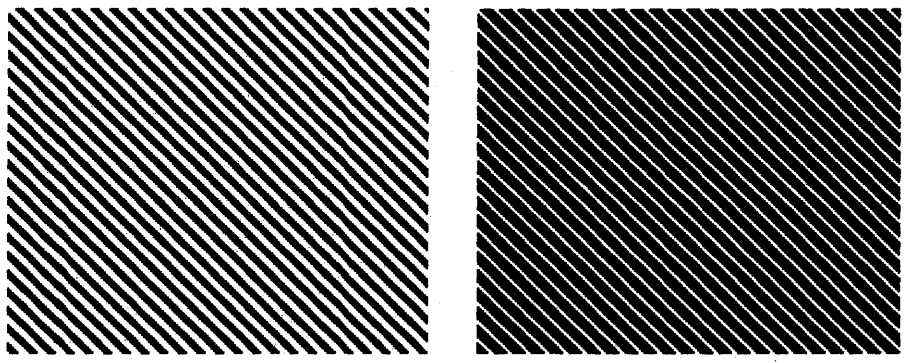
ZK-4597-85

Figure D–15 PATT$C_DOWNDIAG1_7 and PATT$C_DOWNDIAG2_6



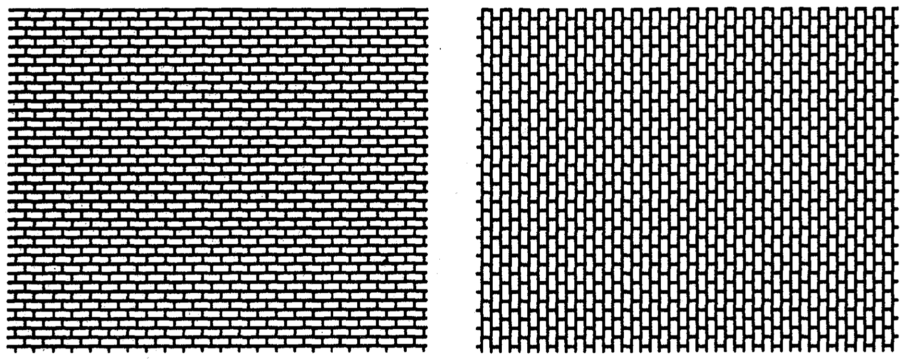
ZK-4598-85

Figure D–16 PATT$C_DOWNDIAG4_4 and PATT$C_DOWNDIAG6_2



ZK-4599-85

Figure D–17 PATT$C_BRICK_HORIZ and PATT$C_BRICK_VERT



ZK-4600-85

Figure D-18 PATT$C_BRICK_DOWNDIAG and PATT$C_BRICK_UPDIAG

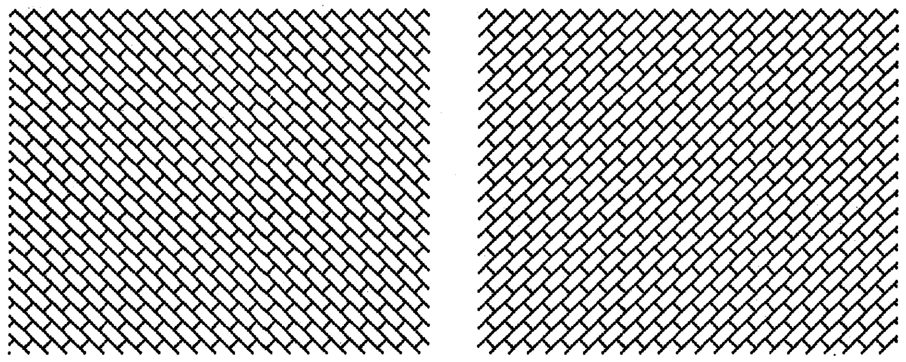


Figure D-19 PATT$C_GREY4_16D and PATT$C_GREY12_16D

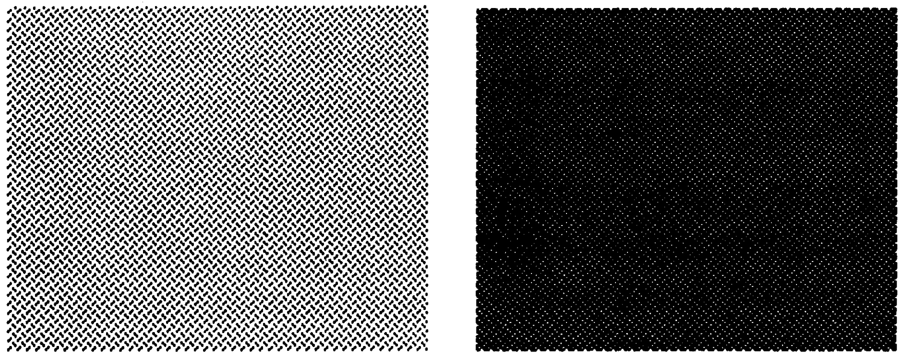


Figure D–20 PATT$C_BASKET_WEAVE and PATT$C_SCALE_DOWN

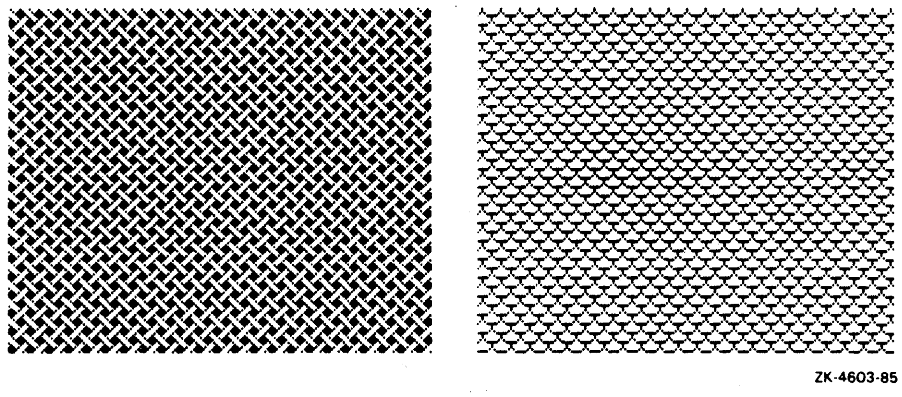


Figure D–21 PATT$C_SCALE_UP and PATT$C_SCALE_RIGHT

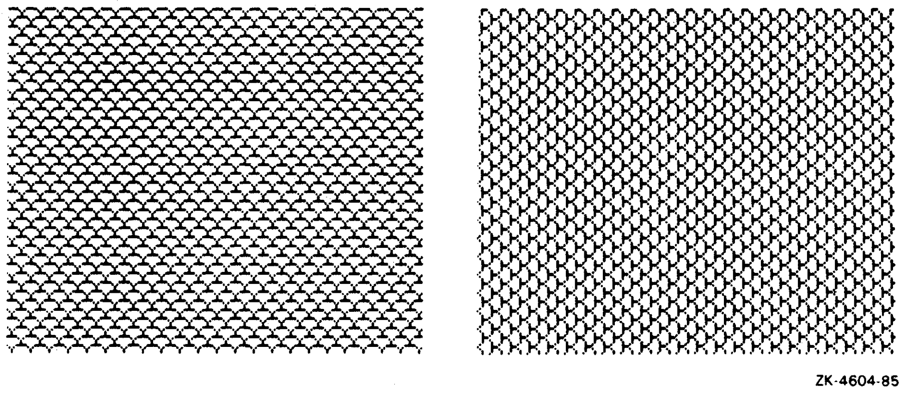


Figure D–22 PATT$C_SCALE_LEFT and PATT$C_GREY1_16

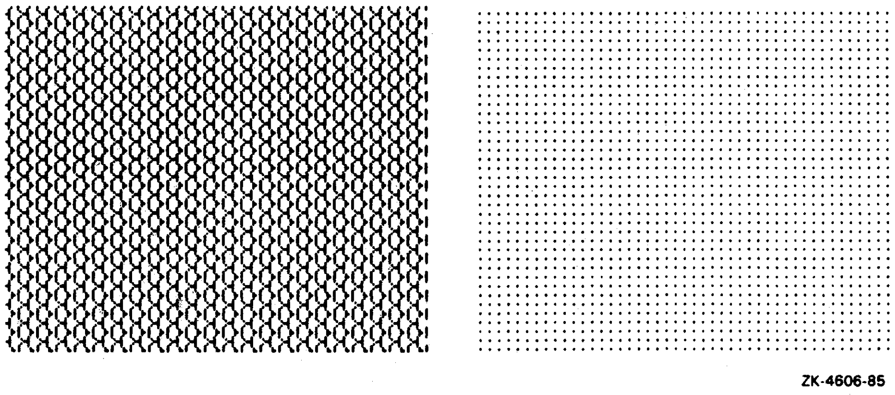


Figure D–23 PATT$C_GREY2_16 AND PATT$C_GREY3_16

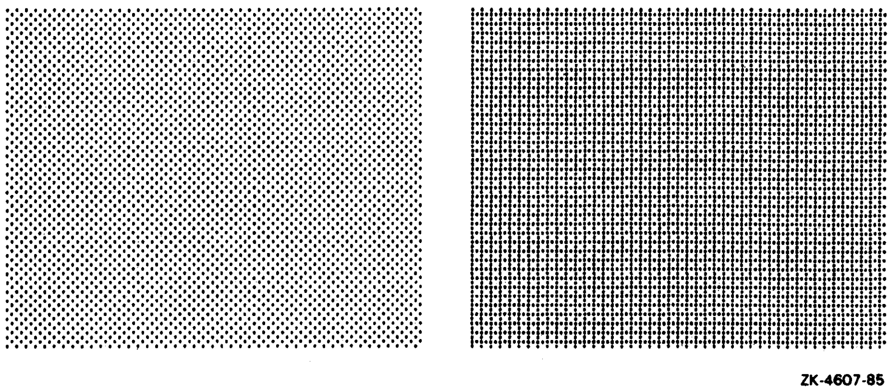


Figure D–24 PATT$C_GREY4_16 and PATT$C_GREY5_16

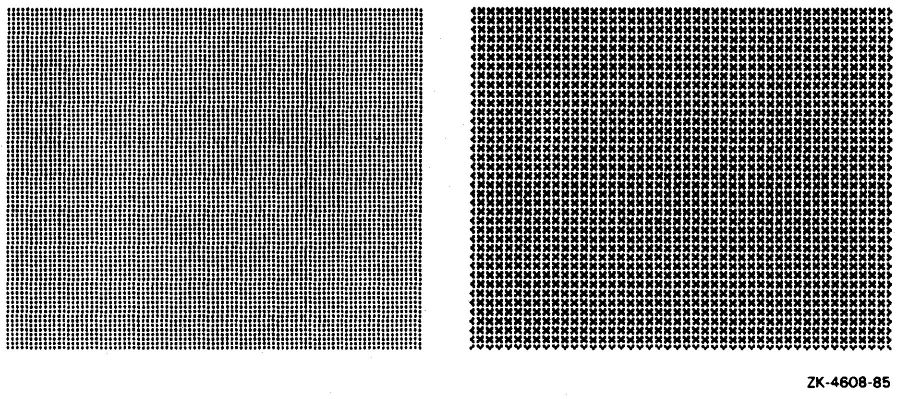

ZK-4608-85

Figure D–25 PATT$C_GREY6_16 and PATT$C_GREY7_16

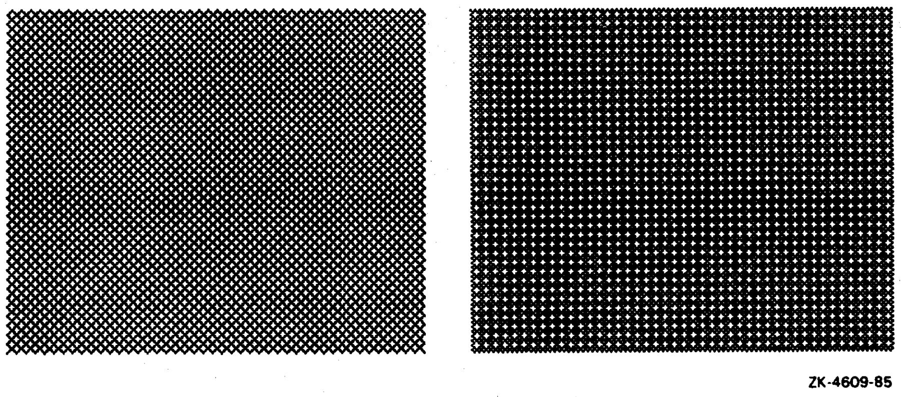

ZK-4609-85

Figure D–26 PATT$C_GREY8_16 and PATT$C_GREY9_16

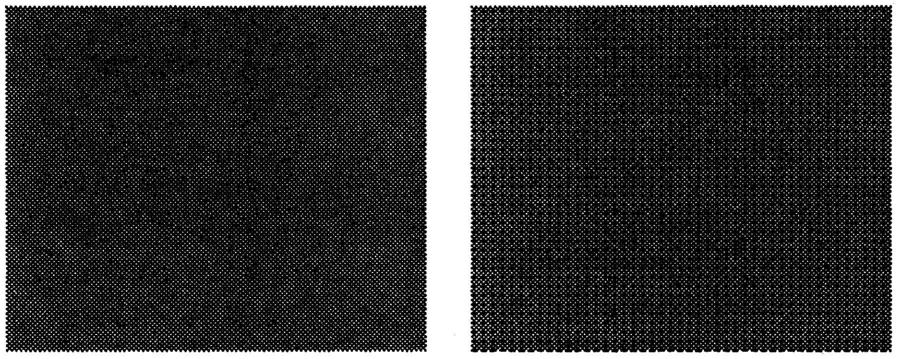


Figure D–27 PATT$C_GREY10_16 and PATT$C_GREY11_16

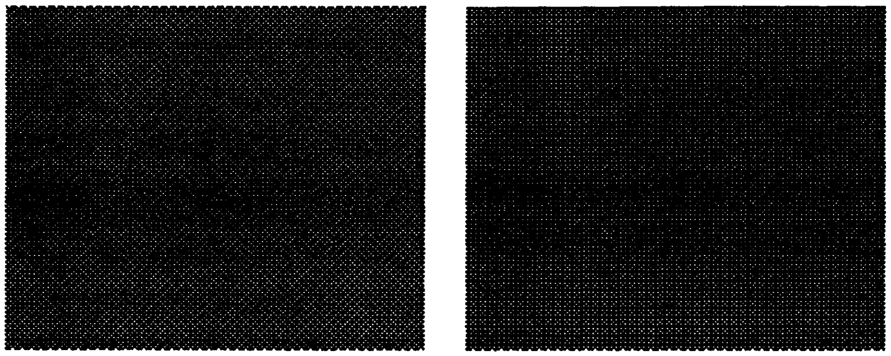
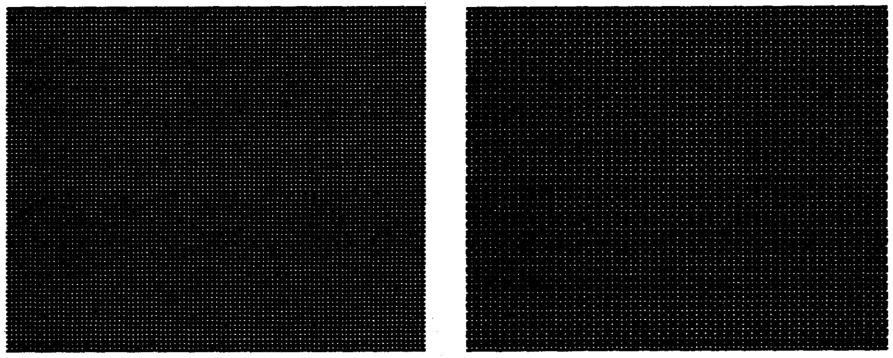
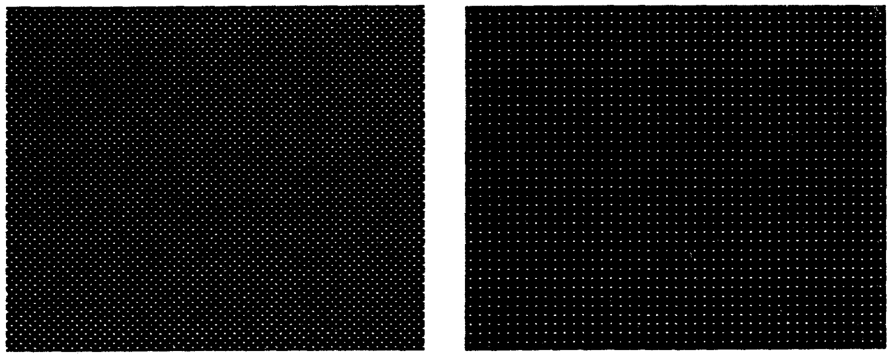


Figure D-28 PATT$C_GREY12_16 and PATT$C_GREY13_16



ZK-4612-85

Figure D-29 PATT$C_GREY14_16 and PATT$C_GREY15_16



ZK-4613-85

E ERROR MESSAGES

This appendix lists the error messages generated by MicroVMS workstation graphics software. Each message description consists of the message text, a brief explanation of the message, and the possible remedy.

BAD_ATB, Illegal attempt to modify attribute block 0 (read-only).

Explanation: An attempt is made to modify an attribute in attribute block #0, which is defined to be read-only. The modification request is ignored.

User Action: Check for a programming error.

BAD_COLOR_VALUE, Color value out of range.

Explanation: An attempt is made to specify one or more color values that are out of range.

User Action: Check for a programming error.

BAD_CMS, Illegal color map segment identifier.

Explanation: An illegal color map segment identifier is given to a UIS routine as an argument.

User Action: Check for a programming error.

BAD_DISP, Display list has been corrupted.

Explanation: An illegal display list type code is encountered while the program traverses a display list.

User Action: Check the validity of the UIS metafile you are executing.

BAD_DOP, Illegal drawing packet (DOP) format.

Explanation: An attempt is made to pass a drawing operation primitive that is 0, the type field is not UIS$C_DYN_DOP, or the size field is less than DOP$C_LENGTH.

User Action: Check for a programming error.

BAD_FONT, !AS is not a valid font.

Explanation: An attempt is made to reference an activated but invalid VWS font. The font is not placed in the specified attribute block. The program continues after this error.

User Action: Check for a programming error.

BAD_ICON_WD, Invalid icon window identifier.

Explanation: An attempt is made to specify the icon WD_ID that does not match the one saved by UIS. UIS saves the icon WD_ID when it shrinks a viewport to an icon or when UIS$ASSOCIATE_ICON_WITH_VP is called.

User Action: Check for a programming error.

BAD_KB, Illegal virtual keyboard identifier.

Explanation: An illegal virtual keyboard identifier is given to a UIS routine as an argument.

User Action: Check for a programming error.

BAD_OBJ_ID, Illegal object identifier.

Explanation: An illegal object (segment or primitive) identifier is given to a UIS routine as an argument.

User Action: Check for a programming error.

BAD_OPCODE, Unrecognized generic encoding item.

Explanation: The generic encoding interpreter detects an unknown item opcode. The rest of the generic encoding stream is skipped.

User Action: Check for a programming error.

BAD_STRING, String too long.

Explanation: An attempt is made to pass a string that is too long.

User Action: Shorten the string.

BAD_TB, Illegal tablet identifier.

Explanation: An illegal tablet identifier is given to a UIS routine as an argument.

User Action: Check for a programming error.

BAD_TEXT_ITEM, Unrecognized text control item, item=!XL.

Explanation: An illegal text control item is specified.

User Action: Check for a programming error.

BADTITLE, Illegal window title string.

Explanation: An illegal window title string is passed when a user attempts to create a window.

User Action: Check for a programming error.

BAD_TR, Illegal transformation identifier.

Explanation: An illegal transformation identifier is given to a UIS routine as an argument.

User Action: Check for a programming error.

BAD_VD, Illegal virtual display identifier.

Explanation: An illegal virtual display identifier is given to a UIS routine as an argument.

User Action: Check for a programming error.

BAD_VCM, Illegal virtual color map identifier.

Explanation: An illegal virtual color map index is given to a UIS routine as an argument. The color map index must be less than the VCM size.

User Action: Check for a programming error.

BAD_VCM_ATTR, Illegal or missing virtual color map attributes.

Explanation: One or more illegal virtual color map attributes are given to a UIS routine as an argument, or one or more attributes are missing.

User Action: Check for a programming error.

BAD_VCM_INDEX, Virtual color map index out of range.

Explanation: An illegal virtual color map index is given to a UIS routine as an argument.

User Action: Check for a programming error.

BAD_VCM_NAME, Illegal or missing virtual color map name.

Explanation: An illegal virtual color map name is given to a UIS routine as an argument, or the name is missing.

User Action: Check for a programming error.

BAD_VCM_SIZE, Virtual color map size out of range or illegal.

Explanation: An illegal virtual color map size is given to a UIS routine as an argument, or the process references an existing shareable virtual color map that specifies a different size than that of the existing map.

User Action: Check for a programming error.

BAD_VER, Bad display list version number.

Explanation: An attempt is made to pass an unsupported version of the display list to UIS$EXECUTE.

User Action: Use a supported display list.

BAD_VOLUME, Illegal volume level specified.

Explanation: An illegal volume level is given to the UIS$SOUND routine. The volume must be in the range of 1 to 8.

User Action: Check for a programming error.

BAD_WD, Illegal display window identifier.

Explanation: An illegal display window identifier is given to a UIS routine as an argument.

User Action: Check for a programming error.

BADWDPL, Window placement attribute list has an invalid format.

Explanation: An illegal window attribute list is passed when a user attempts to create a window.

User Action: Check for illegal item types in the window attribute list.

CMS_ACTIVE, Color map segment is still referenced by virtual color map(s).

Explanation: An attempt is made to delete a color map segment that is still referenced by one or more virtual color maps.

User Action: Check for a programming error.

CMS_CREATE_ERR, Requested color map segment cannot be created as specified.

Explanation: An attempt to create a color map segment fails because of illegal, missing, or incompatible parameters or insufficient hardware resources.

User Action: Check for a programming error.

DIGIT_ACTIVE, Digitizing already active.

Explanation: An attempt is made to begin a new digitizing program while another digitizing program is still running. The current digitizing program must be disabled or deleted before a new program can be declared.

User Action: Exit from first digitizing program to run the second.

FONT_TOO_BIG, Specified font is too big for driver font block.

Explanation: An attempt is made to specify a font that does not fit into a QDSS driver font block. The font must at least fit the top raster or each glyph into a single QDSS font block (currently 1024 x 35 pixels).

User Action: Change either the font or the driver font block routines.

INSFARG, Insufficient arguments.

Explanation: A required argument is not specified.

User Action: Check for a programming error.

IN_SEG, Object not in segment.

Explanation: An attempt is made to specify an object that is not in the specified segment.

User Action: Check for a programming error.

NODEFFONT, The default font, !AD, is not in the system font queue.

Explanation: An attempt is made to request a font in attribute block 0 that is not present in the system font queue.

User Action: Check SYS$SYSTEM:UISMEMFONTS.COM.

NO_DEL, Root segment cannot be deleted.

Explanation: An attempt is made to delete the root segment.

User Action: Check for a programming error.

NODEV, No physical display device.

Explanation: An attempt is made to create a display window for a virtual display that has no physical display device associated with it.

User Action: Check your hardware configuration.

NO_END, Root segment not ended.

Explanation: An attempt is made to end the root segment.

User Action: Check for a programming error.

NO_FONT, The font cannot be found.

Explanation: An attempt is made to reference a font that cannot be satisfied, even by looking for other similar fonts. All references to the attribute block that specifies this font produce this error. The program continues after this error.

User Action: Specify font contained in the SYS$FONT directory.

NO_INSERT, Segment cannot be inserted in itself.

Explanation: An attempt is made to insert a segment in itself.

User Action: Check for a programming error.

NO_KB, No keyboard is bound to the specified display window.

Explanation: An attempt is made to specify a display window that is not bound to a keyboard with UIS$BIND_KB.

User Action: Check for a programming error.

NO_TABLET, No tablet device.

Explanation: An attempt is made to use a mouse as a pointing device rather than a tablet. Only a tablet can be used for digitizing.

User Action: Replace the mouse with a tablet.

NOTVAFONT, Font !AS is not a VA font.

Explanation: An attempt is made to load a font or intensity that is invalid for the color workstation.

User Action: Check for a programming error.

NOURG, Cannot disable region AST because no matching region can be found.

Explanation: An attempt is made to disable a user region AST with an ASTADR=0 and the region boundary used in the original enable request. However, no entry is found with matching boundary coordinates. The program must ensure that the boundary coordinates match exactly to disable an existing request.

User Action: Check for a programming error.

SHRINK_ICON, Request to shrink an icon to another icon ignored.

Explanation: An attempt is made to delete a request that an application shrink an icon to an icon. UIS ignores the request.

User Action: Check for a programming error.

TOODEEP, Cannot interrupt allocation more than 5 levels deep.

Explanation: An attempt is made to allocate storage when the allocation routines are already five levels deep.

User Action: Check for an error in the graphics services.

UNSUP_FONT, Font !AS is an unsupported version.

Explanation: An attempt is made to request an activated but unsupported version of a font. The font is not placed in the specified attribute block. The program continues after this error.

User Action: Check for a programming error.

VAFONTERR, Error loading !AS.VWS$VAFONT into the driver.

Explanation: Internal error.

User Action: Submit a Software Performance Report (SPR).

VCM_ACTIVE, Virtual color map is still active.

Explanation: An attempt is made to delete a virtual color map that is still referenced by one or more virtual displays.

User Action: Check for a programming error.

VCM_BOUND, Virtual color map is already bound to a color map segment.

Explanation: An attempt is made to create a color map segment for a virtual color map that is already bound to another color map segment.

User Action: Check for a programming error.

VCM_EXISTS, Virtual color map already accessed by process.

Explanation: An attempt is made to create a virtual color map that already exists.

User Action: Check for a programming error.

VCM_NOTBOUND, Virtual color map is not bound to a color map segment.

Explanation: An attempt is made to create a window to which the virtual display virtual color map is not bound and the NOBIND attribute is specified for the virtual color map.

User Action: Check for a programming error.

VPTOOSMALL, Requested size of the viewport is too small.

Explanation: The desired size of the viewport is too small to be displayed on the screen.

User Action: Request larger viewport.

F VMS Data Types

The VMS Usage entry in the documentation format for system routines indicates the argument VMS data type. Each VMS data type has only one storage representation. For example, the VMS data type **access_mode** is an unsigned byte. In addition, a VMS data type might or might not have a conceptual meaning.

F.1 VMS Data Types

Most VMS data types are conceptual; that is, their meaning is unique in the context of the VMS operating system. For example, the storage representation of data type **access_mode** is an unsigned byte. This unsigned byte designates a hardware access mode and therefore has only four valid values:

0—Kernel mode
1—Executive mode
2—Supervisor mode
3—User mode

However, some VMS data types are not conceptual; that is, they specify a storage representation but carry no other VAX/VMS semantic content. For example, **byte_signed** is not a conceptual data type.

NOTE: **The VMS Usage entry is not a traditional data type such as byte, word, longword, and so on. The VMS Usage entry is significant only in the context of the VMS operating system environment and is intended solely to expedite data declarations within application programs.**

To use the VMS Usage entry, perform the following procedure:

1 Find the data type in Table F–1 and read its definition.

2 Find the same VMS data type in the appropriate VAX language implementation (Tables F–2 through F–7) and corresponding source language type declaration.

3 Use this code as your application program type declaration. Note that, in some instances, you might have to modify the declaration.

Table F–1 lists and describes the VMS data types.

Table F–1 VMS Data Types

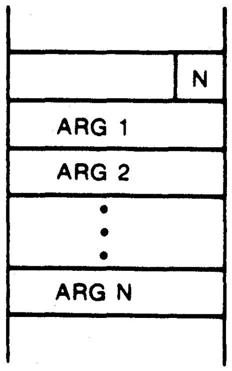
Data Type	Definition
access_bit_names	Each 32 quadword descriptor defines the name of one of the 32 bits in an access mask. The first descriptor names bit <0>, the second descriptor names bit <1>, and so on.
access_mode	This unsigned byte denotes a hardware access mode; takes four values: *0*, kernel mode; *1*, executive mode; *2*, supervisor mode; *3*, user mode.
address	This unsigned longword denotes virtual memory address of either data or code, but not of a procedure entry mask (which is of type **procedure**).
address_range	This unsigned quadword denotes a range of virtual addresses that identify an area of memory. The first longword specifies the beginning address in the range; the second longword specifies the ending address in the range.
arg_list	This procedure argument list consists of 1 to 256 longwords. The first longword contains an unsigned integer count of the number of successive, contiguous longwords, each of which is an argument to be passed to a procedure by means of a VAX CALL instruction. The argument list has the following format:  MLO–1072–87
ast_procedure	This unsigned longword integer denotes the entry mask to a procedure to be called at AST level. (Procedures not to be called at AST level are of type **procedure**.)
boolean	This unsigned longword denotes a Boolean truth value flag with only two values: *1* (true) and *0* (false).
byte_signed	This VMS data type is the same as **byte integer (signed)** in *Introduction to VAX/VMS System Routines,* Table 1-3.
byte_unsigned	This VMS data type is the same as type **byte (unsigned)** in *Introduction to VAX/VMS System Routines,* Table 1-3.
channel	This unsigned word integer is an index to an I/O channel.
char_string	This VMS data type is a string of from 0 to 65,535 8-bit characters, the same as **character string** in *Introduction to VAX/VMS System Routines,* Table 1-3. The following diagram shows the character string XYZ.

Table F-1 (Cont.) VMS Data Types

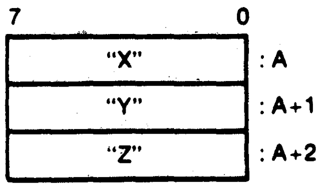
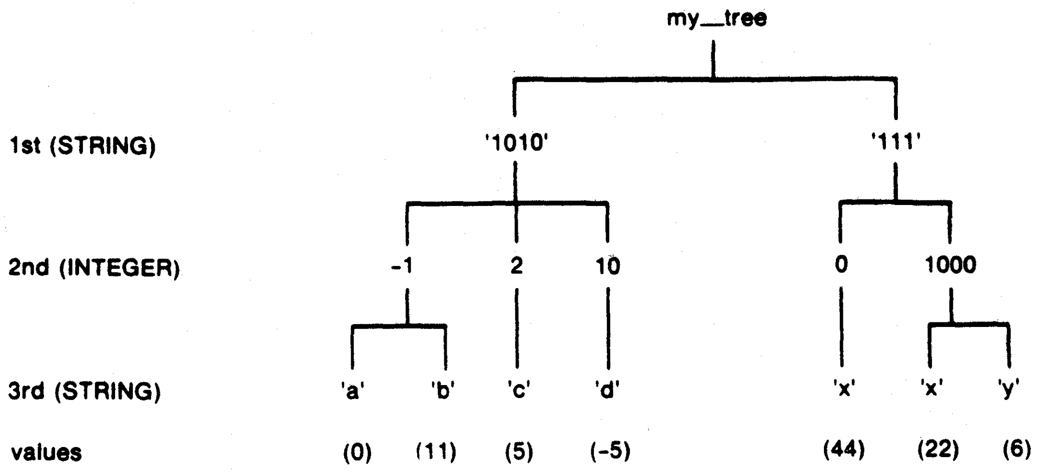
Data Type	Definition
	 MLO-1073-87
complex_number	This number denotes one of the VAX standard complex floating-point data types: F_floating complex, D_floating complex, and G_floating complex. An F_floating complex number (r,i) consists of two F_floating point numbers: **1** The real part (r) of the complex number **2** The imaginary part (i) The structure of an F_floating complex number is as follows:  my_tree 1st (STRING) '1010' '111' 2nd (INTEGER) -1 2 10 0 1000 3rd (STRING) 'a' 'b' 'c' 'd' 'x' 'x' 'y' values (0) (11) (5) (-5) (44) (22) (6) MLO-1074-87 A D_floating complex number (r,i) consists of two D_floating point numbers: **1** The real part (r) of the complex number and **2** The imaginary part (i) The structure of a D_floating complex number is as follows:

Table F-1 (Cont.) VMS Data Types

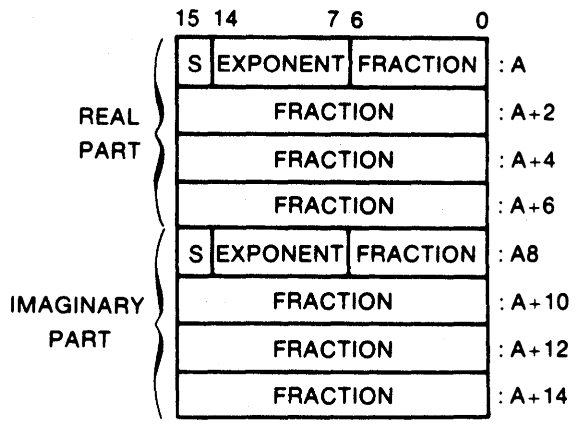
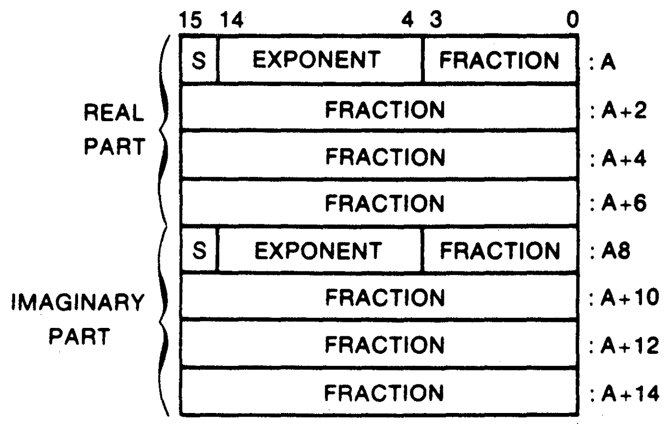
Data Type	Definition
	 15 14 7 6 0 S EXPONENT FRACTION FRACTION FRACTION FRACTION S EXPONENT FRACTION FRACTION FRACTION FRACTION :A :A+2 :A+4 :A+6 :A8 :A+10 :A+12 :A+14 REAL PART IMAGINARY PART MLO-1075-87 A G_floating complex number (r,i) consists of two G_floating point numbers: 1 The real part (r) of the complex number and 2 The imaginary part (i) The structure of a G_floating complex number is as follows:  15 14 4 3 0 S EXPONENT FRACTION FRACTION FRACTION FRACTION S EXPONENT FRACTION FRACTION FRACTION FRACTION :A :A+2 :A+4 :A+6 :A8 :A+10 :A+12 :A+14 REAL PART IMAGINARY PART MLO-1076-87
cond_value	This unsigned longword integer denotes a condition value (that is, a return status or system condition code), typically returned by a procedure in R0. The structure of a condition value is as follows:

Table F-1 (Cont.) VMS Data Types

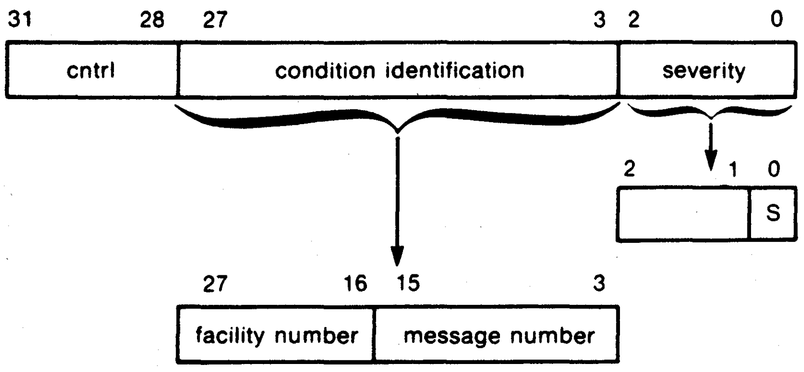
Data Type	Definition
	 31 28 27 3 2 0 cntrl condition identification severity 2 1 0 S 27 16 15 3 facility number message number MLO-1077-87 Depending on your needs, you can test just the low-order bit, the low-order three bits, or the entire value. • The low-order bit indicates successful (1) or unsuccessful (0) completion of the service. • The low-order three bits, taken together, represent the severity of the error. • The remaining bits <31:3> classify the return condition and the operating system component that issued the condition value. Each numeric condition value has a unique symbolic name in the following format, where code is a mnemonic that describes the return condition. SS$_code
context	This unsigned longword is used by a called procedure to maintain position over an iterative sequence of calls. It is usually initialized by the caller, but is thereafter manipulated by the called procedure.
date_time	This 64-bit unsigned, binary integer denotes a date and time as the number of elapsed 100-nanosecond units since 00:00 o'clock, November 17, 1858. This VMS data type is the same as **absolute date and time** in *Introduction to VAX/VMS System Routines*, Table 1-3.
device_name	This character string denotes the 1- to 15-character name of a device. If the string is a logical name, it must translate to a valid device name. If the device name contains a colon (:), the colon and the characters after it are ignored. When an underscore (_) precedes device name string, it indicates the string is a physical device name.
ef_cluster_name	This character string denotes the 1- to 15-character name of an event flag cluster. If the string is a logical name, it must translate to a valid event flag cluster name.
ef_number	This unsigned longword integer denotes the number of an event flag. Local event flags numbered 32 to 63 are available to your programs.
exit_handler_block	This variable-length structure denotes an exit handler control block. The following diagram depicts the control block, which describes the exit handler.

Table F-1 (Cont.) VMS Data Types

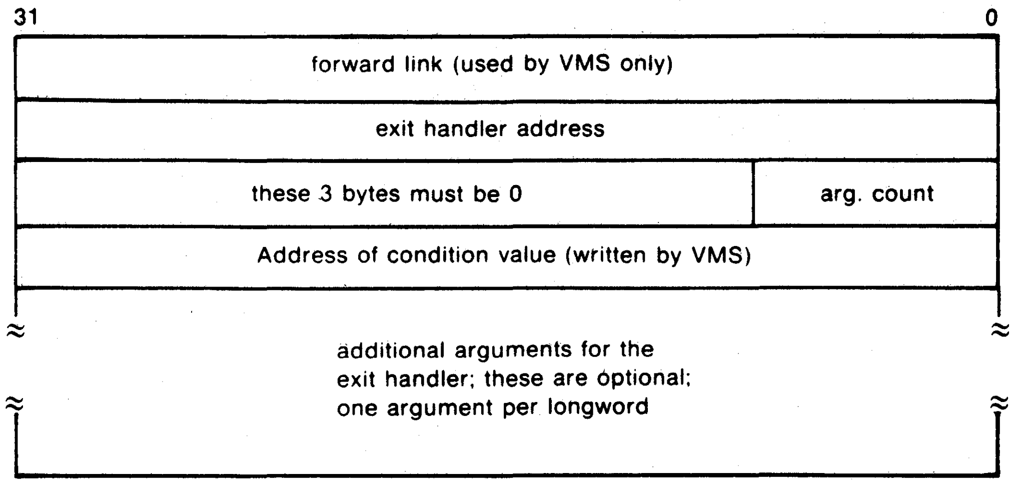
Data Type	Definition
	 31 0 forward link (used by VMS only) exit handler address these 3 bytes must be 0 arg. count Address of condition value (written by VMS) additional arguments for the exit handler; these are optional; one argument per longword MLO-1078-87
fab	This structure denotes an RMS file access block.
file_protection	This unsigned word is a 16-bit mask that specifies file protection. The mask contains four 4-bit fields, each specifying the protection to be applied to file access attempts by one of the four categories of users. From the right-most field to the left-most field: **1** System users **2** File owner **3** Users in the same UIC group as the owner **4** All other users (the world) Each field specifies, from the right-most bit to the left-most bit: **1** Read access **2** Write access **3** Execute access **4** Delete access Set bits indicate that access is denied. The following diagram depicts the 16-bit file-protection mask:

Table F-1 (Cont.) VMS Data Types

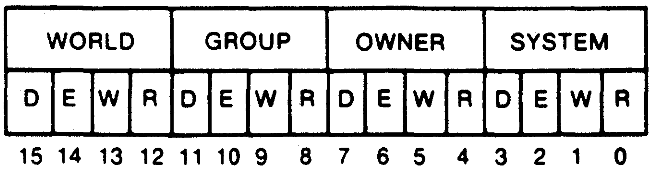
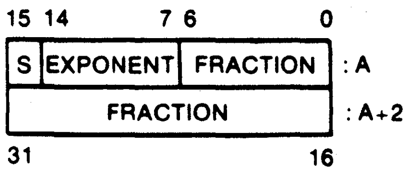
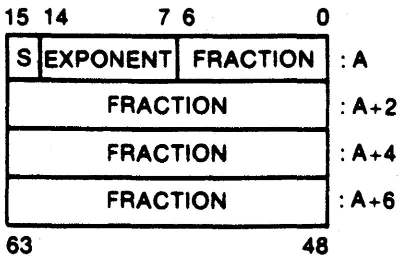
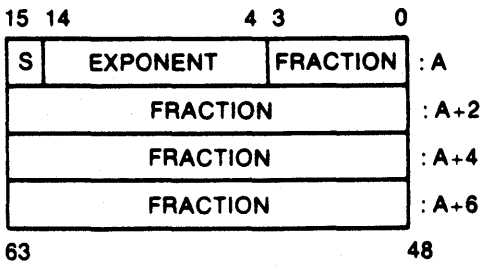
Data Type	Definition
	 WORLD GROUP OWNER SYSTEM D E W R D E W R D E W R D E W R 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 MLO-1079-87
floating_point	This parameter denotes one of the VAX standard floating-point data types: F_floating, D_floating, G_floating, and H_floating. The structure of an F_floating number is as follows:  MLO-1080-87 The structure of a D_floating number is as follows:  15 14 7 6 0 S EXPONENT FRACTION :A FRACTION :A+2 FRACTION :A+4 FRACTION :A+6 63 48 MLO-1081-87 The structure of a G_floating number is as follows:  15 14 4 3 0 S EXPONENT FRACTION :A FRACTION :A+2 FRACTION :A+4 FRACTION :A+6 63 48 MLO-1082-87

Table F-1 (Cont.) VMS Data Types

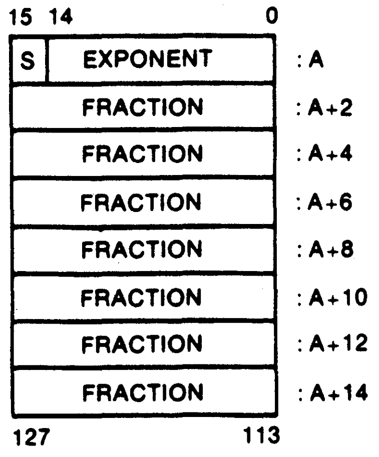
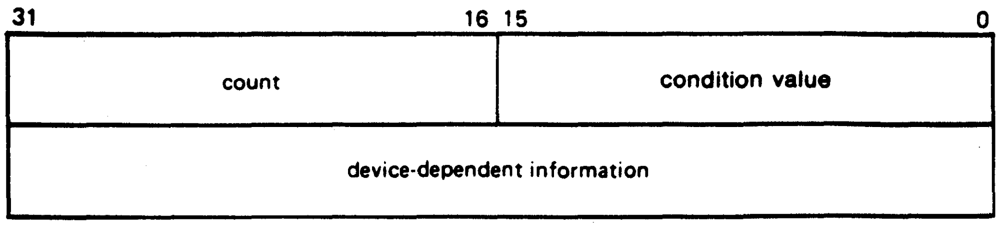
Data Type	Definition
	The structure of an H_floating number is as follows:  15 14 0 S EXPONENT :A FRACTION :A+2 :A+4 :A+6 :A+8 :A+10 :A+12 :A+14 127 113 MLO-1083-87
function_code	This unsigned longword specifes the exact operations a procedure is to perform. This longword has two word-length fields: (1) A number to specify the major operation and (2) A mask or bit vector to specify various suboperations within the major operation.
identifier	This unsigned longword identifies an object returned by the system.
io_status_block	This quadword structure contains information returned by a procedure that completes asynchronously. The returned information varies, depending on the procedure. The following figure illustrates the format of the information written in the IOSB for SYS$QIO.  31 16 15 0 count condition value device-dependent information MLO-1084-87 The first word contains a condition value that indicates the success or failure of the operation. The condition values are the same as those used for all returns from system services; for example, SS$_NORMAL indicates successful completion. The second word contains the number of bytes actually transferred in the I/O operation. Note that for some devices this word contains only the low-order word of the count. The second longword contains device-dependent return information. To ensure successful I/O completion and the integrity of data transfers, check the IOSB should following I/O requests, particularly for device-dependent I/O functions.

Table F-1 (Cont.) VMS Data Types

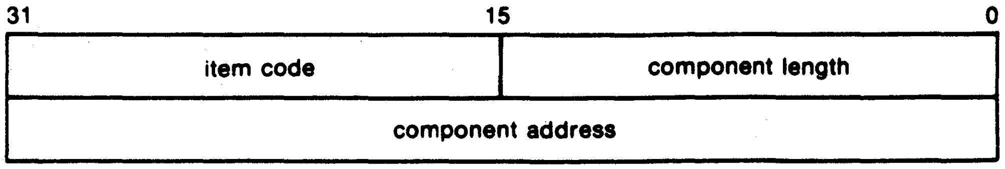
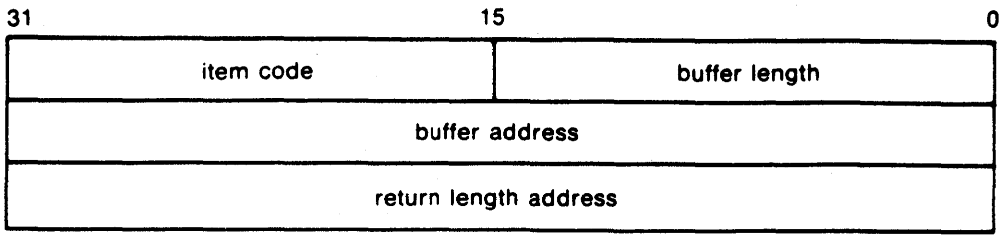
Data Type	Definition
item_list_2	This structure consists of one or more item descriptors and is terminated by a longword containing *0*. Each item descriptor is a 2-longword structure with three fields. The following diagram depicts a single item descriptor.  31 15 0 item code component length component address MLO-1085-87 Field 1 is a word in which the service writes the length (in characters) of the requested component. If the service does not locate the component, it returns the value *0* in this field and in the component address field. Field 2 contains a user-supplied, word-length symbolic code that specifies the component desired. The item codes are defined by the macros specific to the service. Field 3 is a longword where the service writes the starting address of the component. This address is within the input string itself.
item_list_3	This structure consists of one or more item descriptors and is terminated by a longword containing *0*. Each item descriptor is a 3-longword structure that contains four fields. The following diagram depicts the format of a single item descriptor.  31 15 0 item code buffer length buffer address return length address MLO-1086-87 Field 1 is a word containing a user-supplied integer that specifies the length (in bytes) of the buffer in which the service writes the information. The length of the buffer depends upon the item code specified in the item code field of the item descriptor. If the value of buffer length is too small, the service truncates the data. Field 2 is a word with a user-supplied symbolic code that specifies the item of information the service is to return. This code is defined by macros specific to the service. Field 3 is a longword with the user-supplied address of the buffer where the service writes the information. Field 4 is a longword with the user-supplied address of a word where the service writes the length in bytes of the information it actually returns.

Table F-1 (Cont.) VMS Data Types

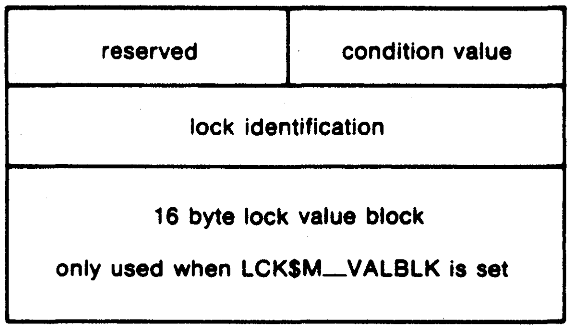
Data Type	Definition
item_list_pair	This structure consists of one or more longword pairs, or *doublets*, and is terminated by a longword containing *0*. Typically, the first longword contains an integer value, such as a code. The second longword can contain a real or integer value.
item_quota_list	This structure consists of one or more quota descriptors and is terminated by a byte with a value defined by the symbolic name PQL$_LISTEND. Each quota descriptor consists of a 1-byte quota name followed by an unsigned longword with the value for that quota.
lock_id	This unsigned longword integer denotes a lock identifier, assigned by the lock manager facility to a lock when the lock is granted.
lock_status_block	The lock manager facility writes status information about a lock into this structure. A lock status block always contains at least two longwords: the first word of the first longword contains a condition value; the second word of the first longword is reserved to DIGITAL; the second longword contains the lock identifier. The lock status block receives the final condition value and the lock identification, and optionally contains a lock value block. When a request is queued, the lock identification is stored in the lock status block, even if the lock has not been granted. This allows a procedure to dequeue locks that have not been granted. The condition value is placed in the lock status block only when the lock is granted (or when errors occur in granting the lock). The following diagram depicts a lock status block that includes the optional 16-byte lock value block.  reserved condition value lock identification 16 byte lock value block only used when LCK$M__VALBLK is set MLO-1087-87
lock_value_block	The lock manager facility includes this 16-byte block in a lock status block if the user requests it. The contents of the lock value block are user-defined and are not interpreted by the lock manager facility.
logical_name	This character string from 1 to 255 characters identifies a logical name or equivalence name to be manipulated by VMS logical name system services. Logical names that denote specific VMS objects have their own VMS types: for example, a logical name identifying a device has the VMS type **device_name**.
longword_signed	This VMS data type is the same as **longword integer (signed)** in *Introduction to VAX/VMS System Routines,* Table 1-3.
longword_unsigned	This VMS data type is the same as **longword (unsigned)** in *Introduction to VAX/VMS System Routines,* Table 1-3.

Table F-1 (Cont.) VMS Data Types

Data Type	Definition
mask_byte	This unsigned byte has each bit interpreted by the called procedure. A mask is also referred to as a set of flags or as a bit mask.
mask_longword	This unsigned longword has each bit interpreted by the called procedure. A mask is also referred to as a set of flags or as a bit mask.
mask_quadword	This unsigned quadword has each bit interpreted by the called procedure. A mask is also referred to as a set of flags or as a bit mask.
mask_word	This unsigned word has each bit interpreted by the called procedure. A mask is also referred to as a set of flags or bit mask.
null_arg	This unsigned longword denotes a null argument that holds a place in the argument list.
octaword_signed	This VMS data type is the same as **octaword integer (signed)** in *Introduction to VAX/VMS System Routines*, Table 1-3.
octaword_unsigned	This VMS data type is the same as **octaword (unsigned)** in *Introduction to VAX/VMS System Routines*, Table 1-3.
page_protection	This unsigned longword specifies page protection to be applied by the VAX hardware. Protection values are specified using bits <3:0>; bits <31:4> are ignored. The $PRTDEF macro defines the following symbolic names for the protection codes: (see table below) If the protection is specified as *0*, the protection defaults to kernel read only.
procedure	This unsigned longword denotes the entry mask to a procedure that is not to be called at AST level. (Arguments that specify procedures to be called at AST level have the VMS type **ast_procedure**.)
process_id	This unsigned longword integer denotes a process identifier (PID). This process identifier is assigned by VMS to a process when the process is created.

Symbol	Description
PRT$C_NA	No access
PRT$C_KR	Kernel read only
PRT$C_KW	Kernel write
PRT$C_ER	Executive read only
PRT$C_EW	Executive write
PRT$C_SR	Supervisor read only
PRT$C_SW	Supervisor write
PRT$C_UR	User read only
PRT$C_UW	User write
PRT$C_ERKW	Executive read; kernel write
PRT$C_SRKW	Supervisor read; kernel write
PRT$C_SREW	Supervisor read; executive write
PRT$C_URKW	User read; kernel write
PRT$C_UREW	User read; executive write
PRT$C_URSW	User read; supervisor write

Table F-1 (Cont.) VMS Data Types

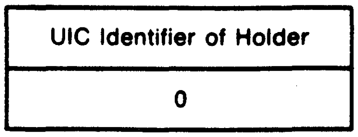
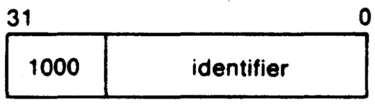
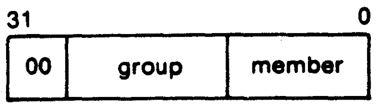
Data Type	Definition
process_name	This character string contains 1 to 15 characters that specify the name of a process.
quadword_signed	This VMS data type is the same as **quadword integer (signed)** in *Introduction to VAX/VMS System Routines,* Table 1-3.
quadword_unsigned	This VMS data type is the same as **quadword (unsigned)** in *Introduction to VAX/VMS System Routines,* Table 1-3.
rights_holder	This unsigned quadword specifes user access rights to a system object contains two fields: (1) An unsigned longword identifier (VMS type **rights_id**) and (2) A longword bit mask where each bit specifies an access right. The following diagram shows the format of a rights holder.  MLO-1088-87
rights_id	This unsigned longword denotes an interest group rights identifier in the context of the VMS security environment. This rights environment can consist of all or part of a UIC (user identification code). Identifiers have two formats in the rights data base: UIC format (VMS type **uic**) and ID format. The high-order bits of the identifier value specify the format of the identifier. Two high-order zero bits identify a UIC format identifier; bit <31>, set to *1*, identifies an ID format identifier. Bit <31>, set to *1*, specifies ID format. Bits <30:28> are reserved by DIGITAL. The remaining bits specify the identifier value. The following diagram depicts the ID format of a rights identifier.  ID Format MLO-1089-87 To the system, an identifier is a binary value; however, to make identifiers easy to use, the system translates the binary identifier value into an identifier name. The binary value and the identifier name are associated in the rights data base. An identifier name consists of 1 to 31 alphanumeric characters and contains at least one nonnumeric character. An identifier name cannot consist entirely of numeric characters. It can include the characters A through Z, dollar signs ($) and underscores (_), as well as the numbers 0 through 9. Any lowercase characters are automatically converted to uppercase.
rab	This structure denotes an RMS record access block.
section_id	This unsigned quadword denotes a global section identifier. This identifier specifies the version of a global section and the criteria to be used in matching that global section.

Table F–1 (Cont.) VMS Data Types

Data Type	Definition
section_name	This character string denotes a 1- to 43-character global-section name. This character string can be a logical name but it must translate to a valid global-section name.
system_access_id	This unsigned quadword denotes a system identification value that is to be associated with a rights data base.
time_name	This character string specifies a time value in VMS format.
uic	This unsigned longword denotes a UIC. Each UIC is unique and represents a system user. The UIC identifier contains two high-order bits that designate format, a member field, and a group field. Member numbers range from 0 to 65,534; group numbers range from 1 to 16,382. The following diagram depicts the UIC format.  UIC Format
user_arg	This unsigned longword denotes a user-defined argument. This longword is passed to a procedure as an argument, but the contents of the longword are defined and interpreted by the user.
varying_arg	This unsigned longword denotes a variable argument. A variable argument can have variable types, depending on specifications made for other arguments in the call.
vector_byte_signed	The elements of this homogeneous array are all signed bytes.
vector_byte_unsigned	The elements of this homogeneous array are all unsigned bytes.
vector_longword_ signed	The elements of this homogeneous array are all signed longwords.
vector_longword_ unsigned	The elements of this homogeneous array are all unsigned longwords.
vector_quadword_ signed	The elements of this homogeneous array are all signed quadwords.
vector_quadword_ unsigned	The elements of this homogeneous array are all unsigned quadwords.
vector_word_signed	The elements of this homogeneous array are all signed words.
vector_word_ unsigned	The elements of this homogeneous array are all unsigned words.
word_signed	This VMS data type is the same as **word integer (signed)** in *Introduction to VAX/VMS System Routines,* Table 1-3.
word_unsigned	This VMS data type is the same as **word (unsigned)** in *Introduction to VAX/VMS System Routines,* Table 1-3.

F.2 VAX BLISS Implementation

Table F–2 lists VMS data types and their corresponding VAX BLISS data type declarations.

Table F-2 VAX BLISS Implementation

VMS Data Type	VAX BLISS Declaration
access_bit_names	BLOCKVECTOR[32,8,BYTE]
access_mode	UNSIGNED BYTE
address	UNSIGNED LONG
address_range	VECTOR[2,LONG,UNSIGNED]
arg_list	VECTOR[n,LONG,UNSIGNED] where n is the number of arguments + 1
ast_procedure	UNSIGNED LONG
boolean	UNSIGNED LONG
byte_signed	SIGNED BYTE
byte_unsigned	UNSIGNED BYTE
channel	UNSIGNED WORD
char_string	VECTOR[65536,BYTE,UNSIGNED]
complex_number	F_Complex: VECTOR[2,LONG] D_Complex: VECTOR[4,LONG] G_Complex: VECTOR[4,LONG] H_Complex: VECTOR[8,LONG]
cond_value	UNSIGNED LONG
context	UNSIGNED LONG
date_time	VECTOR[2,LONG,UNSIGNED]
device_name	VECTOR[n,BYTE,UNSIGNED] where n is the length of the device name
ef_cluster_name	VECTOR[n,BYTE,UNSIGNED] where n is the length of the event flag cluster name
ef_number	UNSIGNED LONG
exit_handler_block	BLOCK[n,BYTE] where n is the size of the exit handler control block
fab	$FAB_DECL (from STARLET.REQ)
file_protection	BLOCK[2,BYTE]
floating_point	F_Floating: VECTOR[1,LONG] D_Floating: VECTOR[2,LONG] G_Floating: VECTOR[2,LONG] H_Floating: VECTOR[4,LONG]
function_code	BLOCK[2,WORD]
identifier	UNSIGNED LONG
io_status_block	BLOCK[8,BYTE]
item_list_2	BLOCKVECTOR[n,8,BYTE] where n is the number of the item descriptors + 1
item_list_3	BLOCKVECTOR[n,12,BYTE] where n is the number of the item descriptors + 1 $ITMLST_DECL/$ITMLST_INIT from STARLET.REQ

Table F-2 (Cont.) VAX BLISS Implementation

VMS Data Type	VAX BLISS Declaration
item_list_pair	BLOCKVECTOR[n,2,LONG] where n is the number of the item descriptors + 1
item_quota_list	BLOCKVECTOR[n,5,BYTE] where n is the number of the quota descriptors + 1
lock_id	UNSIGNED_LONG
lock_status_block	BLOCK[n,BYTE] where n is the size of the lock_status_block –at least 8
lock_value_block	BLOCK[16,BYTE]
logical_name	VECTOR[255,BYTE,UNSIGNED]
longword_signed	SIGNED LONG
longword_unsigned	UNSIGNED LONG
mask_byte	BITVECTOR[8]
mask_longword	BITVECTOR[32]
mask_quadword	BITVECTOR[64]
mask_word	BITVECTOR[16]
null_arg	UNSIGNED LONG
octaword_signed	VECTOR[4,LONG,UNSIGNED]
octaword_unsigned	VECTOR[4,LONG,UNSIGNED]
page_protection	UNSIGNED LONG
procedure	UNSIGNED LONG
process_id	UNSIGNED LONG
process_name	VECTOR[n,BYTE,UNSIGNED] where n is the length of the process name
quadword_signed	VECTOR[2,LONG,UNSIGNED]
quadword_unsigned	VECTOR[2,LONG,UNSIGNED]
rights_holder	BLOCK[8,BYTE]
rights_id	UNSIGNED LONG
rab	$RAB_DECL from STARLET.REQ
section_id	VECTOR[2,LONG,UNSIGNED]
section_name	VECTOR[n,BYTE,UNSIGNED] where n is the length of the global section name
system_access_id	VECTOR[2,LONG,UNSIGNED]
time_name	VECTOR[n,BYTE,UNSIGNED] where n is the length of the time value in VMS format
uic	UNSIGNED LONG
user_arg	UNSIGNED LONG
varying_arg	UNSIGNED LONG
vector_byte_signed	VECTOR[n,BYTE,SIGNED] where n is the size of the array

Table F-2 (Cont.) VAX BLISS Implementation

VMS Data Type	VAX BLISS Declaration
vector_byte_unsigned	VECTOR[n,BYTE,UNSIGNED] where n is the size of the array
vector_longword_signed	VECTOR[n,LONG,SIGNED] where n is the size of the array
vector_longword_ unsigned	VECTOR[n,LONG,UNSIGNED] where n is the size of the array
vector_quadword_ signed	BLOCKVECTOR[n,2,LONG] where n is the size of the array
vector_quadword_ unsigned	BLOCKVECTOR[n,2,LONG] where n is the size of the array
vector_word_signed	VECTOR[n,BYTE,SIGNED] where n is the size of the array
vector_word_unsigned	VECTOR[n,BYTE,UNSIGNED] where n is the size of the array
word_signed	SIGNED WORD
word_unsigned	UNSIGNED WORD

F.3 VAX C Implementation

Table F-3 lists VMS data types and their corresponding VAX C data type declarations.

Table F-3 VAX C Implementation

VMS Data Type	VAX C Declaration
access_bit_names	User-defined [1]
access_mode	unsigned char
address	int*pointer [2,3]
address_range	int*array [2] [2,3,4]
arg_list	User-defined [1]
ast_procedure	Pointer to function [2]
boolean	unsigned long int
byte_signed	char
byte_unsigned	unsigned char
channel	unsigned short int

[1]The declaration of a user-defined data structure depends on how the data is used. You can declare such data structures in a variety of ways, each suitable to specific applications.

[2]The term *pointer* refers to several declarations involving pointers. Pointers are declared with special syntax and associated with the data type of the object being pointed to. This object is often *user-defined*.

[3]The data type specified can be changed to any valid VAX C data type.

[4]The term *array* denotes the syntax of a VAX C array declaration.

Table F-3 (Cont.) VAX C Implementation

VMS Data Type	VAX C Declaration
char_string	char array[n] [4,5]
complex_number	User-defined [1]
cond_value	unsigned long int
context	unsigned long int
date_time	User-defined [1]
device_name	char array[n] [4,5]
ef_cluster_name	char array[n] [4,5]
ef_number	unsigned long int
exit_handler_block	User-defined [1]
fab	#include fab from text library struct FAB
file_protection	unsigned short int, or User-defined [1]
floating_point	float or double
function_code	Unsigned long int or User-defined [1]
identifier	int*pointer [2,3]
io_status_block	User-defined [1]
item_list_2	User-defined [1]
item_list_3	User-defined [1]
item_list_pair	User-defined [1]
item_quota_list	User-defined [1]
lock_id	unsigned long int
lock_status_block	User-defined [1]
lock_value_block	User-defined [1]
logical_name	char array[n] [4,5]
longword_signed	long int
longword_unsigned	unsigned long int
mask_byte	unsigned char
mask_longword	unsigned long int
mask_quadword	User-defined [1]
mask_word	unsigned short int
null_arg	unsigned long int
octaword_signed	User-defined [1]
octaword_unsigned	User-defined [1]

[1]The declaration of a user-defined data structure depends on how the data is used. You can declare such data structures in a variety of ways, each suitable to specific applications.

[2]The term *pointer* refers to several declarations involving pointers. Pointers are declared with special syntax and associated with the data type of the object being pointed to. This object is often *user-defined*.

[3]The data type specified can be changed to any valid VAX C data type.

[4]The term *array* denotes the syntax of a VAX C array declaration.

[5]The size of the array must be substituted for n.

Table F-3 (Cont.) VAX C Implementation

VMS Data Type	VAX C Declaration
page_protection	unsigned long int
procedure	Pointer to function [2]
process_id	unsigned long int
process_name	char array[n] [4,5]
quadword_signed	User-defined [1]
quadword_unsigned	User-defined [1]
rights_holder	User-defined [1]
rights_id	unsigned long int
rab	#include rab from text library struct RAB
section_id	User-defined [1]
section_name	char array[n] [4,5]
system_access_id	User-defined [1]
time_name	char array[n] [4,5]
uic	unsigned long int
user_arg	User-defined [1]
varying_arg	User-defined [1]
vector_byte_signed	char array[n] [4,5]
vector_byte_unsigned	unsigned char array[n] [4,5]
vector_longword_ signed	long int array[n] [4,5]
vector_longword_ unsigned	unsigned long int array[n] [4,5]
vector_quadword_ signed	User-defined [1]
vector_quadword_ unsigned	User-defined [1]
vector_word_signed	short int array[n] [4,5]
vector_word_ unsigned	unsigned short int array[n] [4,5]
word_signed	short int
word_unsigned	unsigned short int

[1]The declaration of a user-defined data structure depends on how the data is used. You can declare such data structures in a variety of ways, each suitable to specific applications.

[2]The term *pointer* refers to several declarations involving pointers. Pointers are declared with special syntax and associated with the data type of the object being pointed to. This object is often *user-defined*.

[4]The term *array* denotes the syntax of a VAX C array declaration.

[5]The size of the array must be substituted for n.

F.4 VAX FORTRAN Implementation

Table F–4 lists VMS data types and their corresponding VAX FORTRAN data type declarations.

Table F–4 VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration
access_bit_names	INTEGER*4(2,32)
	or
	STRUCTURE /access_bit_names/
	INTEGER*4 access_name_len
	INTEGER*4 access_name_buf
	END STRUCTURE !access_bit_names
	RECORD /access_bit_names/ my_names(32)
access_mode	BYTE
address	INTEGER*4
address_range	INTEGER*4(2)
	or
	STRUCTURE /address_range/
	INTEGER*4 low_address
	INTEGER*4 high_address
	END STRUCTURE
arg_list	INTEGER*4(n)
ast_procedure	EXTERNAL
boolean	LOGICAL*4
byte_signed	BYTE
byte_unsigned	BYTE [1]
channel	INTEGER*2
char_string	CHARACTER*n
complex_number	COMPLEX*8
	COMPLEX*16
cond_value	INTEGER*4
context	INTEGER*4
date_time	INTEGER*4(2)
device_name	CHARACTER*n
ef_cluster_name	CHARACTER*n
ef_number	INTEGER*4

[1]Unsigned data types are not directly supported by VAX FORTRAN. However, in most cases you can substitute the signed equivalent so long as you do not exceed the range of the signed data structure.

Table F-4 (Cont.) VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration
exit_handler_block	STRUCTURE /exhblock/ INTEGER*4 flink INTEGER*4 exit_handler_addr BYTE(3) /0/ BYTE arg_count INTEGER*4 cond_value ! . ! . (optional arguments ... ! . one argument per longword) ! END STRUCTURE !cntrlblk RECORD /exhblock/ myexh_block
fab	INCLUDE '($FABDEF)' RECORD /fabdef/ myfab
file_protection	INTEGER*4
floating_point	REAL*4 REAL*8 DOUBLE PRECISION REAL*16
function_code	INTEGER*4
identifier	INTEGER*4
io_status_block	STRUCTURE /iosb/ INTEGER*2 iostat, !return status 2 term_offset, !Loc. of line terminator 2 terminator, !value of terminator 2 term_size !size of terminator END STRUCTURE RECORD /iosb/ my_iosb
item_list_2	STRUCTURE /itmlst/ UNION MAP INTEGER*2 buflen,code INTEGER*4 bufadr END MAP MAP INTEGER*4 end_list /0/ END MAP END UNION END STRUCTURE !itmlst RECORD /itmlst/ my_itmlst_2(n) (Allocate n records where n is the number item codes plus an extra element for the end-of-list item)

Table F-4 (Cont.) VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration
item_list_3	STRUCTURE /itmlst/ UNION MAP INTEGER*2 buflen,code INTEGER*4 bufadr,retlenadr END MAP MAP INTEGER*4 end_list /0/ END MAP END UNION END STRUCTURE !itmlst RECORD /itmlst/ my_itmlst_2(n) (Allocate n records where n is the number item codes plus an extra element for the end-of-list item)
item_list_pair	STRUCTURE /itmlist_pair/ UNION MAP INTEGER*4 code INTEGER*4 value END MAP MAP INTEGER*4 end_list /0/ END MAP END UNION END STRUCTURE !itmlst_pair RECORD /itmlst_pair/ my_itmlst_pair(n) (Allocate n records where n is the number item codes plus an extra element for the end-of-list item)
item_quota_list	STRUCTURE /item_quota_list/ MAP BYTE quota_name INTEGER*4 quota_value END MAP MAP BYTE end_quota_list END MAP END STRUCTURE !item_quota_list
lock_id	INTEGER*4
lock_status_block	STRUCTURE/lksb/ INTEGER*2 cond_value INTEGER*2 unused INTEGER*4 lock_id BYTE(16) END STRUCTURE !lock_status_lock
lock_value_block	BYTE(16)
logical_name	CHARACTER*n
longword_signed	INTEGER*4

Table F-4 (Cont.) VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration
longword_unsigned	INTEGER*4 [1]
mask_byte	INTEGER*1
mask_longword	INTEGER*4
mask_quadword	INTEGER*4(2)
mask_word	INTEGER*2
null_arg	%VAL(0)
octaword_signed	INTEGER*4(4)
octaword_unsigned	INTEGER*4(4) [1]
page_protection	INTEGER*4
procedure	INTEGER*4
process_id	INTEGER*4
process_name	CHARACTER*n
quadword_signed	INTEGER*4(2)
quadword_unsigned	INTEGER*4(2) [1]

rights_holder	INTEGER*4(2)
	or
	STRUCTURE /rights_holder/
	    INTEGER*4 rights_id
	    INTEGER*4 rights_mask
	END STRUCTURE !rights_holder

rights_id	INTEGER*4
rab	INCLUDE '($RABDEF)' RECORD /rabdef/ myrab
section_id	INTEGER*4(2)
section_name	CHARACTER*n
system_access_id	INTEGER*4(2)
time_name	CHARACTER*23
uic	INTEGER*4
user_arg	Any longword quantity
varying_arg	INTEGER*4
vector_byte_signed	BYTE(n)
vector_byte_unsigned	BYTE(n) [1]
vector_longword_signed	INTEGER*4(n)
vector_longword_unsigned	INTEGER*4(n) [1]
vector_quadword_signed	INTEGER*4(2, n)
vector_quadword_unsigned	INTEGER*4(2,n) [1]

[1]Unsigned data types are not directly supported by VAX FORTRAN. However, in most cases you can substitute the signed equivalent so long as you do not exceed the range of the signed data structure.

Table F-4 (Cont.) VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration
vector_word_signed	INTEGER*2(n)
vector_word_unsigned	INTEGER*2(n) [1]
word_signed	INTEGER*2(n)
word_unsigned	INTEGER*2(n) [1]

[1]Unsigned data types are not directly supported by VAX FORTRAN. However, in most cases you can substitute the signed equivalent so long as you do not exceed the range of the signed data structure.

F.5 VAX MACRO Implementation

Table F-5 lists VMS data types and their corresponding VAX MACRO data type declarations.

Table F-5 VAX MACRO Implementation

VMS Data Type	VAX MACRO Declaration
access_bit_names	.ASCID /name_for_bit0/ .ASCID /name_for_bit1/ . . . .ASCID /name_for_bit31/
access_mode	.BYTE PSL$C_xxxx
address	.ADDRESSS virtual_address
address_range	.ADDRESS start_address,end_address
arg_list	.LONG n_args, arg1, arg2,...
ast_procedure	.ADDRESS ast_procedure
boolean	.LONG 1 or .LONG 0
byte_signed	.SIGNED_BYTE byte_value
byte_unsigned	.BYTE byte_value
channel	.WORD channel_number
char_string	.ASCID /string/
complex_number	NA
cond_value	.LONG cond_value
context	.LONG 0
date_time	.QUAD date_time
device_name	.ASCID /ddcu:/
ef_cluster_name	.ASCID /ef_cluster_name/
ef_number	.LONG ef_number

Table F-5 (Cont.) VAX MACRO Implementation

VMS Data Type	VAX MACRO Declaration
exit_handler_block	.LONG 0 .ADDRESS exit_handler_routine .LONG 1 .ADDRESS status STATUS: .BLKL 1
fab	MYFAB: $FAB
file_protection	.WORD prot_value
floating_point	.FLOAT, .G_FLOAT, or .H_FLOAT
function_code	.LONG code!mask
identifier	.ADDRESSS virtual_address
io_status_block	.QUAD 0
item_list_2	.WORD component_length .WORD item_code .ADDRESS component_address
item_list_3	.WORD buffer_length .WORD item_code .ADDRESS buffer_address .ADDRESS return_length_address
item_list_pair	.LONG item_code .LONG data
item_quota_list	.BYTE PQL$_xxxx .LONG value_for_quota .BYTE pql$_listend
lock_id	.LONG lock_id
lock_status_block	.QUAD 0
lock_value_block	.BLKB 16
logical_name	.ASCID /logical_name/
longword_signed	.LONG value
longword_unsigned	.LONG value
mask_byte	.BYTE mask_byte
mask_longword	.LONG mask_longword
mask_quadword	.QUAD mask_quadword
mask_word	.WORD mask_word
null_arg	.LONG 0
octaword_signed	NA
octaword_unsigned	.OCTA value
page_protection	.LONG page_protection
procedure	.ADDRESS procedure
process_id	.LONG process_id
process_name	.ASCID /process_name/
quadword_signed	NA

Table F-5 (Cont.) VAX MACRO Implementation

VMS Data Type	VAX MACRO Declaration
quadword_unsigned	.QUAD value
rights_holder	.LONG identifier, access_right_bitmask
rights_id	.LONG rights_id
rab	MYRAB: $RAB
section_id	.LONG sec$k_matXXX, version_number
section_name	.ASCID /section_name/
system_access_id	.QUAD system_access_id
time_name	.ASCID /dd-mmm-yyyy:hh:mm:ss.cc/
uic	.LONG uic
user_arg	.LONG data
varying_arg	Dependent upon application
vector_byte_signed	.SIGNED_BYTE val1,val2,...valN
vector_byte_unsigned	.BYTE val1,val2,...valN
vector_longword_ signed	.LONG val1,val2,...valN
vector_longword_ unsigned	.LONG val1,val2,...valN
vector_quadword_ signed	NA
vector_quadword_ unsigned	.QUAD val1 .QUAD val2 . . . .QUAD valN
vector_word_signed	.SIGNED_WORD val1,val2,...valN
vector_word_ unsigned	.WORD val1,val2,...valN
word_signed	.SIGNED_WORD value
word_unsigned	.WORD value

F.6 VAX PASCAL Implementation

Table F-6 lists VMS data types and their corresponding VAX PASCAL data type declarations:

Table F-6 VAX PASCAL Implementation

VMS Data Type	VAX PASCAL Declaration
access_bit_names	PACKED ARRAY [1..32] OF [QUAD] RECORD END; [1,2]
access_mode	[BYTE] 0..3; [2]
address	UNSIGNED;
address_range	PACKED ARRAY [1..2] OF UNSIGNED; [2]
arg_list	PACKED ARRAY [1..n] OF UNSIGNED; [2]
ast_procedure	UNSIGNED;
boolean	BOOLEAN; [3]
byte_signed	[BYTE] -128..127; [2]
byte_unsigned	[BYTE] 0..255; [2]
channel	[WORD] 0..65535; [2]
char_string	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR; [4]
complex_number	[LONG(2)] RECORD END; * F_Floating Complex * [1,2] [QUAD(2)] RECORD END; * D/G_Floating Complex * [OCTA(2)] RECORD END; * H_Floating Complex *
cond_value	UNSIGNED;
context	UNSIGNED;
date_time	[QUAD] RECORD END; [1,2]
device_name	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR; [4]
ef_cluster_name	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR; [4]
ef_number	UNSIGNED;
exit_handler_block	PACKED ARRAY [1..n] OF UNSIGNED; [2]
fab	FAB$TYPE; [5]
file_protection	[WORD] RECORD END; [1,2]

floating_point	REAL; { F_Floating }
	SINGLE; { F_Floating }
	DOUBLE; { D_Floating/G_Floating } [6]
	QUADRUPLE; { H_Floating }

function_code	UNSIGNED;
identifier	UNSIGNED;
io_status_block	[QUAD] RECORD END; [1,2]

Table F-6 (Cont.) VAX PASCAL Implementation

VMS Data Type	VAX PASCAL Declaration

[1]This type is not available in VAX PASCAL and an empty record has been inserted. To manipulate the contents, declare with explicit field components. If you pass an empty record as a parameter to a PASCAL routine, you must use the VAR keyword.

[2]VAX PASCAL expects either a type identifier or conformant schema. Declare this under the TYPE declaration and use the type identifier in the formal parameter declaration.

[3]VAX PASCAL allocates a byte for a BOOLEAN variable. Use the [LONG] attribute when passing to routines that expect a longword.

[4]This parameter declaration accepts VARYING OF CHAR or PACKED ARRAY OF CHAR and produces the CLASS_S descriptor required by system services.

[5]The program must inherit the STARLET environment file located in SYS$LIBRARY:STARLET.PEN.

[6]If the [G_FLOATING] attribute is used in compiling, double-precision variables and expressions are represented in G_floating format. The /G_FLOATING command line qualifier can also be used. Both methods default to no G_floating.

Table F-6 (Cont.) VAX PASCAL Implementation

VMS Data Type	VAX PASCAL Declaration
item_list_2	PACKED ARRAY [1..n] OF PACKED RECORD [2] CASE INTEGER OF 1: (FIELD1 : [WORD] 0..65535; FIELD2 : [WORD] 0..65535; FIELD3 : UNSIGNED); 2: (TERMINATOR : UNSIGNED); END;
item_list_3	PACKED ARRAY [1..n] OF PACKED RECORD [2] CASE INTEGER OF 1: (FIELD1 : [WORD] 0..65535; FIELD2 : [WORD] 0..65535; FIELD3 : UNSIGNED; FIELD4 : UNSIGNED); 2: (TERMINATOR : UNSIGNED); END;
item_list_pair	PACKED ARRAY [1..n] OF PACKED RECORD [2] CASE INTEGER OF 1: (FIELD1 : INTEGER; FIELD2 : INTEGER); 2: (TERMINATOR : UNSIGNED); END;
item_quota_list	PACKED ARRAY [1..n] OF PACKED RECORD [2] CASE INTEGER OF 1: (QUOTA_NAME : [BYTE] 0..255; QUOTA_VALUE: UNSIGNED); 2: (QUOTA_TERM : [BYTE] 0..255); END;
lock_id	UNSIGNED;
lock_status_block	[BYTE(24)] RECORD END; [1,2]
lock_value_block	[BYTE(16)] RECORD END; [1,2]
logical_name	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR; [4]
longword_signed	INTEGER;
longword_unsigned	UNSIGNED;

[1]This type is not available in VAX PASCAL and an empty record has been inserted. To manipulate the contents, declare with explicit field components. If you pass an empty record as a parameter to a PASCAL routine, you must use the VAR keyword.

[2]VAX PASCAL expects either a type identifier or conformant schema. Declare this under the TYPE declaration and use the type identifier in the formal parameter declaration.

[4]This parameter declaration accepts VARYING OF CHAR or PACKED ARRAY OF CHAR and produces the CLASS_S descriptor required by system services.

Table F-6 (Cont.) VAX PASCAL Implementation

VMS Data Type	VAX PASCAL Declaration
mask_byte	[BYTE,UNSAFE] PACKED ARRAY [1..8] OF BOOLEAN; [2]
mask_longword	[LONG,UNSAFE] PACKED ARRAY [1..32] OF BOOLEAN; [2]
mask_quadword	[QUAD,UNSAFE] PACKED ARRAY [1..64] OF BOOLEAN; [2]
mask_word	[WORD,UNSAFE] PACKED ARRAY [1..16] OF BOOLEAN; [2]
null_arg	UNSIGNED;
octaword_signed	[OCTA] RECORD END; [1,2]
octaword_unsigned	[OCTA] RECORD END; [1,2]
page_protection	[LONG] 0..7; [2]
procedure	UNSIGNED;
process_id	UNSIGNED;
process_name	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR; [4]
quadword_signed	[QUAD] RECORD END; [1,2]
quadword_unsigned	[QUAD] RECORD END; [1,2]
rights_holder	[QUAD] RECORD END; [1,2]
rights_id	UNSIGNED;
rab	RAB$TYPE; [5]
section_id	[QUAD] RECORD END; [1,2]
section_name	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR; [4]
system_access_id	[QUAD] RECORD END; [1,2]
time_name	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR; [4]
uic	UNSIGNED;
user_arg	[UNSAFE] UNSIGNED;
varying_arg	[UNSAFE,REFERENCE] PACKED ARRAY [L..U:INTEGER] OF [BYTE] 0..255;
vector_byte_signed	PACKED ARRAY [1..n] OF [BYTE] -128..127; [2]
vector_byte_unsigned	PACKED ARRAY [1..n] OF [BYTE] 0..255; [2]
vector_longword_signed	PACKED ARRAY [1..n] OF INTEGER; [2]
vector_longword_ unsigned	PACKED ARRAY [1..n] OF UNSIGNED; [2]
vector_quadword_signed	PACKED ARRAY [1..n] OF [QUAD] RECORD END; [1,2]
vector_quadword_ unsigned	PACKED ARRAY [1..n] OF [QUAD] RECORD END; [1,2]

[1]This type is not available in VAX PASCAL and an empty record has been inserted. To manipulate the contents, declare with explicit field components. If you pass an empty record as a parameter to a PASCAL routine, you must use the VAR keyword.

[2]VAX PASCAL expects either a type identifier or conformant schema. Declare this under the TYPE declaration and use the type identifier in the formal parameter declaration.

[4]This parameter declaration accepts VARYING OF CHAR or PACKED ARRAY OF CHAR and produces the CLASS_S descriptor required by system services.

[5]The program must inherit the STARLET environment file located in SYS$LIBRARY:STARLET.PEN.

Table F-6 (Cont.) VAX PASCAL Implementation

VMS Data Type	VAX PASCAL Declaration
vector_word_signed	PACKED ARRAY [1..n] OF [WORD] -32768..32767; [2]
vector_word_unsigned	PACKED ARRAY [1..n] OF [WORD] 0..65535; [2]
word_signed	[WORD] -32768..32767; [2]
word_unsigned	[WORD] 0..65535; [2]

[2]VAX PASCAL expects either a type identifier or conformant schema. Declare this under the TYPE declaration and use the type identifier in the formal parameter declaration.

F.7 VAX PL/I Implementation

Table F-7 lists VMS data types and their corresponding VAX PL/I data type declarations.

Table F-7 VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration
access_bit_names	1 ACCESS_BIT_NAMES(32), 2 LENGTH FIXED BINARY(15), 2 DTYPE FIXED BINARY(7) INITIAL((32)DSC$K_ DTYPE_T), 2 CLASS FIXED BINARY(7) INITIAL((32)DSC$K_ CLASS_S), 2 CHAR_PTR POINTER; [1] The length of the LENGTH field in each element of the array should correspond to the length of a string of characters pointed to by the CHAR_PTR field. The constants DST$K_CLASS_S and DST$K_DTYPE_T can be used by including the module $DSCDEF from PLISTARLET or by declaring it GLOBALREF FIXED BINARY(31) VALUE.
access_mode	FIXED BINARY(7) (The constants for this type— PSLC_KERNEL, PSLC_EXEC, PSLC_SUPER, PSLC_USER—are declared in module $PSLDEF in PLISTARLET.)[3]
address	POINTER
address_range	(2) POINTER [1]
arg_list	1 ARG_LIST BASED, 2 ARGCOUNT FIXED BINARY(31), 2 ARGUMENT (X REFER (ARGCOUNT)) POINTER; [1]

[1]Routines declared in PLISTARLET often use ANY, so you can declare the data structure in the most convenient way for the application. ANY might be necessary in some cases, since PL/I does not allow parameter declarations for some data types used by VMS. (In particular, PL/I parameters with arrays passed by reference cannot be declared to have nonconstant bounds.)

[3]System routines are often written so the parameter passed occupies more storage than the object requires. For example, some system services have parameters that return a bit value as a longword. Those variables must be declared BIT(32) ALIGNED (not BIT(n) ALIGNED) so adjacent storage is not overwritten by return values or incorrectly used as input. (Longword parameters are always declared BIT(32) ALIGNED.)

Table F-7 (Cont.) VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration
	If the arguments are passed by value, it may be appropriate to change the type of the ARGUMENT field of the structure. Alternatively, you can use the POSINT, INT, or UNSPEC built-in functions/pseudovariables to access the data. X should be an expression with a value in the range 0-255 at the time the structure is allocated.
ast_procedure	PROCEDURE or ENTRY [2]
boolean	BIT ALIGNED [3]
byte_signed	FIXED BINARY(7)
byte_unsigned	FIXED BINARY(7) [4]
channel	FIXED BINARY(15)
char_string	CHARACTER(n) [5]
complex_number	(2) FLOAT BINARY(n) (See floating_point for values of n.)
cond_value	See module STS$VALUE in PLISTARLET [1]
context	FIXED BINARY(31)
date_time	BIT(64) ALIGNED [6]
device_name	CHARACTER(n) [5]
ef_cluster_name	CHARACTER(n) [5]
ef_number	FIXED BINARY(31)

exit_handler_block	1 EXIT_HANDLER_BLOCK BASED,
	    2 FORWARD_LINK POINTER,
	    2 HANDLER POINTER,
	2 ARGCOUNT FIXED BINARY(31),
	    2 ARGUMENT (n REFER
	    (ARGCOUNT)) POINTER; [1]
	Replace n with an expression that will yield a value between 0 and 255 at the time the structure is allocated.

fab	See module $FABDEF in PLISTARLET [1]

[1]Routines declared in PLISTARLET often use ANY, so you can declare the data structure in the most convenient way for the application. ANY might be necessary in some cases, since PL/I does not allow parameter declarations for some data types used by VMS. (In particular, PL/I parameters with arrays passed by reference cannot be declared to have nonconstant bounds.)

[2]AST procedures and those passed as parameters of type ENTRY VALUE or ANY VALUE must be external procedures. This applies to all system routines that take procedure parameters.

[3]System routines are often written so the parameter passed occupies more storage than the object requires. For example, some system services have parameters that return a bit value as a longword. Those variables must be declared BIT(32) ALIGNED (not BIT(n) ALIGNED) so adjacent storage is not overwritten by return values or incorrectly used as input. (Longword parameters are always declared BIT(32) ALIGNED.)

[4]This is actually an unsigned integer. This declaration is interpreted as a signed number; use the POSINT function to determine the actual value.

[5]System services require CHARACTER string representation for parameters. Most other system routines allow either CHARACTER or CHARACTER VARYING. For parameter declarations, n should be an asterisk.

[6]VAX PL/I does not support FIXED BINARY numbers with precisions greater than 32. To use larger values, declare variables to be BIT variables of the appropriate size and use the POSINT and SUBSTR bits as necessary to access the values, or declare the item as a structure. The RTL routines LIB$ADDX and LIB$SUBX might be useful if you perform arithmetic on these types.

Table F-7 (Cont.) VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration
file_protection	BIT(16) ALIGNED [3]
floating_point	FLOAT BINARY(n) The values for n are as follows: 1 <= n <= 24 - F floating 25 <= n <= 53 - D floating 25 <= n <= 53 - G floating (with /G_FLOAT) 54 <= n <= 113 - H floating
function_code	BIT(32) ALIGNED
identifier	POINTER

io_status_block	Since there are different formats for I/O status blocks for various system services, different definitions will be appropriate for different uses. Some of the common formats are shown here. [1]
	
	See p. SYS-229
	1 IOSB_SYS$GETSYI,
	        2 STATUS FIXED BINARY(31),
	        2 RESERVED FIXED BINARY(31);
	
	See Fig. 8-16 in Part I of the *I/O User's Guide*
	1 IOSB_TTDRIVER_A,
	        2 STATUS FIXED BINARY(15),
	        2 BYTE_COUNT FIXED BINARY(15),
	        2 MBZ FIXED BINARY(31) INITIAL(0);
	
	See Fig. 8-16 in Part I of the *I/O User's Guide*
	1 IOSB_TTDRIVER_B,
	        2 STATUS FIXED BINARY(15),
	        2 TRANSMIT_SPEED FIXED BINARY(7),
	        2 RECEIVE_SPEED FIXED BINARY(7),
	        2 CR_FILL FIXED BINARY(7),
	        2 LF_FILL FIXED BINARY(7),
	        2 PARITY_FLAGS FIXED BINARY(7),
	        2 MBZ FIXED BINARY(7) INITIAL(0);

item_list_2	1 ITEM_LIST_2,
	        2 ITEM(SIZE),
	           3 COMPONENT_LENGTH FIXED BINARY(15), 3 ITEM_CODE FIXED BINARY(15),
	           3 COMPONENT_ADDRESS POINTER,
	        2 TERMINATOR FIXED BINARY(31) INITIAL(0);[1]
	Replace SIZE with the number of items you want.

[1]Routines declared in PLISTARLET often use ANY, so you can declare the data structure in the most convenient way for the application. ANY might be necessary in some cases, since PL/I does not allow parameter declarations for some data types used by VMS. (In particular, PL/I parameters with arrays passed by reference cannot be declared to have nonconstant bounds.)

[3]System routines are often written so the parameter passed occupies more storage than the object requires. For example, some system services have parameters that return a bit value as a longword. Those variables must be declared BIT(32) ALIGNED (not BIT(n) ALIGNED) so adjacent storage is not overwritten by return values or incorrectly used as input. (Longword parameters are always declared BIT(32) ALIGNED.)

Table F-7 (Cont.) VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration

item_list_3	
	```
	1 ITEM_LIST_3,
	        2 ITEM(SIZE),
	          3 BUFFER_LENGTH FIXED BINARY(15),
	          3 ITEM_CODE FIXED BINARY(15),
	          3 BUFFER_ADDRESS POINTER,
	          3 RETURN_LENGTH POINTER,
	        2 TERMINATOR FIXED BINARY(31) INITIAL(0);[1]
	```
	 Replace SIZE with the number of items you want.

item_list_pair	
	```
	1 ITEM_LIST_PAIR,
	        2 ITEM(SIZE),
	          3 ITEM_CODE FIXED BINARY(31),
	          3 ITEM UNION,
	            4 INTEGER FIXED BINARY(31),
	            0 REAL FLOAT BINARY(24),
	        2 TERMINATOR FIXED BINARY(31) INITIAL(0);[1]
	```
	 Replace SIZE with the number of items you want.

item_quota_list	
	```
	1 ITEM_QUOTA_LIST,
	        2 QUOTA(SIZE),
	          3 NAME FIXED BINARY(7),
	          3 VALUE FIXED BINARY(31),
	        2 TERMINATOR FIXED BINARY(7)
	INITIAL(PQL$_LISTEND); [1]
	```
	 Replace SIZE with the number of quota entries that you want to use. The constant PQL$_LISTEND can be used by including the module $PQLDEF from PLISTARLET or by declaring it GLOBALREF FIXED BINARY(31) VALUE.

lock_id	FIXED BINARY(31)

lock_status_block	
	```
	1 LOCK_STATUS_BLOCK,
	        2 STATUS_CODE FIXED BINARY(15),
	        2 RESERVED FIXED BINARY(15),
	        2 LOCK_ID FIXED BINARY(31); [1]
	```

lock_value_block	The declaration of an item of this structure will depend on the use of the structure, since VMS does not interpret the value. [1]
logical_name	CHARACTER(n) [5]
longword_signed	FIXED BINARY(31)
longword_unsigned	FIXED BINARY(31) [4]
mask_byte	BIT(8) ALIGNED
mask_longword	BIT(32) ALIGNED
mask_quadword	BIT(64) ALIGNED

[1]Routines declared in PLISTARLET often use ANY, so you can declare the data structure in the most convenient way for the application. ANY might be necessary in some cases, since PL/I does not allow parameter declarations for some data types used by VMS. (In particular, PL/I parameters with arrays passed by reference cannot be declared to have nonconstant bounds.)

[4]This is actually an unsigned integer. This declaration is interpreted as a signed number; use the POSINT function to determine the actual value.

[5]System services require CHARACTER string representation for parameters. Most other system routines allow either CHARACTER or CHARACTER VARYING. For parameter declarations, n should be an asterisk.

Table F-7 (Cont.) VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration
mask_word	BIT(16) ALIGNED
null_arg	Omit the corresponding parameter in the call. For example, FOO(A,,B) would omit the second parameter.
octaword_signed	BIT(128) ALIGNED [6]
octaword_unsigned	BIT(128) ALIGNED [4,6]
page_protection	FIXED BINARY(31) (The constants for this type are declared in module $PRTDEF in PLISTARLET.)
procedure	PROCEDURE or ENTRY [2]
process_id	FIXED BINARY(31)
process_name	CHARACTER(n) [5]
quadword_signed	BIT(64) ALIGNED [6]
quadword_unsigned	BIT(64) ALIGNED [4,6]
rights_holder	1 RIGHTS_HOLDER, 2 RIGHTS_ID FIXED BINARY(31), 2 ACCESS_RIGHTS BIT(32) ALIGNED; [1]
rights_id	FIXED BINARY(31)
rab	See module $RABDEF in PLISTARLET [1]
section_id	BIT(64) ALIGNED
section_name	CHARACTER(n) [5]
system_access_id	BIT(64) ALIGNED
time_name	CHARACTER(n) [5]
uic	FIXED BINARY(31)
user_arg	ANY
varying_arg	ANY with OPTIONS(VARIABLE) on the routine declaration.
vector_byte_signed	(n) FIXED BINARY(7) [7]
vector_byte_unsigned	(n) FIXED BINARY(7) [4,7]

[1]Routines declared in PLISTARLET often use ANY, so you can declare the data structure in the most convenient way for the application. ANY might be necessary in some cases, since PL/I does not allow parameter declarations for some data types used by VMS. (In particular, PL/I parameters with arrays passed by reference cannot be declared to have nonconstant bounds.)

[2]AST procedures and those passed as parameters of type ENTRY VALUE or ANY VALUE must be external procedures. This applies to all system routines that take procedure parameters.

[4]This is actually an unsigned integer. This declaration is interpreted as a signed number; use the POSINT function to determine the actual value.

[5]System services require CHARACTER string representation for parameters. Most other system routines allow either CHARACTER or CHARACTER VARYING. For parameter declarations, n should be an asterisk.

[6]VAX PL/I does not support FIXED BINARY numbers with precisions greater than 32. To use larger values, declare variables to be BIT variables of the appropriate size and use the POSINT and SUBSTR bits as necessary to access the values, or declare the item as a structure. The RTL routines LIB$ADDX and LIB$SUBX might be useful if you perform arithmetic on these types.

[7]For parameter declarations, the bounds must be constant for arrays passed by reference. For arrays passed by descriptor, use asterisks for the array extent instead. (VMS system routines almost always take arrays by reference.)

Table F-7 (Cont.) VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration
vector_longword_signed	(n) FIXED BINARY(31) [7]
vector_longword_unsigned	(n) FIXED BINARY(31) [4,7]
vector_quadword_signed	(n) BIT(64) ALIGNED [6,7]
vector_quadword_unsigned	(n) BIT(64) ALIGNED [4,6,7]
vector_word_signed	(n) FIXED BINARY(15) [7]
vector_word_unsigned	(n) FIXED BINARY(15) [4,7]
word_signed	FIXED BINARY(15)
word_unsigned	FIXED BINARY(15) [4]

[4]This is actually an unsigned integer. This declaration is interpreted as a signed number; use the POSINT function to determine the actual value.

[6]VAX PL/I does not support FIXED BINARY numbers with precisions greater than 32. To use larger values, declare variables to be BIT variables of the appropriate size and use the POSINT and SUBSTR bits as necessary to access the values, or declare the item as a structure. The RTL routines LIB$ADDX and LIB$SUBX might be useful if you perform arithmetic on these types.

[7]For parameter declarations, the bounds must be constant for arrays passed by reference. For arrays passed by descriptor, use asterisks for the array extent instead. (VMS system routines almost always take arrays by reference.)

NOTE: **All system services and many system constants and data structures are declared in PLISTARLET.TLB. For examples of system services, see either the *VAX-11 PL/I User's Guide* or *Programming in VAX-11 PL/I.***

Important note: While the current version of VAX PL/I Version 2 does not support unsigned fixed binary numbers or fixed binary numbers with a precision greater than 31, future versions may support these features. If VAX PL/I is extended to support these types, declarations in PLISTARLET will change to use the new data types where appropriate.

Glossary

absolute pointing device: A pointing device that reports all movement to the workstation.

array: Any organized arrangement of related elements.

address: A 32-bit VAX address positioned in a longword item.

argument list: A vector of longwords that represents a procedure parameter list and possibly a function value.

aspect ratio: The ratio between the height and width of a graphic object. In reference to a virtual display, the aspect ratio is a comparison of the relative proportions of the vertical and horizontal components of objects in the virtual display.

attribute: A quality or characteristic that determines the appearance of an object displayed on the screen. For example, the attributes of a line are its width, style, and color.

baseline: The side of a geometric object or drawing from which the object is constructed or drawn.

call: The transfer of processing control to a specified subroutine.

Cartesian coordinate system: A system of measuring distances in which the location of a point is defined as its distance from two straight lines that intersect at right angles. It is used as the basis of coordinate measurements in computer graphics systems.

clipping: Any graphic data outside a specified boundary that are removed from the display or the file. It is often used in mapping applications to remove data that would otherwise confuse the image being represented.

clipping rectangle: The physical limit in a graphics file beyond which data are either not visible or automatically deleted.

color palette: Number of possible colors you can specify.

condition value: A 32-bit value used to identify uniquely an exception condition. A condition value can be returned to a calling program as a function value or signaled using the VAX signaling mechanism.

coordinate system origin: Center of the coordinate system.

current text position: The world coordinate position that defines the current drawing location for UIS text routines.

cursor: A position indicator used on a display screen to pinpoint where data will be displayed. The cursor is often represented by a blinking block character.

data tablet: The name for a variety of data entry devices consisting of a stylus (pen) or puck, and a board with a coordinate grid superimposed on its surface. When the input object (pen or puck) touches the board, graphic information describing the location of the point touched is transmitted as input information. The data tablet is an absolute pointing device.

descriptor: A mechanism for passing parameters in which the address of a descriptor is provided in the longword argument list entry. The descriptor contains the address of the parameter, the data type, size, and additional information needed to describe fully the data passed.

device coordinates: The device-dependent Cartesian coordinates that specify positions on the VMS display screen. Sometimes referred to as physical device coordinates, these coordinates are involved in mapping of the display window to the display screen.

direct color value: Each pixel value directly specifies a color.

display viewport: The area of the physical display screen into which a display window is mapped. It is the physical region on the terminal screen that is created by the VMS workstation and controlled by the user.

display window: The portion of world coordinate space mapped to the graphics viewport. The display window is used to control how much of the virtual display is potentially available for the user to view.

emulated terminal: A virtual I/O device whose programming interface matches the programming interface of a specific physical terminal and whose appearance on the VMS workstation screen is similar to the appearance of the physical terminal.

exception condition: A hardware- or software-detected event that alters the normal flow of instruction execution.

font: A specific representation of a text character. The attributes of a font are family (type face), type size, and rendition.

function: A procedure that returns a single value according to standard conventions. If additional values are returned, they are returned by means of the argument list.

graphics data tablet: An optional input device that consists of a rigid tablet and a puck containing a crosshair cursor and a number of buttons, or a pen. The position of the cursor can be read by application programs. The tablet is an absolute pointing device.

graphics display: Describes any graphics data output device that can present an image of graphic data derived from a computer graphics system. An example of a graphics display is a display screen or a printer.

graphic object: The graphic image constructed by an application program using UIS routines. A graphic object could be a simple line or a complex drawing.

graphics text: Text output primitives displayed using the UIS routines.

grey scale: The level of brightness that describes the illumination of a cathode-ray tube screen.

HLS: Hue lightness saturation.

HSV: Hue saturation value.

image: The output form of on-line graphics data. That is, a displayed or drawn representation of a graphics file.

language-support procedures: Procedures called implicitly to implement high-level language constructs. They are not intended to be called explicitly from user programs.

library procedures: Procedures called explicitly using the equivalent of a CALL statement or function reference. They are usually language independent.

major path: Direction in which characters are drawn on a line.

mapped color value: Pixels that indirectly specify an active color value.

mapping: Any process by which a graphics system translates graphic data from one coordinate system into a form useful on another coordinate system.

minor path: Direction used for beginning a new line of text.

mouse: A data entry device consisting of a small control box, on rollers, that is pushed along a surface and transmits its changing position to the workstation. Often, function keys or buttons are mounted on the device and can be used to enter information or make selections. This device is the user's means for pointing to and selecting objects on the screen. The mouse is a relative pointing device.

output primitive: A part of an image created with UIS procedures, such as a graphics object or a text string, that has a specific appearance. Values of attributes determine some aspects of this appearance.

physical device coordinates: Device-dependent Cartesian coordinates that specify the addressable points on a physical device.

pixel: The density of one picture element. The smallest displayable unit on a display screen.

pointer: The cursor on the screen that tracks movements of the mouse. The shape of the pointer depends upon its current use.

primitives: The most basic graphic entities available on a graphics system, such as points, line segments, or characters.

procedure: A closed sequence of instructions that is entered from, and returns control to, the calling program.

puck: A hand-held graphics device with a cross hair sight used to pinpoint coordinates on a data tablet or digitizer.

raster: A pattern of scanning lines in a cathode-ray tube that divide the display area into addressable points.

reference: A mechanism for passing parameters in which the address of the parameter is provided in the longword argument list by the calling program.

relative pointing device: A pointing device that reports movement to the workstation based how far and in what direction the device is moved from one point to another.

resizing: The process of scaling or changing the size of a graphics viewport according to predetermined data.

RGB: Red Green Blue

rotation: A graphic object turning on an axis.

scaling: Proportional expansion or reduction of a graphic object on the screen.

stretchy box: The outline of a clipping rectangle used in the UIS functions PRINT SCREEN and RESIZE WINDOW. This rectangle can be manipulated to assume practically any rectangular dimensions and is limited only by the display screen size.

subroutine: A procedure that does not return a value according to the standard conventions. If values are returned, they are returned by the argument list.

text path: Direction of text drawing.

text slope: The angle between the actual path of text drawing and the major text path.

tablet: A device that can convert a stylus position into Cartesian coordinates. When connected to a graphic display screen, it can control the real-time positioning of a cursor or pointer.

transformations: The ability of the UIS graphics system to manipulate coordinate data. Transformations occur when mapping one coordinate system into another coordinate system.

translation: Defining the position of the graphic object in a coordinate system.

UIS: The graphics software called User Interface Services.

value: A mechanism for passing input parameters in which the actual value is provided in the longword argument list entry by the calling program.

viewport: A rectangle that maps the image defined by a window into a virtual display onto the display screen. The user controls the visibility and placement of viewports on the physical screen.

viewing transformation: The viewing transformation is the process of mapping the world coordinates of a graphic object in a display window to the device coordinates of a display viewport on a physical display device.

virtual color maps: Color maps swapped in and out of memory—this follows the same concepts as virtual memory.

virtual display: The world coordinate space defined by an application program. An application program uses a virtual display as a place in which to build graphic images. It can be thought of as a virtual output device that has the properties of a physical screen, but is not necessarily visible on a physical screen.

virtual keyboard: A virtual input device associated with a window. When users select a window into a virtual display with a virtual keyboard, input from the physical keyboard is directed to the virtual keyboard and can be read by an application program.

window: A defined area within a virtual display that can be used for viewing the virtual display. A window is the area of the virtual display that is to be mapped to a viewport.

world coordinates: Device-independent Cartesian coordinates defined by the application program in order to describe objects to UIS.

x axis: The reference line of a rectangular coordinate system used to determine horizontal distance and positions.

x-height: The height of lowercase characters excluding descenders and ascenders.

y axis: The reference line of a rectangular coordinate system used to determine vertical distance and positions.

zooming: The process by which the perspective on a displayed graphics file moves rapidly closer or farther from the operator.

Index

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

VWS Engineering
Digital Equipment Corporation
110 Spitbrook Rd. ZKO3-2/S30
Nashua, New Hampshire 03062-2698

Do Not Tear - Fold Here