A Guide to Migrating VWS Applications to DECwindows

Order Number: AA-MI67A-TE

This manual aids software developers and architects in migrating existing UIS applications to DECwindows format.

Operating System and Version: VMS V5.1 or later

Software Version: VWS V4.2

September 1989

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

This document was prepared using VAX DOCUMENT, Version 1.2

66314

Contents

TABLES

Preface

DECwindows is Digital's window based, distributed application environment. It is based on the industry standard X Window System™ Version 11, developed by Project Athena and the Laboratory for Computer Science at the Massachusetts Institute of Technology with funding and participation by Digital Equipment Corporation.

The X distributed windowing system provides a client-server model in which the server provides display and input services while the client provides the main application code. Because the system provides a message-based interface between client and server, the physical location of each can be on the same or different systems. Communications between the client and server can be implemented with any number of transports, including DECnet, TCP/IP, and shared memory. This model lends itself well to a heterogeneous local network environment where mixed hardware and software systems exist. The display from a VMS DECwindows application can be directed to PCs or workstations running VAX/VMS, MS-DOS™, or ULTRIX, and to hardware from a range of manufacturers.

DECwindows replaces the VWS windowing software for VAX/VMS. VWS is a tightly coupled, kernel-based windowing system for the VAXstation. Its procedural interface, the User Interface Services (UIS), provides a high-level programming interface to the graphic subsystem designed and optimized for VAX Workstations and the VMS Operating System. Because VWS is a kernel-based procedural interface rather than the message-based, client-server model of the X Window System, VWS does not address the issues of mixed hardware and software in local area networks or distributed and open computing.

DECwindows provides a further step in Digital's commitment to open systems and distributed heterogeneous computing environments. It provides a standard program interface across multiple hardware and software platforms while encouraging a common look and feel for all Digital applications. The DECwindows Applications Interface (API) has been accepted as the standard by the Open Software Foundation. Users and vendors can change the look and feel of the windowing system while application programs can run without modification.

NOTE:

- **MS-DOS is a trademark of MicroSoft.**
- **The X Window System is a trademark of MIT.**

Intended Audience

This document provides information to aid software developers and architects in migrating existing UIS applications to DECwindows. The intended audience of this document is the application developer who has written UIS applications and who is preparing to move this application to the DECwindows platform.

This document gives some general advice on how to evaluate the application port. It provides pointers to the DECwindows documentation you need to perform the migration.

Document Structure

This document consists of six chapters and six appendixes that contain the following information:

- Chapter 1: Comparative Overview of VWS and DECwindows
- Chapter 2: Getting Started
- Chapter 3: The DECwindows Look and Feel
- Chapter 4: Using the Toolkit
- Chapter 5: User Interface Language (UIL)
- Chapter 6: Resource Management
- Appendix A: Overview of VMS DECwindows Documentation
- Appendix B: UIS$ Routine Reference
- Appendix C: UISDC$ Routine Reference
- Appendix D: HCUIS$ Routine Reference
- Appendix E: UIS Fonts to DECwindow Equivalents
- Appendix F: Color Conversion Routines
- Appendix G: Colormap Example
- Appendix H: Mapping UIS Writing Modes to X11 Attributes

Documentation Standards

- Although VWS refers to the entire windowing system and UIS refers to the runtime library interface, you can use VWS and UIS incterchangeably.
- The terms X and X11 represent the X Window System, Version 11. You can use them interchangeably with DECwindows if the functionality being described is the same.

1 Comparative Overview of VWS and DECwindows

This chapter compares VWS and DECwindows. Use this chapter to help plan a strategy for migrating existing UIS applications to DECwindows/X11. This chapter covers the following aspects of the window systems:

- Architecture
- Coordinate systems
- Windows
- Graphics output
- Color
- Virtual displays and display lists
- Fonts
- Input
- Window manipulation
- Data association
- Terminal emulation

1.1 Architectures

Both VWS and X11 provide window functions. The major differences between VWS and X11 result from their respective design philosophies.

- VWS is designed as a "complete" graphics and window system.
- X11 is designed as a low-level graphics and window system in which high-level features such as display lists, virtual displays, backing store, and world coordinates must be implemented as user libraries. Some of these higher-level layers are part of the base DECwindows software, and some are part of layered applications such as GKS and PHIGS. In addition, X11 borrows heavily from its original UNIX®[1] background, and thus the approach to application design and responsibility is very different.

Therefore, VWS applications that take advantage of the built-in, high-level features of UIS or of VMS-specific features such as ASTs might be difficult to port.

[1] UNIX is a registered trademark of American Telephone & Telegraph.

1.1.1 Significant Design Differences between UIS and X11

UIS is designed as a procedural, *kernel-based* window system, while X11 is designed as a message-passing window system. UIS interface design presupposes that the display hardware exists on the same system on which the application executes. In contrast, X11 is a client/server design that relies on message-passing of graphics and input data between the application (referred to as the *client*) and the graphics subsystem (referred to as the *server*). The separation of the window system from the application by means of a communication link lends itself to implementations using high speed local area networks (LANs) as the medium for the message exchange. By allowing this separation, clients can execute on nodes other than the workstation, taking advantage of special or more powerful hardware than the workstation itself.

Because of this separation of application and graphics system by a communications medium, the hardware and operating system of both client and server become irrelevant. Much like an asynchronous terminal connected to a computer, as long as they both use X11 and the same transport medium, any X11 client can use any X11 server. A DECwindows application makes window system calls without concern for where the output is presented or where the application is executing. Output and input pass between the DECwindows Client application and the X11 Server application, which executes on a workstation. The transport mechanism can be any suitable hardware and protocol including Ethernet, token ring, or shared memory; the protocol might be DECnet, TCP/IP, or any other suitable communication protocol.

Because building and decoding protocol message packets is a difficult task, X11 provides a procedural interface that builds and decodes the message packets for the application while managing the communication link. This procedural interface is called "Xlib."

NOTE: Digital discourages direct communication at the X11 "wire" protocol level.

Although X11 supplies a rich set of input and output capabilities, it was designed to be device-independent. Some graphics hardware provides highly specialized features that X11 cannot utilize. However, X11 has an extension mechanism that enables application developers to enhance the window system to take advantage of these unique features.

X11 does not protect windows against modification by other applications, nor does it hide the window system workings from the application. Think of X11 applications as extensions to the window systems, while UIS applications deal with a "virtual workstation" where each application believes there are no other users of the graphic hardware.

The combination of the network-based, client-server design, a rich set of low-level graphic routines, and fewer restrictions on what applications can access (that is, other windows not created by the application) can provide much more powerful (and complex) capabilities than are available under UIS. On the other hand, UIS is more approachable for the novice and protects each application from other windowing operations.

1.2 Coordinate Systems

When an application performs output to a window, it must specify the (x,y) location in the window where the output is to be drawn. The interpretation of the (x,y) pair is determined by the window coordinate system.

The lowest-level coordinate system that is supported by both UIS and X11 interprets (x,y) pairs as pixel locations. In both systems, the pixel locations are relative to the origin of the window. The origin of an X11 window is the upper left-hand corner with the y-axis increasing downwards (this is a natural view of a raster device). In contrast, the origin of a UIS window is the lower left-hand corner with the y-axis increasing upwards (this is a natural view as the upper-right quadrant of a Cartesian coordinate system). In both systems, the x-axis increases from left to right.

Inversion of the y-axis is a simple operation that requires subtracting the coordinate from the maximum y value for the window.

UIS also provides a world coordinate system. In this system, the application defines any convenient unit of measure such as inches, miles or microns. Thus, the application can define its coordinates to represent inches, kilometers, seconds, and so on, rather than pixels. To perform output, the window system must convert from world coordinates to device coordinates. In addition, window size and location are specified in centimeters and are thus screen size/resolution-independent. X requires window size and location to be specified in pixels.

Since X11 does not support the use of a world coordinate system, an application that requires this capability must provide its own routines to perform world-to-device coordinate conversion.

To translate world coordinates, use the maximum world coordinate value for the window and the maximum device (pixel) coordinate value for the window to transform the current world coordinate into a device (pixel) coordinate as follows:

```
( DEVICE_MAXIMUM / WORLD_MAXIMUM ) * CURRENT_WORLD
```

1.3 Windows

To perform any input or output on the workstation, an application must first create a window on the display. This window defines the region into which the application can perform output. The window system ensures that no output extends beyond the boundaries of the window. This is known as *clipping*. Both UIS and X11 clip to the window boundaries in the same manner.

Before it creates a window in X11, the application must receive permission from the X11 server on the desired workstation. To provide system security, each X11 server determines which clients are permitted access to its resources. This is implementation-dependent but is implemented on all Digital X11 servers.

All X11 windows are arranged in a hierarchy, or tree structure, of arbitrary depth. The entire surface of the screen is covered by a single window called the *root* window. All other windows are descendants of this root window. When a window is created, its *parent* window must be specified. If this new window (known as a *child*) extends beyond the boundaries of the parent, the parent clips it. Thus, no window can output outside the boundaries of any of its ancestors. A window can have multiple children, but a child has only a single parent.

Normally, an X11 window has a solid background that causes it to obscure any window it overlaps. However, a window can be defined without a background, which makes it transparent. Among other things, such windows facilitate temporary overlays like gridlines.

NOTE: **Transparent windows are not possible in UIS.**

When any portion of an X11 window that was obscured by another window becomes visible, the application must be capable of recreating what should be displayed in the newly exposed area. In contrast, UIS automatically restores exposed areas so the application need not be responsible for refreshing the window. This is known as *backing store.* Backing store is guaranteed under VWS. Under X11, backing store can be provided as an optional feature, but it is not a required function. Backing store can be provided at the server's option and applications should not depend on it.

UIS and X11 use windows in different ways.

- On UIS, a window is an onscreen representation of a *virtual display.* The contents of the virtual display and any window associated with the viewport are guaranteed to be preserved. A virtual display gives each application the illusion that the application has exclusive use of the workstation.
- On X11, there is no equivalent concept. Under X11, windows simply represent a data structure to the X11 Server, and when *mapped,* they also represent a region on the display to which graphics operations can write.

NOTE: **UIS windows require much more overhead than X11 windows. Thus, X11 relies on windows much more than UIS. X11 uses windows for buttons (usually gadgets without a window), hotspots, and other functions rather than simply as output surfaces.**

1.3.1 Graphics Output

Window systems offer a variety of methods to create graphics output. Most systems provide a number of *primitives* to draw lines, polygons, and text. All output primitives have various characteristics that affect their appearance. Some window systems also furnish other primitives such as arcs. In addition, applications can group a set of polygons together into a single entity known as a *region.* Other functions include the ability to manipulate arbitrary rectangular areas. The following sections explore these topics in further detail.

1.3.1.1 Lines and Polylines

Both UIS and X11 provide routines to draw a single line, multiple disjoint lines, and multiple connected lines. You can draw these lines with any specified width, line style, and color. In X11, when you draw a sequence of wide, connected lines, you can use a number of methods to join them smoothly at their endpoints. X11 enables the application to choose from a number of joint and cap styles. UIS provides a single method for joining and ending lines.

1.3.1.2 Writing Modes

UIS writing modes provide a high-level means of specifying combinations of specific attributes. Writing modes affect the appearance of graphic objects at areas where they intersect.

In X11, writing modes are a combination of *functions* and *fill style*. An application specifies each attribute separately. In X11 you must set the following attributes:

- Function

  The function describes the logical operation that is performed. For example, GXor indicates the source is logically ORed with the bitmap.

- Background pixel

  The background pixel is typically drawn in replace-style operations for off pixels.

- Foreground pixel

  The foreground pixel typically specifies the source for the logical operation.

- Stipple pattern

  The stipple pattern is generally a bitmap that is used for fill operations.

- Fill style

  The fill style specifies the type of fill, such as solid, stippled, or stippled-opaque. A stippled-opaque operation sets the off pixels in the pattern to the background pixel, while a stippled fill does not change the off pixels in a pattern. All three set the on pixels in the pattern to the foreground.

Appendix G shows the mapping of UIS writing modes to X11 attributes.

Some of these operations might not work as expected: for example, some are documented as device-dependent and will have different results based on the location of the pixels in the colormap. To ensure that all writing modes work correctly all the time, you must create a private application colormap in which all pixel values are mapped with a starting index of zero. If you follow the directions provided for creating colormaps, the device-independent writing modes should always work as expected.

NOTE: Creating a private colormap can cause repercussions. The colors for additional windows are not guaranteed.

1.3.1.3 Colormaps

A significant portion of the cost of graphics systems can be consumed by the memory used to store the pixel values. Limiting the number of planes is one means of reducing the cost of the system. Unfortunately, this affects the number of colors that can be displayed. However, the impact of this limitation can be reduced through the use of a *colormap.* You use a colormap to convert a pixel value to a color on the display. Such a system is called a *pseudocolor* display.

An additional benefit of using a colormap is that it can be updated by the application at any time. This feature is necessary to implement a number of algorithms in fields such as image-processing and computer-aided design (CAD).

UIS and X11 handle colormaps dissimilarly. The UIS model is to create and manage *virtual* colormaps that are portions of the hardware colormap. This allows the sharing of the hardware by allocating part (or all) of the hardware colormap for exclusive (or shared) use by an application.

The X11 color model is designed for a wider variety of display hardware and supports the direct use and emulation of the following types of systems:

- Monochrome (bitonal)—Black and white.
- Pseudocolor
- Direct color—Each pixel value is composed of the actual red, green, and blue values sent to the video, rather than an index into an array of RGB triplets used by pseudocolor. X uses colormaps for direct color. Only the R, G, and B values are treated as three separate indexes.

Generally, X11 supports a single installed (virtual) colormap, shared by all applications that request the allocation of contiguous or disjoint individual colors for either shared or exclusive use. The preferred method of colormap usage by X11 is the request of "named" colors, such as "RED," "Red," or "red" (it is case sensitive). X11 applications can emulate many UIS colormap concepts directly, but these functions might not fit into preferred DECwindows usage.

Because UIS virtual colormaps are allocated to guarantee the ability to perform logical operations on the pixel values, X11 users must perform the following specific set of operations to emulate a UIS colormap.

1. Round up the number of colors needed to the power of 2, and determine the number of planes required (see Appendix F).
2. Using the Allocate Color Cells call, request the number of planes found in step one and a single color cell.
3. Using the array of plane masks and the pixel value returned, create an array of indexes of all the permutations of the plane masks and pixels.
4. Create an XOR mask by the logical OR of all of the plane mask bits.

Appendix E contains an example code fragment that shows how this can be accomplished. All logical operations that can be performed in UIS can then be performed with the values in the array and/or the plane mask bits.

Commonly, you specify colors with one of three models: RGB, HLS, or HSV. UIS supplies routines to convert color specifications between systems. X11 does not directly support the HLS and HSV color models. Appendix E contains example conversion routines.

1.3.1.4 Plane Masks

Lines are drawn as a sequence of pixels. A pixel can consist of multiple bits to produce colors or shades of gray. The number of planes in the system determines the number of bits per pixel. Each bit within a pixel resides on a separate plane of display memory. A plane contains all the bits from the same bit position in every pixel. Thus, if there are four planes in a system, each pixel value is determined by one bit from the same location in each of the four planes.

Both UIS and X11 provide a means to limit the planes that can be modified by a drawing command via a plane mask. The X11 mechanism is somewhat more complex due to its generalized nature of color support.

1.3.1.5 Polygons

UIS and X11 provide methods for drawing polygons as well as lines. All output characteristics for lines are available for polygons. Polygons can optionally be filled. In UIS, fill patterns must be chosen from a character font. In X11, the application has the flexibility to define the fill patterns, although no predefined patterns are supplied. X11 also provides the ability to perform filling through a stencil. Such stencils are called stipple patterns. Fill patterns are also called stipples. The difference is the fill style used.

1.3.1.6 Arcs

Another primitive supported in both UIS and X11 is an arc drawing primitive. All the aforementioned output characteristics are available for arcs. In UIS, the endpoints of an arc can optionally be joined with a chord or to form a pie slice. In X11, this option is only provided for filled arcs.

1.3.1.7 Attribute Blocks and Graphic Contexts

All possible output characteristics are grouped into structures. UIS calls these structures attribute blocks (ATBs) and X11 calls these structures graphics contexts (GCs). In both systems, the applications must specify the structure to use when drawing a primitive. UIS provides a means of querying the characteristics in an attribute block; X11 does not provide such a method.

1.3.1.8 Text

In a window system, text is also a graphics primitive. Both UIS and X11 support a large variety of fonts. UIS provides a number of text attributes that are not available in X11. These attributes include the following:

- Text path
- Slope
- Character slant (sheeting)
- Scaling

- Rotation
- Formatting (text margins, centering, and so on)
- Character spacing

UIS can also maintain the notion of current text position for each window to allow automatic formatting of blocks of text in separate text calls.

1.3.1.9 Regions

It is often convenient to refer to one or more polygons as a single entity. X11 enables applications to create and manipulate these entities known as regions. These regions can be copied, moved, shrunk, and expanded. New regions can be created from the intersection, union, subtraction, and XOR of two other regions. Regions can also be compared. X11 provides routines to determine if an arbitrary point or rectangle is within a given region.

NOTE: **UIS does not incorporate the concept of regions.**

1.3.1.10 Direct Manipulation Of Pixels

Both UIS and X11 empower the application to manipulate rectangular areas of pixels within a window. Arbitrary areas can be moved to any window location. Images can be read from a window into the application memory or, conversely, written from memory into a window. This differs from the functionality of UIS, however, because X11 does not guarantee the contents of a window; also, it is important to ensure that the window contents are valid when read under X11, because window occlusion destroys the window contents.

1.3.1.11 Images

Both UIS and DECwindows provide the capability to write image data to the display. In UIS, you can also scale images by integral multiples. In UIS, you can write image data "as is" into the display because UIS allocates virtual colormaps as follows:

- The base index in the hardware colormap of the virtual colormap contains zeros.
- The high-order bits are an offset.

In X11, you can accomplish this only by creating a private colormap and allocating the colors manually.

NOTE: **Creating a private colormap can cause repercussions. The colors for additional windows are not guaranteed.**

When you allocate colors by using the method described here, the image data must be translated from the image index data into actual pixel values. If you allocate the planes by using the "contiguous" flag, you might simply have to shift the data. Otherwise, you must use a translation table to look up the data.

1.3.2 Virtual Displays and Display Lists

The VWS concept of a virtual display is an imaginary surface onto which objects can be drawn. The virtual display is defined by a range of values in the world coordinate system. The entire virtual display or any subset can be displayed in a window. Multiple windows can share the same virtual display.

A display list is a hierarchical tree of objects in a virtual display. VWS converts all world coordinates into a *normalized device space* ranging from 0 to 1.0. These device-independent, normalized coordinates can then be mapped to any window. An application can scale, translate, rotate, or edit objects in the display list. A display list can be saved into a disk file called a *metafile* for storage, printing, or later retrieval and reexecution into a UIS window.

Since X11 does not support virtual displays or display lists, an application that requires these capabilities must provide its own set of display list routines or use a higher level interface such as GKS or PHIGS.

1.3.3 Fonts

UIS uses the Common Font File Format (CFFF) for the source format of fonts. It then compiles this into formats optimized for color or bitonal systems.

DECwindows uses the X Logical Font Description (XLFD) or Binary Distribution Format (BDF) for its source and compiles this into a Server Natural Format (SNF) file. In addition, DECwindows can contain font metric files that provide information on print-spacing that is not available in the UIS-compiled font files.

1.3.4 Input

One of the responsibilities of a window system is to receive input from workstation devices and deliver the events to the proper processes.

X11 provides input events to applications by sending an event as a message to the client application. The client application transport logic inserts the X11 event message into a queue of events to be processed. This enables the application to determine when input is to be processed by periodically checking its event queue. In contrast, UIS uses the VMS AST mechanism to interrupt the application's user execution and start a concurrent thread of execution to handle the input. The input can be queued for later processing by the user mode thread of execution, or the application might be completely event-driven and designed so that all execution is at the AST level. DECwindows can provide a *doorbell* AST for VMS that is a general notification of an event.

This difference leads to an *event processing loop* design for DECwindows application and *event driven* design for UIS. Because of this major difference, the structure of an application can change significantly.

Both UIS and X11 enable applications to specify which event types they want to receive. Event types include mouse button change, mouse movement, keyboard key pressed, and so on. Because a single queue is used for all events, X11 guarantees that the events will be delivered in the same order in which they occurred. In addition, since X11 uses a queue, the application can extract events from the queue in any order. Applications *must* service the input queue in a timely manner. If the server cannot obtain a free input packet, it breaks the connection between itself and the application. Long computational loops should be broken up to scan the input queue periodically, or the AST doorbell notification should be used to remove the input events from the X11 queue onto a private application queue to prevent the loss of the connection (note that this is not always possible when you are using the X11 or DECwindows toolkit).

1.3.5 Window Manipulation

Workstation users can use a window manager to manipulate windows on the display. Window manipulations in both UIS and X11 include the following functions:

- Move
- Resize
- Pop (raise)
- Push (lower)
- Iconify
- De-iconify

In addition, the window manager in X11 enables you to change the focus of keyboard input to any window. This is similar to attaching the keyboard to a window in UIS.

Because VWS was written as a VMS-specific window system, VWS chose to reserve the first 5 function keys on the LK201 for VWS specific functions. Because X11 is not bound to the LK201 for its design, the function keys are used as normal application keys. The VMS DECwindows manager does not provide the cycle key (F5) to allow changing of input focus, nor is a Hold Screen function provided. To change input focus under the DECwindows/VMS window manager, click on the window or title bar with the mouse. The operator console can be toggled by F2 under VWS and CONTROL-F2 under the DECwindows window manager.

1.3.6 Data Association (Context Management)

X11 applications often use a large number of windows because of the relatively low overhead associated with them. X11 provides routines to assist the application in managing many windows. These routines associate window system information with the application's own information. For instance, they can be used to help the application determine what action to take for a given input event.

1.3.7 Terminal Emulation

In many respects, UIS is "VAX/VMS with graphics." UIS does not provide an object-oriented window system and programming toolkit and many users still rely heavily on terminals for applications and development. "Terminals" on UIS and on DECwindows are provided in the form of "terminal emulation windows" that appear to the VMS system as physical terminal devices connected to the workstation. Many UIS workstations are used primarily as multisession terminals.

VWS provides three terminal emulators:

- An enhanced VT200 emulator that provides the following functionality:
  - ReGIS and SIXEL graphics with from 2 to 256 colors.
  - ANSI color text for color workstations.
  - Control over a number of font styles and sizes.
  - Escape sequences for manipulating the terminal window.
  - Icon and escape sequence reports and terminal mailbox messages for notification of window system events (moving the window, resizing, and so on).
- A Tektronix® 4010/14 emulator that provides simple monochrome emulation of the Tektronix vector terminal.
- A Tektronix® 4125 color emulator that provides a more powerful vector terminal emulation with display list support and most functions provided by the actual hardware.

DECwindows provides a VT340 compatible terminal (DECterm) that offers both ReGIS and SIXEL graphics with 16 colors. Two font sizes are available for terminal windows and the last 100 lines scrolled off the top of the terminal can be reviewed.

Both emulators provide copy and paste functionality between terminal windows (DECwindows can also copy and paste between non-terminal windows) as well as variable geometry.

DECterm and the VWS 4125 emulator are implemented using a psuedo-terminal interface, and they both execute as normal user processes. The VWS VT200 and TEK4014 emulators are tightly coupled to the VMS terminal Port/Class driver architecture and execute in kernel mode as part of the VMS system software. They provide high performance with very little overhead.

1.3.8 Conclusion

Table 1-1 summarizes the issues discussed here. The table lists features specific to UIS not implemented in DECwindows. UIS features are listed with a rating estimating the amount of effort required to implement the given feature in X11.

Table 1–1 UIS Features not Implemented in DECwindows

Features	Difficulty[1]
Asynchronous input	2
Attribute query	1
Color segmentation	2
Color system conversion	1
Display lists	3
Metafiles	3
Geometry transformation	3
Predefined fill patterns	1
Text attributes	3
Virtual displays	3
Window damage	3
World coordinates	2
Access from kernel mode	Not possible
Direct hardware access	Not possible

[1]Difficulty ratings estimate the amount of effort required to implement this feature in X11. The ratings range from one to three, with one requiring the least amount of effort.

2 Getting Started

This chapter provides a guide for evaluating your existing application and designing a strategy to port it to DECwindows. Be aware that there is no easy way to convert a UIS application to DECwindows unless the application was designed for ease of portability. In many cases, you will have to redesign and rewrite your applications. The level of difficulty depends on how closely the application is designed around the graphic interface.

2.1 Migrating a UIS Application to a DECwindows Application

2.1.1 Writing a DECwindows Application

Before you try to determine how to port a UIS application, you should learn how to write a DECwindows application. The DECwindows installation provides a variety of example applications that can be modified and used to understand the basic concepts of programming for DECwindows. Digital's Educational Services and Software Services also offer courses, consulting, and individual training for DECwindows programming. It is important to understand the DECwindows programming and design philosophy before you attempt to port an application.

2.1.1.1 Sample Xlib Application Port

A sample UIS application is included with the VWS Version 4.2 distribution. This is a simple, partially functional graphics editor called "FREDIT" (frame editor). This application has been adapted for use under X11 by a combination of UIS "emulation" and application code changes.

The module FREDIT$X11_UIS_EMULATION.C illustrates what UIS functions can be accomplished in X11. It also points out differences and possible problems. See the *VMS Workstation Software Version 4.2 Release Notes* for information on installing this sample application. This release includes the following modules:

- Application source code (in VAX C)
- Application object code
- Application executables

2.1.2 Differences Between UIS and DECwindows

VWS and DECwindows are very different. VWS provides the following design functions:

- Several levels of interface to the graphic hardware from a high-level, floating point, world coordinate interface, with display list capabilities, backing store, and metafile support.
- Integer (pixel) coordinate device-dependent interface.

- Low-level, direct interface to the graphic hardware through the QIO and drawing operation queues via the windowing system.

At all levels, UIS protects the user from the effects of other window applications. In the X11 Window System, applications have the following characteristics:

- They are extensions of the windowing system.
- They are responsible for the maintenance and integrity of their own windows.
- They can modify or destroy data in any other windows on the display.

X11 provides access to graphic windows at a level comparable to the UIS device-dependent (UISDC$) level and provides a rich set of drawing primitives, many of which are not provided by UIS. The X11 Window System approach to program flow is derived from its UNIX origins. Generally, X11 programs are written as event-dispatch loops, and only minimal AST support is provided.

NOTE: **AST is a Digital-only extension. It is not part of the X11 standard.**

Converting a UIS application to DECwindows always requires some amount of recoding, depending on what the application does. DECwindows does not support the following functionality:

- Guaranteed backing store (this enables UIS to maintain the contents of each window without user intervention)
- Display lists
- Scaled or rotated text
- Colormap segments
- QIO or DOP interface
- Direct hardcopy (HCUIS)
- World coordinate systems
- Multiple views and window zooming

If your application uses the UIS device-dependent (UISDC$) routines, it is easier to translate your application to DECwindows. In most respects, you can consider UISDC$ drawing routines (excluding the DOP routines) as a subset of the X11 graphic capabilities. (Text attributes, which do not exist under X, are a major exception to this statement.) Converting higher-level UIS$ calls is more challenging, since X11 does not directly support viewports, world coordinates, and display lists. In both cases, the immediate challenge is the application's ability to regenerate any area of the screen on request. Under VWS, the integrity of all windows is guaranteed by the windowing system. Under DECwindows, in an occluded window (one that is covered by another window), the contents of the occluded area are destroyed; when the window is later uncovered (exposed), the X server requests the application to redraw the area's contents.

Porting an application can be an enormous challenge, but if you consider the port in terms of implementing the existing functionality rather than translating the existing code, DECwindows provides several functions that can make the job easier:

- If the application does very little actual graphics but interacts primarily with the use of menus, push buttons, dialogue areas, and simple text, you might be able to use the User Interface Language (UIL) and the DECwindows Toolkit routines with very little programming to replace much of the UIS code.
- If the application does extensive graphics and relies on display lists, you might be able to rewrite it by using one of the industry standard graphic interfaces such as GKS or PHIGS. It is important to know whether the application requires writing to the lowest level of the graphic subsystem (X11). Because the GKS and PHIGS interfaces are designed to be portable across hardware and base software platforms, code should run unchanged across both VWS and DECwindows. The performance of GKS and PHIGS is adequate for most user applications, while providing device- and graphic-subsystem-independence.
- The DECwindows toolkit provides many commonly used operations, such as menus, text input, and the like. These might not need to be coded by the application.

If your application does extensive low-level graphics or requires higher than average performance, you might have to reimplement the code directly in the following ways:

- By calling the Xlib procedural interface.
- By using the DECwindows toolkit routines to provide the standard menu interface and "look and feel."

NOTE: **Generally, DECwindows Version 1.0 applications require more memory than UIS applications. This is true partly because of the extensive library routines DECwindows provides for standard user interface objects such as menus, forms, and dialog areas. Also, the Xlib interface requires virtual address space to provide the procedural interface to the byte stream messaging protocol.**

Users with small memory systems (4mb) should consider increasing the amount of physical memory according to the guidelines provided in the *DECwindows Installation Guide*.

2.2 Tools that Get You from Here to There

You can install both DECwindows and VWS on the same VAXcluster or standalone VAX system. Set the WINDOW_SYSTEM SYSGEN parameter to 1 to start DECwindows at system boot; set it to 2 to start VWS. (If you set the WINDOW_SYSTEM SYSGEN parameter to 0, you disable any windowing system.)

As part of the overall Digital strategy to provide tools to assist in the migration to DECwindows, a DECwindows X11 Server has been created to enable DECwindows to run as a UIS application. When you use the DECwindows X11 server, DECwindows and X11 applications use a single UIS window as a *virtual workstation*.

The server is started during system boot by the standard DECwindows startup procedures (or at any time from the SYSTEM account). This server emulates a VS2000 monochrome workstation. Although it provides relatively low performance, it offers an exact emulation of DECwindows output (it is in fact a real DECwindows component). You can write, compile, and test DECwindows code under this server, and only when production testing is required must the workstation be rebooted. This server is included on the VWS Version 4.2 kit and will run on any UIS workstation running VMS Version 5.1 or later.

To help modify your UIS data files into a format that can be used by DECwindows applications, a .UIS to .DDIF conversion library is provided as part of the CONVERT utility.

NOTE: UIS (User Interface Services) defines a file format for the storage and retrieval of picture information. DDIF (Digital Document Interchange Format) can be used by any Digital product that interprets it. It provides users with a means of porting software from one format to another. For more information about the CONVERT utility and DDIF, see the *UIS to DDIF Converter Installation and User's Guide*.

2.3 Xlib, the Toolkit and Widgets

X11 is based on a byte-stream protocol that is generally not accessible to most applications. In fact, direct output of the X11 *wire protocol* is actively discouraged by Digital. Access to the graphic system is provided by a procedural interface (Xlib) that generates wire protocol messages. In addition to converting procedural calls into protocol commands, Xlib also provides buffering services and can compress many calls into single X11 protocol requests.

The DECwindows Toolkit provides a set of higher level routines that are not comparable to any library routines provided by VWS. These library routines offer a standardized method for interaction with the user.

The term *widget* describes *window object*. Widgets are created and managed by the toolkit library. The toolkit implements widgets as semi-independent windows providing input and output interaction to the user for such things as menus, text messages, text input, and dialogue boxes. A *gadget* is a widget that has no window. A gadget is managed by the toolkit as a part of a user window. Typically, gadgets are used as buttons or *hot spots* that are part of the main user display window. Gadgets are less costly (in terms of memory and performance) than widgets.

2.3.1 Xlib

Xlib programming differs greatly from UIS programming. UIS programmers should become familiar with the following characteristics of Xlib programming:

- The typical Xlib program is written as an event-dispatch loop, as opposed to the typical UIS application that would rely instead on AST-processing routines.
- UIS provides 256 attribute blocks (ATBs) per segment (with default 0), with inquiry routines for each attribute in an ATB. Xlib provides an unlimited number of Graphic Contexts (GCs) with no ability to inquire the settings.
- UIS relies on virtual colormaps for each viewport, while Xlib prefers that applications share the colormap, avoid installing private colormaps, and request specific colors such as "red" or "Red."
- Xlib programmers must ensure that they can recreate the contents of the window, while the contents of the display is guaranteed for UIS programmers.
- Xlib applications must service their input event queue in a timely manner or the connection to the server will be broken. Lengthy computational loops must be interrupted to keep the queue clear, or a secondary queue must be implemented within the application.
- X11 uses windows, widgets, and gadgets to accomplish things done by means of extensive user-coding under UIS. The use of regions and child windows in X11 can greatly simplify coding.
- There is no native X11 metafile format. Instead, Digital has implemented a Compound Document Architecture using DDIF as an extension to RMS to describe a variety of textual and graphic information.
  - Access to reading and writing files in DDIF format has been simplified by the CDA toolkit, which provides prepackaged library routines for most common input and output.
  - Applications are encouraged to use DDIF as their metafile data format.
  - The CONVERT utility has been provided to allow DDIF encoded files to be converted to a number of other data types such as PostScript™[1]. Also, VMS utilities will be able to display such files. The application programmer must determine how to represent the drawing information as both X11 and DDIF output.
- No standard display list library exists for X11. Either use a higher level interface such as GKS or PHIGS, or write your own display list management routines.

[1] PostScript is a trademark of Adobe Systems, Inc.

The Xlib interface is described in the *VMS DECwindows Guide to Xlib Programming: VAX Binding*. (The MIT: C language binding documentation is also available.)

A typical Xlib-based program would be structured as follows:

```
Start:

        Initialization

Main:
        Check for Event or Wait
        Dispatch Event
        Loop to Main

Event1:
        ...

Event2:
        ...

End:
```

A typical UIS-based program might be structured as follows:

```
Start:

        Initialization
        Hibernate

End:

AST Routines:

Event1:
        ...

Event2:
        ...
```

2.3.2 Window Contents

When you use Xlib, you must first decide how to maintain an internal representation of the application window.

NOTE: **This is generally an application-specific area, and only limited advice can be given.**

2.3.2.1 Regeneration from Data

Normally, native X11 applications can regenerate a portion of the window only if the application is structured such that the window contents can be reconstructed simply or computed from an internal representation. This might entail the recomputation of a spreadsheet or the redrawing of a text array.

2.3.2.2 Display Lists

You use display lists to maintain a representation of the data drawn to the screen. Many graphic applications already use some form of display list or can be adapted to use one. DECwindows provides no general display list library, so you must design and implement this logic for each application.

Display lists are usually implemented as a tree structure or linked list of *objects*. Each object contains the information necessary to draw some discrete portion of the display. The object might consist of X11 drawing commands or internal identifiers such as *pie chart* and the like. Display lists might also suffer from unbounded memory requirements, since complex drawings can require very large numbers of objects. Be sure each display list element has an extent rectangle.

2.3.2.3 PIXMAPS

A PIXMAP is an offscreen bitmap that can be created by X11. PIXMAPs can be written to and read from; they act like windows. An application can implement *poor man's backing store* by allocating a PIXMAP the same size as the user window and doing all writes twice—once to the window and once to the PIXMAP. On a graphic exposure, a bitmap-to-bitmap copy from the PIXMAP to the window restores the area. PIXMAPs are a limited resource and you cannot be guaranteed of allocating one. The use of PIXMAPS for this purpose is an abusive practice that increases the server resource demands and X11 protocol traffic between the client and server.

3 The DECwindows Look and Feel

3.1 DECwindows Benefits

DECwindows provides several benefits, which include both a standard application programming interface (API) and a consistent look and feel to applications running in the DECwindows environment.

The DECwindows toolkit and the DECwindows Window Manager provide much of an application's look and feel transparently to the programmer.

NOTE: **How the application interacts with the user and the system also should also be consistent. This is the programmer's responsibility.**

3.2 DECwindows Style Guide

To promote a common feel to both Digital and customer-written applications, DECwindows documentation comes with the *XUI Style Guide*. The style guide contains the following information:

- An overall statement of the philosophy of DECwindows.
- Specifics as to how applications should respond, look, and be organized to provide an integrated look to the DECwindows environment.

Study the *XUI Style Guide* and use it as a reference when you design the layout and behavior of a DECwindows application. UIS programmers will find that unlike the relatively unstructured and simple human interface of UIS (which can also lead to complications), DECwindows is very firm about how application programs should look and behave.

4 DECwindows Toolkit

4.1 Using the DECwindows Toolkit

The DECwindows toolkit consists of a set of runtime programming routines for building application interfaces that look and feel well integrated into the DECwindows environment.

The toolkit provides the following routines:

- Intrinsic Routines—Enable the programmer to manage a wide variety of toolkit components.
- Low-Level Widget-Creation Routines—Provide the most general interface for creating widgets. By using low-level routines, you can customize the widgets you create. These routines give the programmer control over all the widget attributes.
- High-Level Widget-Creation Routines—Provide the programmer with ease in creating and manipulating widgets. These routines allow access only to the most common attributes and provide mechanisms to obtain information about existing widgets. These routines simplify the creation and management of widgets that conform to DECwindows standards.
- Copy and Paste Routines—Provide access to the toolkit clipboard facility and enable transfer of both text and graphics between applications.
- Digital Resource Manager (DRM) Routines—Provide a high-level interface for applications using the output from the User Interface Language (UIL) compiler.
- Gadget-Creation Routines—Provide the most general interface for creating gadgets. Gadgets are simple, low-cost, windowless widgets. These routines are similar to the low-level widget routines.
- Compound String Routines—Enable the creation and manipulation of compound strings and font lists.
- Convenience Routines—Provide common functions that relieve the application programmer from "reinventing the wheel".

The toolkit and its use are described in the *VMS DECwindows Toolkit Reference Manual.*

4.2 Copy and Paste

DECwindows provides both application-to-application copy and paste, and a clipboard that can be copied from or pasted to. You can do a cut as well as a copy. You can transfer any type of data, including text and graphics. DECwindows provides data management as well as data transfer, but it is up to the application to decode the type of data passed. A common

selection "style" has been developed. The *Guide to Application Programming* and the *XUI Style Guide* provide guidelines and examples.

UIS does not provide user applications with any copy and paste facilities.

4.3 Compound Strings

DECwindows defines a new type of string called a *compound string*. This string can have the following features:

- Multiple fonts.
- Multiple character sets.
- Multiple font styles.

UIS does not provide this capability. You would need independent text strings to provide it in UIS.

You use compound strings with the Toolkit. The *DECwindows Guide to Application Programming* gives a description of compound strings and their use.

4.4 Widgets

The toolkit includes the following set of user interface objects:

- Menus
- Push buttons
- Scroll bars
- Text widgets
- Boxes

These objects, used as building blocks for DECwindows applications, are called *widgets*.

Widgets compose an input/output window that can display text and graphics. The predefined widgets included as part of the Toolkit can significantly reduce the amount of work required to implement many common programming tasks.

NOTE: **The predefined widgets do not provide any graphics drawing. You must use Xlib calls to draw graphics.**

For a more detailed description of widgets, see the *DECwindows Guide to Application Programming*.

4.4.1 Gadgets

Because maintaining a window and a separate context for each widget involves some significant overhead, windowless widgets, called *gadgets*, are available. A gadget has no window. Rather, it is associated with an existing window. Thus, a collection of gadgets can all share a single window context

rather than requiring a hierarchical structure of parent and child windows for the widgets.

NOTE: **Use gadgets the same way you use widgets.**

4.5 Intrinsic Routines

The X Toolkit routines, called *intrinsics*, enable you to manipulate widgets at runtime. Intrinsics perform the following functions:

- Initialize the toolkit.
- Map and unmap widgets to the screen.
- Process user input.
- Get information on widgets.
- Set widget attributes.

5 User Interface Language (UIL)

5.1 Description of the UIL

You use UIL to specify the initial state of the user interface for a DECwindows application. With UIL, you can "design" the look of your application by describing the following functions:

- Widgets and gadgets
- Callback routines
- Resources
- Widget hierarchy

5.2 Modifying your Application with UIL

When you use UIL to specify the layout of your application interface, you can rapidly prototype the look of the application. This enables easy modification.

The Open Software Foundation (OSF) has adopted UIL as part of its application interface architecture, and UIL should be portable across platforms.

For more information on UIL, refer to the *VMS DECwindows User Interface Language Reference Manual.*

6 Resource Management

6.1 Digital Resource Manager

The Digital Resource Manager (DRM) manages the output from the User Interface Language (UIL). The DRM performs the following functions:

- Initializes and registers user-defined classes and names.
- Opens the User Interface Description (UID) (output file from UIL) hierarchy.
- Fetches the user interface widgets and gadgets.
- "Realizes" (manages and maps the windows to the display).

The DRM operates during initialization, using data that has been statically defined by the UIL. After initialization, the application manages the user interface via runtime calls.

By using the UIL and DRM facilities, you can design interfaces quickly, change them easily, and enable them to be used by other applications.

A Overview of VMS DECwindows Documentation

The following overview is extracted from the *Overview of VMS DECwindows*.

The VMS DECwindows documentation set provides the following types of information:

- General User—Describes the DECwindows end-user environment; explains how to use DECwindows applications.
- Programming—Provides the information necessary to develop DECwindows applications.

The following sections describe the manuals in the DECwindows documentation set.

General User Documentation

The following manuals compose the general user documentation:

- *VMS DECwindows User's Guide*

  This guide describes the DECwindows user environment. Its chapters contain the following information:

  1 Provides hands-on experience through sample sessions.

  2 Describes how to use FileView.

  3 Explains how to manage files.

  4 Explains how to customize the DECwindows environment.

  The guide also provides a quick reference that explains the basic techniques used in the DECwindows environment, such as scrolling and selecting text.

- *VMS DECwindows Desktop Applications Guide*

  This guide describes, in a task-oriented manner, the DECwindows desktop applications, which include the following functionality:

  - Mail handling
  - Text editing
  - Terminal emulation
  - Calendar
  - Drawing screen images
  - Clock
  - Card filer
  - Calculator
  - Game

One chapter is devoted to each application; you can read about one application at a time without having to read the entire book.

Programming Documentation

The following manuals compose the programming documentation:

- *XUI Style Guide*

  This guide outlines standards for the development of DECwindows application interfaces. One of the major goals of the DECwindows product is to provide consistent application interfaces for the user environment. Using extensive illustrations of interface objects, this guide establishes guidelines that developers should follow to create applications that conform to the XUI style.

- *VMS DECwindows Xlib Routines Reference Manual*

  This manual provides the following information:

  - Description of the documentation format for Xlib routines.
  - Instructions for calling an Xlib routine using the VMS format and the MIT C format.
  - Description of each routine argument.
  - Description of how each routine functions.
  - List of values returned by the routine.

- *VMS DECwindows Toolkit Routines Reference Manual*

  This manual provides the following information:

  - Description of the documentation format for XUI Toolkit routines.
  - Instructions for calling an XUI Toolkit routine using the VMS format and the MIT C format.
  - Description of each routine argument.
  - Description of how each routine functions.

- *VMS DECwindows Guide to Xlib Programming: MIT C Binding* and *VMS DECwindows Guide to Xlib Programming: VAX binding*

  These guides explain how to use Xlib routines in DECwindows applications. They include the following information:

  - Overview of Xlib.
  - Discussion of programming considerations in C, FORTRAN, ADA, and PASCAL.
  - Tutorials that instruct you on how to use Xlib routines.

- *VMS Guide to Application Programming*

  This guide explains how to use the XUI Toolkit to develop applications. It includes an explanation of the User Interface Language (UIL) compiler.

- *VMS DECwindows User Interface Language Reference Manual*

This manual contains the following information:

- Description of the syntax and features of the User Interface Language (UIL).
- Description of the syntax of UIL low-level elements and module components.
- Instructions on how to use the UIL compiler and how to interpret compilation diagnostics.

- *VMS DECwindows Device Driver Manual*

  This manual provides reference information about the DECwindows device drivers. It explains the DECwindows device driver architecture and provides the information necessary to write port and class drivers.

- *VMS Compound Document Architecture Manual*

  This manual describes the concepts and tools associated with Digital Compound Document Architecture (CDA). This manual includes documentation of the CDA Toolkit routines.

B UIS$ Routine Reference

B.1 Introduction to UIS$ Routines

UIS$ routines provide the highest level interface to the VWS system. These routines use floating point coordinates that must be converted into appropriate device coordinates before you call the equivalent Xlib routine. Neither display list nor text-formatting UIS calls have equivalent library functions. If a display list is required and GKS, PHIGS, or some other interface is used, you must provide these routines independently of Xlib.

NOTE: GKS and PHIGS provide display lists themselves.

The UIS concept of an attribute block (ATB) is roughly equivalent to the X11 concept of a graphic context (GC). Each display list segment in UIS has 256 ATBs. The 256 ATBs from the default root are always available when you are not using display lists. In X11, an application can request that any number of GCs be created. Remember, however, that depending on system resources, the call might fail. No provisions exist for determining the current settings within an X11 GC. Therefore, no equivalent UIS$GET... routines are provided for the settings.

Table B–1 shows UIS$ routines with their equivalent Xlib routines, and an explanation of the routine functionality.

NOTE: If no equivalent Xlib routine exists, this is indicated in the table by N/A.

Table B–1 UIS$ Routines and their Equivalent Xlib Routines

UIS$ Routines	Xlib Routines	Explanation
UIS$BEGIN_SEGMENT	N/A	X11 provides no equivalent to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$CIRCLE	X$DRAW_ARC	You use the Xlib draw arc routine to draw circles. See "Drawing Graphics" in the *VMS DECwindows Guide to Xlib Programming*.
UIS$CLOSE_WINDOW	SYS$EXIT	The CLOSE_WINDOW routine is equivalent to a SYS$EXIT system service call and is the default action for the UIS$CLOSE_AST.

Table B-1 (Cont.) UIS$ Routines and their Equivalent Xlib Routines

UIS$ Routines	Xlib Routines	Explanation
UIS$COPY_OBJECT	N/A	X11 provides no equivalent to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$CREATE_COLOR_MAP	X$ALLOC_COLOR_CELLS	X11 incorporates the concept of *private* colormaps that can be created by the application and installed (see X$CREATE_COLORMAP). However, the installation of a colormap other than the default usually alters the colors in other windows. The use of COLOR by X11 applications is described in "Using Color" in the *VMS DECwindows Guide to Xlib Programming*. In general, to allocate colors for exclusive use (that is, you intend to alter the color), use X$ALLOC_COLOR_CELLS, requesting N planes and 1 color. This provides you with a single pixel value and a set of plane mask bits that can then be permuted to form a colormap that maintains the ability to be complemented (when you use GXxor mode with the plane mask bits). If no arithmetic operations must be performed on the bitmap, make the call with 1 plane and N colors. This has a better chance of succeeding. For applications using static colors, you can request "named" colors such as "Red."
UIS$CREATE_COLOR_MAP_SEG	X$CREATE_COLORMAP	To emulate this feature, you must create and install a private colormap for the entire hardware colormap, and the application must manage this colormap. See "Using Color" in the *VMS DECwindows Guide to Xlib Programming*. Complete control over the entire colormap is the only way to accomplish this.
UIS$CREATE_DISPLAY	N/A	This routine has no counterpart in X11. VWS uses this routine to initialize structures and create any needed colormap. In an X11 application, this routine would be replaced by more generic application initialization.

Table B–1 (Cont.) UIS$ Routines and their Equivalent Xlib Routines

UIS$ Routines	Xlib Routines	Explanation
UIS$CREATE_KB	X$SELECT_INPUT	X11 has no equivalent to the UIS virtual keyboard. In X11, you can select the types of input events. See "Handling Events" in the *VMS DECwindows Guide to Xlib Programming*.
UIS$CREATE_TERMINAL	N/A	The DECterm VT340 terminal emulation windows can only be created from the DECwindows session manager. No mechanism exists to create a terminal window from within a program.
UIS$CREATE_TB	N/A	X11 provides no digitizer support. Tablets are supported only as replacements for the mouse.
UIS$CREATE_ TRANSFORMATION	N/A	Since X11 provides only a device-dependent integer coordinate space with each unit representing a pixel, programmers must provide their own world coordinates and transformations.
UIS$CREATE_WINDOW	X$CREATE_WINDOW	This routine performs a device assignment to the workstation screen. This is the equivalent of X$OPEN_ DISPLAY, which establishes the link to the display. In addition, an X$CREATE_WINDOW would be performed to create and initialize the window structures; this would be followed by an X$MAP_WINDOW to make the window visible. See "Managing the Client-Server Connection" and "Working with Windows" in the *VMS DECwindows Guide to Xlib Programming*.
UIS$DELETE_COLOR_MAP	X$FREE_COLORMAP	See "Using Color," in the *VMS DECwindows Guide to Xlib Programming*.
UIS$DELETE_COLOR_MAP_ SEGMENT	X$FREE_COLORMAP	See "Using Color," in the *VMS DECwindows Guide to Xlib Programming*.
UIS$DELETE_DISPLAY	X$CLOSE_DISPLAY	Closing the display is the nearest equivalent under X11. The X$DESTROY_WINDOW call is more like moving the viewport offscreen under VWS. It leaves everything set up (like the connection), but does not leave the window.
UIS$DELETE_KB	X$SELECT_INPUT	X11 provides no equivalent to a virtual keyboard. Types of input events can be selected and keyboard events ignored. See "Handling Events," in the *VMS DECwindows Guide to Xlib Programming*.

Table B–1 (Cont.) UIS$ Routines and their Equivalent Xlib Routines

UIS$ Routines	Xlib Routines	Explanation
UIS$DELETE_OBJECT	N/A	X11 provides no equivalent to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$DELETE_PRIVATE	N/A	X11 provides no equivalent to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$DELETE_TB	N/A	X11 provides no digitizer support. Tablets are supported only as replacements for the mouse.
UIS$DELETE_ TRANSFORMATION	N/A	Since X11 provides only a device-dependent integer coordinate space with each unit representing a pixel, programmers must provide their own world coordinates and transformations.
UIS$DELETE_WINDOW	X$CLOSE_DISPLAY	Closing the display is the nearest equivalent under X11. The X$DESTROY_WINDOW call is more like moving the viewport offscreen under VWS. It leaves everything set up (like the connection), but does not leave the window.
UIS$DISABLE_DISPLAY_LIST	N/A	X11 provides no equivalent to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$DISABLE_KB	X$SELECT_INPUT	X11 provides no equivalent to a virtual keyboard. The types of input events can be selected and keyboard events ignored. See "Handling Events," in the *VMS DECwindows Guide to Xlib Programming*.
UIS$DISABLE_TB	N/A	X11 provides no digitizer support. Tablets are supported only as replacements for the mouse.

Table B–1 (Cont.) UIS$ Routines and their Equivalent Xlib Routines

UIS$ Routines	Xlib Routines	Explanation
UIS$DISABLE_VIEWPORT_KB	N/A	UIS uses this routine to disconnect a virtual KB from a window and remove the window from the list of windows that can be cycled to. X11 has no concept of a virtual KB. You either accept input focus and keyboard input events or do not choose to receive these events of the input focus. See X$SELECT_INPUT.
UIS$ELLIPSE	X$DRAW_ARC	Draw ellipses by using the Xlib draw arc routine. See "Drawing Graphics," in the *VMS DECwindows Guide to Xlib Programming*.
UIS$ENABLE_DISPLAY_LIST	N/A	X11 provides no equivalent to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$ENABLE_KB	X$SET_INPUT_FOCUS	This call sets the input focus (the closest concept to connecting the physical keyboard to a window).
UIS$ENABLE_KB	N/A	X11 provides no digitizer support. Tablets are supported only as replacements for the mouse.
UIS$ENABLE_VIEWPORT_KB	N/A	UIS uses this routine to associate a virtual KB with a window and add the window to the list of windows that can be cycled to. X11 has no concept of a virtual KB. You either accept input focus and keyboard input events or do not choose to receive these events of the input focus. See X$SELECT_INPUT.
UIS$END_SEGMENT	N/A	X11 provides no equivalents to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$ERASE	X$CLEAR_AREA	Both Clear Area and Clear Window routines are provided to erase portions of windows. You cannot use the Clear Area function on a PIXMAP. Instead, you should use a filled rectangle the size of the screen in the background color. See "Drawing Graphics" in the *VMS DECwindows Guide to Xlib Programming*.

Table B-1 (Cont.) UIS$ Routines and their Equivalent Xlib Routines

UIS$ Routines	Xlib Routines	Explanation
UIS$EXECUTE	N/A	X11 provides no equivalents to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$EXECUTE_DISPLAY	N/A	X11 provides no equivalents to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$EXPAND_ICON	X$SET_WM_HINTS	Generally, the user controls the state of the application window. To set the initial state of a window, use the property routines to communicate to the server. In addition, the server honors the hints after the window has been created and mapped. Thus, if you specify the Initial State for the window as X$C_NORMAL_STATE with the X$SET_WM_HINTS, a window currently in an icon state will be expanded.
UIS$EXTRACT_HEADER	N/A	X11 provides no equivalents to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$EXTRACT_OBJECT	N/A	X11 provides no equivalents to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$EXTRACT_PRIVATE	N/A	X11 provides no equivalents to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$EXTRACT_REGION	N/A	X11 provides no equivalents to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.

Table B-1 (Cont.) UIS$ Routines and their Equivalent Xlib Routines

UIS$ Routines	Xlib Routines	Explanation
UIS$EXTRACT_TRAILER	N/A	X11 provides no equivalents to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$FIND_PRIMITIVE	N/A	X11 provides no equivalents to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$FIND_SEGMENT	N/A	X11 provides no equivalents to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$GET_ABS_POINTER_POS	X$QUERY_POINTER	This function returns the position of the pointer relative to the window as well as the current state of the modifier keys and buttons. See the *Xlib Routines Reference Manual*, Part 1, "Window Routines."
UIS$GET_ALIGNED_POSITION	N/A	X11 does not provide text formatting functions.
UIS$GET_ARC_TYPE	N/A	X11 does not provide inquiry functions for GCs (the equivalent of UIS ATBs).
UIS$GET_BACKGROUND_INDEX	N/A	X11 does not provide inquiry functions for GCs (the equivalent of UIS ATBs).
UIS$GET_BUTTONS	X$QUERY_POINTER	This function returns the position of the pointer relative to the window as well as the current state of the modifier keys and buttons. See the *Xlib Routines Reference Manual*, Part 1, "Window Routines."
UIS$GET_CHAR_ROTATION	N/A	X11 does not provide text rotation.
UIS$GET_CHAR_SIZE	N/A	X11 does not provide text scaling.
UIS$GET_CHAR_SPACING	N/A	X11 does not provide text formatting.
UIS$GET_CLIP	N/A	X11 does not provide inquiry functions for GCs (the equivalent of UIS ATBs).
UIS$GET_COLOR	X$QUERY_COLOR	Provides the RGB values for the specified index.
UIS$GET_COLORS	X$QUERY_COLORS	Provides the RGB values for the specified index values.

Table B-1 (Cont.) UIS$ Routines and their Equivalent Xlib Routines

UIS$ Routines	Xlib Routines	Explanation
UIS$GET_CURRENT_OBJECT	N/A	X11 provides no equivalents to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$GET_DISPLAY_SIZE	See "Explanation."	The X$DISPLAY_WIDTH, X$DISPLAY_WIDTH_MM, X$DISPLAY_HEIGHT, and X$DISPLAY_HEIGHT_MM calls provide the information needed to emulate this.
UIS$GET_FILL_PATTERN	N/A	X11 does not provide inquiry functions for GCs (the equivalent of UIS ATBs).
UIS$GET_FONT	N/A	X11 does not provide inquiry functions for GCs (the equivalent of UIS ATBs).
UIS$GET_FONT_ATTRIBUTES	X$QUERY_FONT	This routine, as well as X$LOOKUP_FONT_WITH_INFO, can return information associated with this call.
UIS$GET_FONT_SIZE	X$QUERY_FONT	This routine, as well as X$LOOKUP_FONT_WITH_INFO, can return information associated with this call.
UIS$GET_HW_COLOR_INFO	N/A	The information returned by this call is available through a number of individual calls. See the *Xlib Routines Reference Manual*, Part 1, "Display Routines."
UIS$GET_INTENSITIES	X$QUERY_COLORS	Use the X$QUERY_COLORS routines.
UIS$GET_INTENSITY	X$QUERY_COLOR	Use the X$QUERY_COLOR routines.
UIS$GET_KB_ATTRIBUTES	X$GET_KEYBOARD_CONTROL	The attributes are not specified in the same manner but are available through this routine.
UIS$GET_LINE_STYLE	N/A	X11 does not provide inquiry functions for GCs (the equivalent of UIS ATBs).
UIS$GET_LINE_WIDTH	N/A	X11 does not provide inquiry functions for GCs (the equivalent of UIS ATBs).
UIS$GET_NEXT_OBJECT	N/A	X11 provides no equivalents to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$GET_OBJECT_ATTRIBUTES	N/A	X11 provides no equivalents to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.

Table B-1 (Cont.) UIS$ Routines and their Equivalent Xlib Routines

UIS$ Routines	Xlib Routines	Explanation
UIS$GET_PARENT_SEGMENT	N/A	X11 provides no equivalents to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$GET_POINTER_POSITION	X$QUERY_POINTER	This function returns the position of the pointer relative to the window as well as the current state of the modifier keys and buttons. See the *Xlib Routines Reference Manual*, Part 1, "Window Routines."
UIS$GET_POSITION	N/A	X11 does not provide text formatting functions.
UIS$GET_PREVIOUS_OBJECT	NA	X11 provides no equivalents to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$GET_ROOT_SEGMENT	NA	X11 provides no equivalents to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$GET_TB_INFO	N/A	X11 provides no digitizer support. Tablets are supported only as replacements for the mouse.
UIS$GET_TB_POSITION	N/A	X11 provides no digitizer support. Tablets are supported only as replacements for the mouse.
UIS$GET_TEXT_FORMATTING	N/A	X11 does not provide text formatting functions.
UIS$GET_TEXT_MARGINS	N/A	X11 does not provide text formatting functions.
UIS$GET_TEXT_PATH	N/A	X11 does not provide text drawing path (left-right) functions.
UIS$GET_TEXT_SLOPE	N/A	X11 does not provide text slope (rotation) functions.
UIS$GET_VCM_ID	N/A	X11 has no equivalent function. The colormap ID for X11 is the nearest equivalent and is returned when the colormap is created or the workstation default can be returned. In general, the X11 colormap is not equivalent to the UIS colormap.

Table B-1 (Cont.) UIS$ Routines and their Equivalent Xlib Routines

UIS$ Routines	Xlib Routines	Explanation
UIS$GET_VIEWPORT_ICON	N/A	In general, icons are managed by the window manager. Communication and inquiry are performed via structures that provide "hints" to the window manager. See "Using Properties" in the *Guide to Xlib Programming* to Communicate with the Window Manager. The X$SET_WM_HINTS routine contains an ICON WINDOW field that you can optionally use to supply a window that serves as the icon. This window ID is user-created. Normally, icons are supplied when you provide a PIXMAP to be used as the icon display data.
UIS$GET_VIEWPORT_POSITION	X$GET_WINDOW_ATTRIBUTES	You can obtain a data structure that provides information about the current position size and other window attributes.
UIS$GET_VIEWPORT_SIZE	X$GET_WINDOW_ATTRIBUTES	You can obtain a data structure that provides information about the current position size and other window attributes.
UIS$GET_VISIBILITY	N/A	A direct method of obtaining this information does not exist in X11. Since the X11 application can be notified of all requests to expose a window and can be notified (after the fact) of any window occlusion, the application can therefore keep track of the current state of visibility.
UIS$GET_WINDOW_ATTRIBUTES	X$GET_WINDOW_ATTRIBUTES	You can obtain a data structure that provides information about the current position size and other window attributes.
UIS$GET_WINDOW_SIZE	X$GET_GEOMETRY	You can obtain a data structure that provides information about the current position size and other window attributes.
UIS$GET_WRITING_INDEX	N/A	X11 does not provide inquiry functions for GCs (the equivalent of UIS ATBs).
UIS$GET_WRITING_MODE	N/A	X11 does not provide inquiry functions for GCs (the equivalent of UIS ATBs).

Table B–1 (Cont.) UIS$ Routines and their Equivalent Xlib Routines

UIS$ Routines	Xlib Routines	Explanation
UIS$GET_WS_COLOR	X$LOOKUP_COLOR	DECwindows contains a set of named colors. This call returns the closest RGB values available for the hardware, as well as the ideal RGB values for the specified color. Appendix C of the *Guide to Xlib Programming* provides the names of the predefined colors for DECwindows.
UIS$GET_WS_INTENSITY	X$LOOKUP_COLOR	The intensity is returned as RGB values. You can use NTSC to convert the RGB values to an intensity. DECwindows contains a set of named colors. This call returns the closest RGB values available for the hardware, as well as the ideal RGB values for the specified color. Appendix C of the *Guide to Xlib Programming* provides the names of the predefined colors for DECwindows.
UIS$HLS_TO_RGB	N/A	The X11 RGB system is based on a 16-bit integer value, while the UIS RGB system uses a floating point between 0 and 1. HLS conversion routines are widely available, and one is included here in Appendix C. Xlib libraries provide no conversion routines.
UIS$HSV_TO_RGB	N/A	The X11 RGB system is based on a 16-bit integer value, while the UIS RGB system uses a floating point between 0 and 1. HLS conversion routines are widely available, and one is included here in Appendix C. Xlib libraries provide no conversion routines.
UIS$IMAGE	X$PUT_IMAGE	See "Drawing Graphics" in the *VMS DECwindows Guide to Xlib Programming*. Note that you might have to reformat image data unless you create and install a colormap for images greater than 1 bit deep.
UIS$INSERT_OBJECT	N/A	X11 provides no equivalents to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.

Table B–1 (Cont.) UIS$ Routines and their Equivalent Xlib Routines

UIS$ Routines	Xlib Routines	Explanation
UIS$LINE	X$DRAW_SEGMENT	The X$DRAW_POINT routine is also used to draw individual points (zero length lines). See "Drawing Graphics" in the *VMS DECwindows Guide to Xlib Programming*.
UIS$LINE_ARRAY	X$DRAW_SEGMENTS	The X$DRAW_POINTS routine is also used to draw individual points (zero length lines). See "Drawing Graphics" in the *VMS DECwindows Guide to Xlib Programming*.
UIS$MEASURE_TEXT	X$QUERY_TEXT_EXTENTS	X11 provides an equivalent function to measure the length of a text string. Note that control strings and text formatting are not provided for text output.
UIS$MOVE_AREA	X$COPY_AREA	This is equivalent to an X$COPY_AREA followed by one or more X$CLEAR_ AREA operations to clear the area no longer covered by the area moved. See "Drawing Graphics" in the *VMS DECwindows Guide to Xlib Programming*.
UIS$MOVE_VIEWPORT	X$MOVE_WINDOW	This function changes the location of the window on the screen. In X11, this function can move the window partially offscreen. This feature is not possible with the UIS call.
UIS$MOVE_WINDOW	N/A	X11 provides no equivalent function, since this relocates the display list. When no display list is used, it works much like UIS$MOVE_AREA.
UIS$NEW_TEXT_LINE	N/A	X11 does not provide text formatting functions.
UIS$PLOT	X$DRAW_LINE	The X$DRAW_POINT routine is also used to draw individual points (zero length lines). See "Drawing Graphics" in the *VMS DECwindows Guide to Xlib Programming*.
UIS$PLOT_ARRAY	X$DRAW_LINES	The X$DRAW_POINTS routine is also used to draw individual points (zero length lines). See "Drawing Graphics" in the *VMS DECwindows Guide to Xlib Programming*.
UIS$POP_VIEWPORT	X$RAISE_WINDOW	These are directly equivalent.

Table B–1 (Cont.) UIS$ Routines and their Equivalent Xlib Routines

UIS$ Routines	Xlib Routines	Explanation
UIS$PRESENT	N/A	DECwindows applications are generally started with SYS$OUTPUT and given a device class of DC$_WORKSTATION (device controller type WS). Applications should first check for this device class as SYS$OUTPUT. If the class is not DC$_WORKSTATION, the application should check for a logical name DECW$DISPLAY to be defined. If this logical is present, the X$OPENDISPLAY call uses this as the display destination. If both of these options fail, and your application supports both UIS and DECwindows, you can call UIS$PRESENT to see if UIS is available.
UIS$PRIVATE	N/A	X11 provides no equivalents to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$PUSH_VIEWPORT	X$LOWER_WINDOW	These are directly equivalent.
UIS$READ_CHAR	N/A	Keyboard input is delivered via the X EVENT mechanism.
UIS$RESIZE_WINDOW	X$CHANGE_WINDOW_ATTRIBUTES	You can use this call to resize the X11 window.
UIS$RESTORE_CMS_COLORS	X$INSTALL_COLORMAP	This X11 function installs a colormap. When you use a private colormap, you can use this function to do the binding to the hardware. Note that all colors are affected by this call.
UIS$RGB_TO_HLS	N/A	The X11 RGB system is based on a 16-bit integer value, while the UIS RGB system uses a floating point between 0 and 1. HLS conversion routines are widely available, and one is included here in Appendix C. Xlib libraries provide no conversion routines.
UIS$RGB_TO_HSV	N/A	The X11 RGB system is based on a 16-bit integer value, while the UIS RGB system uses a floating point between 0 and 1. HLS conversion routines are widely available, and one is included here in Appendix C. Xlib libraries provide no conversion routines.

Table B–1 (Cont.) UIS$ Routines and their Equivalent Xlib Routines

UIS$ Routines	Xlib Routines	Explanation
UIS$SET_ADDOPT_AST	N/A	DECwindows has no additional options selection. Use the DECwindows Toolkit and the appropriate widget set to find equivalent functionality.
UIS$SET_ALIGNED_POSITION	N/A	X11 does not provide text formatting functions.
UIS$SET_ARC_TYPE	X$SET_ARC_MODE	Most of the ARC drawing styles are available in X11.
UIS$SET_BACKGROUND_INDEX	X$SET_BACKGROUND	This is provided by the appropriate GC creation or modification command. See the *Xlib Routines Reference Manual*, Part 1, "Graphics Context Routines." The background index is specified in the BACKGROUND value in the GC Values data structure.
UIS$SET_BUTTON_AST	N/A	This is included in X EVENT processing. See the *Xlib Reference Manual*, Part 1, "Event Routines."
UIS$SET_CHAR_ROTATION	N/A	X11 does not provide character rotation.
UIS$SET_CHAR_SLANT	N/A	X11 does not provide character shearing.
UIS$SET_CHAR_SPACING	N/A	X11 does not provide text formatting functions.
UIS$SET_CLIP	X$SET_CLIP_RECTANGLES	This provides a superset of UIS clipping.
UIS$SET_CLOSE_AST	N/A	The DECwindows Toolkit contains the only equivalent concept in DECwindows.
UIS$SET_COLOR	X$STORE_COLOR	X$STORE_COLOR sets the RGB value in a previously allocated color cell. The RGB values must be converted into 16-bit integer values. See the *Xlib Reference Manual*, Part 1, "Color Routines."
UIS$SET_COLORS	X$STORE_COLORS	X$STORE_COLORS sets RGB values in a list of previously allocated color cells. The RGB values must be converted into 16-bit integer values. See the *Xlib Reference Manual*, Part 1, "Color Routines."
UIS$SET_EXPAND_ICON_AST	N/A	You can obtain equivalent X EVENTS by using the EVENT MASK in the X$CHANGE_WINDOW_ATTRIBUTES call. See the *Xlib Routines Reference Manual*, Part 1, "Window Routines," for more information.

Table B–1 (Cont.) UIS$ Routines and their Equivalent Xlib Routines

UIS$ Routines	Xlib Routines	Explanation
UIS$SET_FILL_PATTERN	X$SET_STIPPLE	UIS fill patterns are the equivalent of stipple patterns in X11. A stipple is a single-bit deep PIXMAP. The PIXMAP must be created and the pattern drawn into it. Usually this is accomplished with the X$PUT_IMAGE operation. The stipple can then be used in a GC as a pattern or mask.
UIS$SET_FONT	X$SET_FONT	This routine sets a font ID into a GC. You must use the X$LOAD_FONT routine to obtain the font ID. See the *Xlib Routines Reference Manual*, Part 1, "Graphics Context Routines" and Part 2, "Font Routines" for information on these routines.
UIS$SET_GAIN_KB_AST	N/A	Equivalent X EVENTS exist for obtaining INPUT FOCUS. See "Handling Events" in the *Guide to Xlib Programming*.
UIS$SET_INSERTION_POSITION	N/A	X11 provides no equivalents to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.
UIS$SET_INTENSITIES	X$STORE_COLOR	X$STORE_COLOR sets an RGB value in a previously allocated color cell. RGB values must be converted into 16-bit integer values. Derive RGB values by using the intensity value for each of the RGB components. See the *Xlib Routines Reference Manual*, Part 1, "Color Routines."
UIS$SET_INTENSITY	X$STORE_COLORS	X$STORE_COLORS sets a list of RGB values in a list of previously allocated color cells. RGB values must be converted into 16-bit integer values. Derive RGB values by using the intensity value for each of the RGB components. See the *Xlib Routines Reference Manual*, Part 1, "Color Routines."
UIS$SET_KB_AST	N/A	You can obtain equivalent X EVENTS by using the EVENT MASK in the X$CHANGE_WINDOW_ATTRIBUTES call. See the *Xlib Routines Reference Manual*, Part 1, "Window Routines" for more information.

Table B-1 (Cont.) UIS$ Routines and their Equivalent Xlib Routines

UIS$ Routines	Xlib Routines	Explanation
UIS$SET_KB_ATTRIBUTES	X$CHANGE_KEYBOARD_ CONTROL	The KB can be remapped as appropriate. Note that this is done in a completely different fashion in X11. See the *Xlib Routines Reference Manual*, Part 2, "Window and Session Manager Routines" for more information.
UIS$SET_KB_COMPOSE2	X$SET_MODIFIER_MAPPING	Along with the X$CHANGE_ KEYBOARD_MAPPING routine, this can remap the keyboard input. See the *Xlib Routines Reference Manual*, Part 2, "Window and Session Manager Routines" for more information.
UIS$SET_KB_COMPOSE3	X$SET_MODIFIER_MAPPING	Along with the X$CHANGE_ KEYBOARD_MAPPING routine, this can remap the keyboard input. See the *Xlib Routines Reference Manual*, Part 2, "Window and Session Manager Routines" for more information.
UIS$SET_KB_KEYTABLE	X$CHANGE_KEYBOARD_ MAPPING	Along with the X$SET_KEYBOARD_ MAPPING routine, this can remap the keyboard input. See the *Xlib Routines Reference Manual*, Part 2, "Window and Session Manager Routines" for more information.
UIS$SET_LINE_STYLE	X$SET_LINE_ATTRIBUTES	See the *Xlib Routines Reference Manual*, Part 1, "Graphic Context Routines" for more information.
UIS$SET_LINE_WIDTH	X$SET_LINE_ATTRIBUTES	See the *Xlib Routines Reference Manual*, Part 1, "Graphic Context Routines" for more information.
UIS$SET_LOSE_KB_AST	N/A	You can obtain equivalent X EVENTS by using the EVENT MASK in the X$CHANGE_WINDOW_ATTRIBUTES call. See the *Xlib Routines Reference Manual*, Part 1, "Window Routines" for more information.
UIS$SET_MOVE_INFO_AST	N/A	You can obtain equivalent X EVENTS by using the EVENT MASK in the X$CHANGE_WINDOW_ATTRIBUTES call. See the *Xlib Routines Reference Manual*, Part 1, "Window Routines" for more information.
UIS$SET_POINTER_AST	N/A	You can obtain equivalent X EVENTS by using the EVENT MASK in the X$CHANGE_WINDOW_ATTRIBUTES call. See the *Xlib Routines Reference Manual*, Part 1, "Window Routines" for more information.

Table B-1 (Cont.) UIS$ Routines and their Equivalent Xlib Routines

UIS$ Routines	Xlib Routines	Explanation
UIS$SET_POINTER_PATTERN	X$DEFINE_CURSOR	See the *Xlib Routines Reference Manual*, Part 2, "Cursor Routines," for more information.
UIS$SET_POINTER_POSITION	X$WARP_POINTER	See the *Xlib Routines Reference Manual*, Part 2, "Window and Session Manager Routines" for more information.
UIS$SET_POSITION	N/A	X11 does not provide text formatting functions.
UIS$SET_RESIZE_AST	N/A	You can obtain equivalent X EVENTS by using the EVENT MASK in the X$CHANGE_WINDOW_ATTRIBUTES call. See the *Xlib Routines Reference Manual*, Part 1, "Window Routines" for more information.
UIS$SET_SHRINK_TO_ICON_AST	N/A	You can obtain equivalent X EVENTS by using the EVENT MASK in the X$CHANGE_WINDOW_ATTRIBUTES call. See the *Xlib Routines Reference Manual*, Part 1, "Window Routines" for more information.
UIS$SET_TB_AST	N/A	X11 provides no digitizer support. Tablets are supported only as replacements for the mouse.
UIS$SET_TEXT_FORMATTING	N/A	X11 does not provide text formatting functions.
UIS$SET_TEXT_MARGINS	N/A	X11 does not provide text formatting functions.
UIS$SET_TEXT_PATH	N/A	X11 does not provide text formatting functions.
UIS$SET_TEXT_SLOPE	N/A	X11 does not provide text formatting functions.
UIS$SET_WRITING_MODE	X$SET_FUNCTION	This is provided by the appropriate GC creation or modification command. See the *Xlib Routines Reference Manual*, Part 1, "Graphics Context Routines" for more information. The FUNCTION is the actual logical operator used for the operation. UIS "modes" are a combination of FUNCTION, FILL STYLE, FILL STIPPLE, FOREGROUND, and BACKGROUND pixel values. A routine that shows the mapping for most UIS writing modes is provided in Appendix G.

Table B–1 (Cont.) UIS$ Routines and their Equivalent Xlib Routines

UIS$ Routines	Xlib Routines	Explanation
UIS$SHRINK_TO_ICON	X$SET_WM_HINTS	The state of the application is generally controlled exclusively by the user. Set the initial state of a window by using the property routines to communicate to the server. In addition, the server honors the hints after the window has been created and mapped. Thus, if you specify the Initial State for the window as X$C_ICONIC_STATE, a window currently in a window state will be iconified.
UIS$SOUND_BELL	X$BELL	See the *Xlib Routines Reference Manual*, Part 2, "Window and Session Manager Routines" for more information.
UIS$SOUND_CLICK	N/A	The keyclick cannot be sounded in X11.
UIS$TEST_KB	N/A	Applications should keep track of this through the X EVENT mechanism for INPUT focus gain and lose events.
UIS$TEXT	X$DRAWTEXT	X11 routines doe not provide any of the text formatting or control lists provided by UIS.
UIS$TRANSFORM_OBJECT	N/A	X11 provides no equivalents to the UIS$ display list routines. Programmers must supply their own display list routines or reprogram in a higher-level graphic interface such as GKS or PHIGS.

C UISDC$ Routine Reference

C.1 Introduction to UISDC$ Routines

In addition to the world coordinate interface (UIS), VWS provides a device-coordinate, or pixel-level, interface (UISDC) to the graphics system services.

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.

Table C–1 gives UISDC routines with their equivalent Xlib routines, and an explanation of the routine functionality.

NOTE: If an equivalent Xlib routine does not exist, this is indicated in the table by N/A.

Table C–1 UISDC Routines and their Equivalent Xlib Routines

UISDC Routines	Xlib Routines	Explanation
UISDC$ALLOCATE_DOP	N/A	The DOP interface is a device-dependent mechanism that queues drawing packets to the VSII/GPX and VS2000/GPX. No comparable hardware interface exists under X11.
UISDC$CIRCLE	X$DRAW_ARC	Use the Xlib draw arc routine to draw circles. See "Drawing Graphics" in the *VMS DECwindows Guide to Xlib Programming* for more information.
UISDC$ELLIPSE	X$DRAW_ARC	Use the Xlib draw arc routine to draw ellipses. See "Drawing Graphics" in the *VMS DECwindows Guide to Xlib Programming* for more information.
UISDC$ERASE	X$CLEAR_AREA	Clear Area and Clear Window routines are both provided to erase portions of windows. Note that you cannot use the Clear Area function on a PIXMAP; instead, a filled rectangle the size of the screen in the background color is also equivalent. See "Drawing Graphics" in the *VMS DECwindows Guide to Xlib Programming* for more information.

Table C-1 (Cont.) UISDC Routines and their Equivalent Xlib Routines

UISDC Routines	Xlib Routines	Explanation
UISDC$EXECUTE_DOP_ASYNCH	N/A	The DOP interface is a device-dependent mechanism that queues drawing packets to the VSII/GPX and VS2000/GPX. No comparable hardware interface exists under X11.
UISDC$EXECUTE_DOP_SYNCH	N/A	The DOP interface is a device-dependent mechanism that queues drawing packets to the VSII/GPX and VS2000/GPX. No comparable hardware interface exists under X11.
UISDC$GET_ALIGNED_POSITION	N/A	X11 provides no text formatting or the concept of a current text-writing position.
UISDC$GET_CHAR_SIZE	N/A	X11 does not provide text scaling.
UISDC$GET_CLIP	N/A	X11 does not provide query routines for GCs.
UISDC$GET_POINTER_POSITION	X$QUERY_POINTER	This function returns the position of the pointer relative to the window. It also returns the the current state of the modifier keys and buttons. See the *Xlib Routines Reverence Manual*, Part 1, "Window Routines."
UISDC$GET_POSITION	N/A	X11 provides no text formatting.
UISDC$GET_TEXT_MARGINS	N/A	X11 provides no text formatting or the concept of text margins.
UISDC$GET_VISIBILITY	N/A	Since an X11 application can be notified of all requests to expose a window and can be notified of the occluding of a window after the fact, there is no direct way to obtain this information. However, the application can keep track of the current state of visibility.
UISDC$IMAGE	X$PUT_IMAGE	See "Drawing Graphics" in the *VMS DECwindows Guide to Xlib Programming*. Note that image data might require reformatting unless you create and install a colormap for images greater than 1 bit deep.
UISDC$LINE	X$DRAW_SEGMENT	See "Drawing Graphics" in the *VMS DECwindows Guide to Xlib Programming* for more information.
UISDC$LINE_ARRAY	X$DRAW_SEGMENTS	See "Drawing Graphics" in the *VMS DECwindows Guide to Xlib Programming* for more information.

Table C-1 (Cont.) UISDC Routines and their Equivalent Xlib Routines

UISDC Routines	Xlib Routines	Explanation
UISDC$LOAD_BITMAP	N/A	This routine loads a user bitmap into offscreen video memory. In some ways, this is similar to the X11 concept of a PIXMAP, but the concepts differ. The principal use for this under UIS is to load font data for drawing with DOPs.
UISDC$MEASURE_TEXT	X$QUERY_TEXT_EXTENTS	X11 provides an equivalent function to measure the length of a text string. Note that control strings and text formatting are not provided for text output.
UISDC$MOVE_AREA	X$COPY_AREA	See "Drawing Graphics" in the *VMS DECwindows Guide to Xlib Programming* for more information.
UISDC$NEW_TEXT_LINE	N/A	X11 provides no text formatting.
UISDC$PLOT	X$DRAW_LINE	See "Drawing Graphics" in the *VMS DECwindows Guide to Xlib Programming* for more information.
UISDC$PLOT_ARRAY	X$DRAW_LINES	See "Drawing Graphics" in the *VMS DECwindows Guide to Xlib Programming* for more information.
UISDC$QUEUE_DOP	N/A	The DOP interface is a device-dependent mechanism that queues drawing packets to the VSII/GPX and VS2000/GPX. No comparable hardware interface exists under X11.
UISDC$READ_IMAGE	X$GET_IMAGE	Since the bitmap contents are not guaranteed under X11, be extremely cautious when you use this function.
UISDC$SET_ALIGNED_POSITION	N/A	X11 provides neither text formatting nor the concept of a current text-writing position.
UISDC$SET_BUTTON_AST	N/A	This is part of X EVENT processing. See the *Xlib Routines Reference Manual*, Part 1, "Event Routines."
UISDC$SET_CHAR_SIZE	N/A	X11 does not provide text scaling.
UISDC$SET_CLIP	X$SET_CLIP_RECTANGLES	X11 provides a superset of UIS clipping.
UISDC$SET_POINTER_AST	N/A	You can accomplish equivalent X EVENTS by using the EVENT MASK in the X$CHANGE_WINDOW_ATTRIBUTES call. See the *Xlib Routines Reference Manual*, Part 1, "Window Routines," for more information.

Table C–1 (Cont.) UISDC Routines and their Equivalent Xlib Routines

UISDC Routines	Xlib Routines	Explanation
UISDC$SET_POINTER_PATTERN	X$DEFINE_CURSOR	See the *Xlib Routines Reference Manual*, Part 2, "Cursor Routines," for more information.
UISDC$SET_POINTER_POSITION	X$QUERY_POINTER	This function returns the position of the pointer relative to the window; it also returns the current state of the modifier keys and buttons. See the *Xlib Routines Reference Manual*, Part 1, "Window Routines."
UISDC$SET_POSITION	N/A	X11 provides no text formatting.
UISDC$SET_TEXT_MARGINS	N/A	X11 provides no text formatting or the concept of text margins.
UISDC$TEXT	X$DRAW_TEXT	The X11 routines do not provide any of the text formatting or control lists provided by UIS.

D HCUIS$ Routine Reference

D.1 Introduction to HCUIS$ Routines

In addition to the world coordinate interface (UIS) and device-coordinate, or pixel-level, interface (UISDC) to the graphics system services, VWS provides a hard copy (HCUIS) interface.

Table D–1 gives HCUIS routines with their equivalent Xlib routines, and an explanation of the routine functionality.

NOTE: **If an equivalent Xlib routine does not exist, this is indicated in the table by N/A.**

Table D–1 HCUIS Routines and their Equivalent Xlib Routines

HCUIS Routines	Xlib Routines	Explanation
HCUIS$BEGIN_TRANSLATOR	N/A	No equivalent routine exists.
HCUIS$END_TRANSLATOR	N/A	No equivalent routine exists.
HCUIS$READ_BUFFER	N/A	No equivalent routine exists.
HCUIS$READ_DISPLAY	N/A	No equivalent routine exists.
HCUIS$TRANSLATE	N/A	No equivalent routine exists.
HCUIS$WRITE_BUFFER	N/A	No equivalent routine exists.
HCUIS$WRITE_DISPLAY	N/A	No equivalent routine exists.

E UIS Fonts to DECwindow Equivalents

Table E–1 gives VWS fonts with their nearest DECwindows equivalent.

Table E–1 VWS and DECwindows Fonts

VWS Font	Nearest DECwindows Equivalent
DTABER0I03WK00GG0001UZZZZ02A000	-*-Courier-Medium-R-Normal–*-140-*-*-M-*-*-*
DEUISPATAAAAAAF000000000DA	There is no DECwindows pattern font.
DTABER0003WK00GG0001UZZZZ02A000	-*-Helvetica-Medium-R-Normal–*-140-*-*-P-*-*-*
DTABER0003WK00PG0001UZZZZ02A000	-*-Helvetica-Bold-R-Normal–*-140-*-*-P-*-*-*
DTABER0G03CK00GG0001UZZZZ02A000	-*-Courier-Medium-R-Normal–*-120-*-*-M-*-*-*
DTABER0I03WK00GG0001UZZZZ02A000	-*-Courier-Medium-R-Normal–*-140-*-*-M-*-*-*
DTABER0I03WK00PG0001UZZZZ02A000	-*-Courier-Bold-R-Normal–*-140-*-*-M-*-*-*
DTABER0M03CK00GG0001UZZZZ02A000	-*-Courier-Medium-R-Normal–*-120-*-*-M-*-*-*
DTABER0M06OK00GG0001UZZZZ02A000	-*-Courier-Medium-R-Normal–*-240-*-*-M-*-*-*
DTABER0R03WK00GG0001UZZZZ02A000	-*-Courier-Medium-R-Normal–*-140-*-*-M-*-*-*
DTABER0R03WK00PG0001UZZZZ02A000	-*-Courier-Bold-R-Normal–*-140-*-*-M-*-*-*
DTABER0R07SK00GG0001UZZZZ02A000	-*-Courier-Medium-R-Normal–*-240-*-*-M-*-*-*
DTABER0R07SK00PG0001UZZZZ02A000	-*-Courier-Bold-R-Normal–*-240-*-*-M-*-*-*
DTERMING03CK00PG0001UZZZZ02A000	-*-Times-Bold-R-Normal–*-120-*-*-P-*-*-*
DTERMINM03CK00PG0001UZZZZ02A000	-*-Times-Bold-R-Normal–*-120-*-*-P-*-*-*
DTERMINM06OK00PG0001UZZZZ02A000	-*-Times-Bold-R-Normal–*-240-*-*-P-*-*-*
DVWSVT0A00KK00GG0001UZZZZ02A000	-*-Times-Medium-R-Normal–*-20-*-*-P-*-*-*
DVWSVT0G03CK00GG0001QZZZZ02A000	-*-Terminal-Medium-R-Narrow–*-140-*-*-C-*-DEC-DECtech
DVWSVT0G03CK00GG0001UZZZZ02A000	-*-Terminal-Medium-R-Narrow–*-140-*-*-C-*-*-*
DVWSVT0G03CK00PG0001QZZZZ02A000	-*-Terminal-Bold-R-Narrow–*-140-*-*-C-*-DEC-DECtech
DVWSVT0G03CK00PG0001UZZZZ02A000	-*-Terminal-Bold-R-Narrow–*-140-*-*-C-*-*-*
DVWSVT0G05AK00GG0001QZZZZ02A000	-*-Terminal-Medium-R-Narrow–*-180-*-*-C-*-DEC-DECtech
DVWSVT0G05AK00GG0001UZZZZ02A000	-*-Terminal-Medium-R-Narrow–*-180-*-*-C-*-*-*
DVWSVT0G05AK00PG0001QZZZZ02A000	-*-Terminal-Bold-R-Narrow–*-180-*-*-C-*-DEC-DECtech
DVWSVT0G05AK00PG0001UZZZZ02A000	-*-Terminal-Bold-R-Narrow–*-180-*-*-C-*-*-*
DVWSVT0I03WK00GG0001QZZZZ02A000	-*-Terminal-Medium-R-Normal–*-140-*-*-C-*-DEC-DECtech
DVWSVT0I03WK00PG0001QZZZZ02A000	-*-Terminal-Bold-R-Normal–*-140-*-*-C-*-DEC-DECtech
DVWSVT0J05AK00GG0001UZZZZ02A000	-*-Terminal-Medium-R-Normal–*-180-*-*-C-*-*-*
DVWSVT0J05AK00PG0001UZZZZ02A000	-*-Terminal-Bold-R-Normal–*-180-*-*-C-*-*-*
DVWSVT0K05AK00GG0001QZZZZ02A000	-*-Terminal-Medium-R-Normal–*-180-*-*-C-*-DEC-DECtech
DVWSVT0K05AK00GG0001UZZZZ02A000	-*-Terminal-Medium-R-Normal–*-180-*-*-C-*-*-*

Table E–1 (Cont.) VWS and DECwindows Fonts

VWS Font	Nearest DECwindows Equivalent
DVWSVT0K05AK00PG0001QZZZZ02A000	-*-Terminal-Bold-R-Normal–*-180-*-*-C-*-DEC-DECtech
DVWSVT0K05AK00PG0001UZZZZ02A000	-*-Terminal-Bold-R-Normal–*-180-*-*-C-*-*-*
DVWSVT0N03CK00GG0001QZZZZ02A000	-*-Terminal-Medium-R-Wide–*-140-*-*-C-*-DEC-DECtech
DVWSVT0N03CK00GG0001UZZZZ02A000	-*-Terminal-Medium-R-Wide–*-140-*-*-C-*-*-*
DVWSVT0N03CK00PG0001QZZZZ02A000	-*-Terminal-Bold-R-Wide–*-140-*-*-C-*-DEC-DECtech
DVWSVT0N03CK00PG0001UZZZZ02A000	-*-Terminal-Bold-R-Wide–*-140-*-*-C-*-*-*
DVWSVT0N05AK00GG0001UZZZZ02A000	-*-Terminal-Medium-R-Normal–*-180-*-*-C-*-*-*
DVWSVT0N05AK00PG0001UZZZZ02A000	-*-Terminal-Bold-R-Normal–*-180-*-*-C-*-*-*
DVWSVT0N06OK00GG0001QZZZZ02A000	-*-Terminal-Medium-R-Narrow–*-280-*-*-C-*-DEC-DECtech
DVWSVT0N06OK00GG0001UZZZZ02A000	-*-Terminal-Medium-R-Narrow–*-280-*-*-C-*-*-*
DVWSVT0N06OK00PG0001QZZZZ02A000	-*-Terminal-Bold-R-Narrow–*-280-*-*-C-*-DEC-DECtech
DVWSVT0N06OK00PG0001UZZZZ02A000	-*-Terminal-Bold-R-Narrow–*-280-*-*-C-*-*-*
DVWSVT0N0AKK00GG0001UZZZZ02A000	-*-Terminal-Medium-R-Narrow–*-360-*-*-C-*-*-*
DVWSVT0N0AKK00PG0001UZZZZ02A000	-*-Terminal-Bold-R-Narrow–*-360-*-*-C-*-*-*
DVWSVT0R03WK00GG0001QZZZZ02A000	-*-Terminal-Medium-R-Double–*-140-*-*-C-*-DEC-DECtech
DVWSVT0R03WK00PG0001QZZZZ02A000	-*-Terminal-Bold-R-Double–*-140-*-*-C-*-DEC-DECtech
DVWSVT0R07SK00GG0001QZZZZ02A000	-*-Terminal-Medium-R-Normal–*-280-*-*-C-*-DEC-DECtech
DVWSVT0R07SK00PG0001QZZZZ02A000	-*-Terminal-Bold-R-Normal–*-280-*-*-C-*-DEC-DECtech
DVWSVT0V05AK00GG0001UZZZZ02A000	-*-Terminal-Medium-R-Double–*-180-*-*-C-*-*-*
DVWSVT0V05AK00PG0001UZZZZ02A000	-*-Terminal-Bold-R-Double–*-180-*-*-C-*-*-*
DVWSVT0V0AKK00GG0001UZZZZ02A000	-*-Terminal-Medium-R-Normal–*-360-*-*-C-*-*-*
DVWSVT0V0AKK00PG0001UZZZZ02A000	-*-Terminal-Bold-R-Normal–*-360-*-*-C-*-*-*
DVWSVT1G03CK00GG0001UZZZZ02A000	-*-Terminal-Medium-R-Narrow–*-120-*-*-C-*-*-*
DVWSVT1G05AK00GG0001UZZZZ02A000	-*-Terminal-Medium-R-Narrow–*-180-*-*-C-*-*-*
DVWSVT1I03WK00GG0001UZZZZ02A000	-*-Terminal-Medium-R-Normal–*-140-*-*-C-*-*-*
DVWSVT1J05AK00GG0001UZZZZ02A000	-*-Terminal-Medium-R-Normal–*-180-*-*-C-*-*-*
DVWSVT1J05AK00PG0001UZZZZ02A000	-*-Terminal-Bold-R-Normal–*-180-*-*-C-*-*-*
DVWSVT1K05AK00GG0001UZZZZ02A000	-*-Terminal-Medium-R-Normal–*-180-*-*-C-*-*-*
DWSMENU003WK00GG0001UZZZZ02B000	-*-Times-Medium-R-Normal–*-140-*-*-P-*-*-*
DWSMENU003WK00PG0001UZZZZ02B000	-*-Times-Bold-R-Normal–*-140-*-*-P-*-*-*
DWSMENU003WK01GG0001UZZZZ02B000	-*-Times-Medium-I-Normal–*-140-*-*-P-*-*-*
DWYSIFPJ03CK00GG0001UZZZZ02A000	-*-Courier-Medium-R-Normal–*-120-*-*-M-*-*-*
DWYSIFPJ03CK00PG0001UZZZZ02A000	-*-Courier-Bold-R-Normal–*-120-*-*-M-*-*-*
DWYSIFPJ03CK02GG0001UZZZZ02A000	-*-Courier-Medium-I-Normal–*-120-*-*-M-*-*-*
DWYSIFPL02SK00GG0001UZZZZ02A000	-*-Courier-Medium-R-Normal–*-100-*-*-M-*-*-*
DWYSINS001OK00GG0001UZZZZ02A000	-*-Times-Medium-R-Normal–*-120-*-*-P-*-*-*
DWYSINS001OK00PG0001UZZZZ02A000	-*-Times-Bold-R-Normal–*-120-*-*-P-*-*-*
DWYSINS001OK01GG0001UZZZZ02A000	-*-Times-Medium-I-Normal–*-120-*-*-P-*-*-*

Table E–1 (Cont.) VWS and DECwindows Fonts

VWS Font	Nearest DECwindows Equivalent
DWYSINS0028K00GG0001UZZZZ02A000	-*-Times-Medium-R-Normal–*-80-*-*-P-*-*-*
DWYSINS0028K00PG0001UZZZZ02A000	-*-Times-Bold-R-Normal–*-80-*-*-P-*-*-*
DWYSINS0028K01GG0001UZZZZ02A000	-*-Times-Medium-I-Normal–*-80-*-*-P-*-*-*
DWYSINS002SK00GG0001UZZZZ02A000	-*-Times-Medium-R-Normal–*-100-*-*-P-*-*-*
DWYSINS002SK00PG0001UZZZZ02A000	-*-Times-Bold-R-Normal–*-100-*-*-P-*-*-*
DWYSINS002SK01GG0001UZZZZ02A000	-*-Times-Medium-I-Normal–*-100-*-*-P-*-*-*
DWYSINS003CK00GG0001UZZZZ02A000	-*-Times-Medium-R-Normal–*-120-*-*-P-*-*-*
DWYSINS003CK00PG0001UZZZZ02A000	-*-Times-Bold-R-Normal–*-120-*-*-P-*-*-*
DWYSINS003CK01GG0001UZZZZ02A000	-*-Times-Medium-I-Normal–*-120-*-*-P-*-*-*
DWYSINS003WK00GG0001UZZZZ02A000	-*-Times-Medium-R-Normal–*-140-*-*-P-*-*-*
DWYSINS003WK00PG0001UZZZZ02A000	-*-Times-Bold-R-Normal–*-140-*-*-P-*-*-*
DWYSINS003WK01GG0001UZZZZ02A000	-*-Times-Medium-I-Normal–*-140-*-*-P-*-*-*
DWYSINS0050K00GG0001UZZZZ02A000	-*-Times-Medium-R-Normal–*-180-*-*-P-*-*-*
DWYSINS0050K00PG0001UZZZZ02A000	-*-Times-Bold-R-Normal–*-180-*-*-P-*-*-*
DWYSINS0050K01GG0001UZZZZ02A000	-*-Times-Medium-I-Normal–*-180-*-*-P-*-*-*
DWYSINS006OK00GG0001UZZZZ02A000	-*-Times-Medium-R-Normal–*-240-*-*-P-*-*-*
DWYSINS006OK00PG0001UZZZZ02A000	-*-Times-Bold-R-Normal–*-240-*-*-P-*-*-*
DWYSINS006OK01GG0001UZZZZ02A000	-*-Times-Medium-I-Normal–*-240-*-*-P-*-*-*
DWYSINS00A0K00GG0001UZZZZ02A000	-*-Times-Medium-R-Normal–*-180-*-*-P-*-*-*
DWYSINS00A0K00PG0001UZZZZ02A000	-*-Times-Bold-R-Normal–*-180-*-*-P-*-*-*
DWYSINS00A0K01GG0001UZZZZ02A000	-*-Times-Medium-I-Normal–*-180-*-*-P-*-*-*
DWYSISS001OK00GG0001UZZZZ02A000	-*-Helvetica-Medium-R-Normal–*-120-*-*-P-*-*-*
DWYSISS001OK00PG0001UZZZZ02A000	-*-Helvetica-Bold-R-Normal–*-120-*-*-P-*-*-*
DWYSISS001OK02GG0001UZZZZ02A000	-*-Helvetica-Medium-R-Normal–*-120-*-*-P-*-*-*
DWYSISS0028K00GG0001UZZZZ02A000	-*-Helvetica-Medium-R-Normal–*-80-*-*-P-*-*-*
DWYSISS0028K00PG0001UZZZZ02A000	-*-Helvetica-Bold-R-Normal–*-80-*-*-P-*-*-*
DWYSISS0028K02GG0001UZZZZ02A000	-*-Helvetica-Medium-R-Normal–*-80-*-*-P-*-*-*
DWYSISS002SK00GG0001UZZZZ02A000	-*-Helvetica-Medium-R-Normal–*-100-*-*-P-*-*-*
DWYSISS002SK00PG0001UZZZZ02A000	-*-Helvetica-Bold-R-Normal–*-100-*-*-P-*-*-*
DWYSISS002SK02GG0001UZZZZ02A000	-*-Helvetica-Medium-R-Normal–*-100-*-*-P-*-*-*
DWYSISS003CK00GG0001UZZZZ02A000	-*-Helvetica-Medium-R-Normal–*-120-*-*-P-*-*-*
DWYSISS003CK00PG0001UZZZZ02A000	-*-Helvetica-Bold-R-Normal–*-120-*-*-P-*-*-*
DWYSISS003CK02GG0001UZZZZ02A000	-*-Helvetica-Medium-R-Normal–*-120-*-*-P-*-*-*
DWYSISS003WK00GG0001UZZZZ02A000	-*-Helvetica-Medium-R-Normal–*-140-*-*-P-*-*-*
DWYSISS003WK00PG0001UZZZZ02A000	-*-Helvetica-Bold-R-Normal–*-140-*-*-P-*-*-*
DWYSISS003WK02GG0001UZZZZ02A000	-*-Helvetica-Medium-R-Normal–*-140-*-*-P-*-*-*
DWYSISS0050K00GG0001UZZZZ02A000	-*-Helvetica-Medium-R-Normal–*-180-*-*-P-*-*-*
DWYSISS0050K00PG0001UZZZZ02A000	-*-Helvetica-Bold-R-Normal–*-180-*-*-P-*-*-*

Table E–1 (Cont.) VWS and DECwindows Fonts

VWS Font	Nearest DECwindows Equivalent
DWYSISS0050K02GG0001UZZZZ02A000	-*-Helvetica-Medium-R-Normal–*-180-*-*-P-*-*-*
DWYSISS006OK00GG0001UZZZZ02A000	-*-Helvetica-Medium-R-Normal–*-240-*-*-P-*-*-*
DWYSISS006OK00PG0001UZZZZ02A000	-*-Helvetica-Bold-R-Normal–*-240-*-*-P-*-*-*
DWYSISS006OK02GG0001UZZZZ02A000	-*-Helvetica-Medium-R-Normal–*-240-*-*-P-*-*-*
DWYSISS00A0K00GG0001UZZZZ02A000	-*-Helvetica-Medium-R-Normal–*-180-*-*-P-*-*-*
DWYSISS00A0K00PG0001UZZZZ02A000	-*-Helvetica-Bold-R-Normal–*-180-*-*-P-*-*-*
DWYSISS00A0K02GG0001UZZZZ02A000	-*-Helvetica-Medium-R-Normal–*-180-*-*-P-*-*-*
RCOURIRG03WK00GG0001QZZZZ02B000	-*-Terminal-Medium-R-Narrow–*-140-*-*-C-*-DEC-DECtech
RCOURIRG03WK00GG0001UZZZZ02B000	-*-Terminal-Medium-R-Narrow–*-140-*-*-C-*-*-*
RCOURIRG03WK00PG0001QZZZZ02B000	-*-Terminal-Bold-R-Narrow–*-140-*-*-C-*-DEC-DECtech
RCOURIRG03WK00PG0001UZZZZ02B000	-*-Terminal-Bold-R-Narrow–*-140-*-*-C-*-*-*
RCOURIRI03WK00GG0001QZZZZ02B000	-*-Terminal-Medium-R-Normal–*-140-*-*-C-*-DEC-DECtech
RCOURIRI03WK00GG0001UZZZZ02B000	-*-Terminal-Medium-R-Normal–*-140-*-*-C-*-*-*
RCOURIRI03WK00PG0001QZZZZ02B000	-*-Terminal-Bold-R-Normal–*-140-*-*-C-*-DEC-DECtech
RCOURIRI03WK00PG0001UZZZZ02B000	-*-Terminal-Bold-R-Normal–*-140-*-*-C-*-*-*
RCOURIRN03WK00GG0001QZZZZ02B000	-*-Terminal-Medium-R-Wide–*-140-*-*-C-*-DEC-DECtech
RCOURIRN03WK00GG0001UZZZZ02B000	-*-Terminal-Medium-R-Wide–*-140-*-*-C-*-*-*
RCOURIRN03WK00PG0001QZZZZ02B000	-*-Terminal-Bold-R-Wide–*-140-*-*-C-*-DEC-DECtech
RCOURIRN03WK00PG0001UZZZZ02B000	-*-Terminal-Bold-R-Wide–*-140-*-*-C-*-*-*
RCOURIRN07SK00GG0001QZZZZ02B000	-*-Terminal-Medium-R-Narrow–*-280-*-*-C-*-DEC-DECtech
RCOURIRN07SK00GG0001UZZZZ02B000	-*-Terminal-Medium-R-Narrow–*-280-*-*-C-*-*-*
RCOURIRN07SK00PG0001QZZZZ02B000	-*-Terminal-Bold-R-Narrow–*-280-*-*-C-*-DEC-DECtech
RCOURIRN07SK00PG0001UZZZZ02B000	-*-Terminal-Bold-R-Narrow–*-280-*-*-C-*-*-*
RCOURIRR03WK00GG0001QZZZZ02B000	-*-Terminal-Medium-R-Double–*-140-*-*-C-*-DEC-DECtech
RCOURIRR03WK00GG0001UZZZZ02B000	-*-Terminal-Medium-R-Double–*-140-*-*-C-*-*-*
RCOURIRR03WK00PG0001QZZZZ02B000	-*-Terminal-Bold-R-Double–*-140-*-*-C-*-DEC-DECtech
RCOURIRR03WK00PG0001UZZZZ02B000	-*-Terminal-Bold-R-Double–*-140-*-*-C-*-*-*
RCOURIRR07SK00GG0001QZZZZ02B000	-*-Terminal-Medium-R-Normal–*-280-*-*-C-*-DEC-DECtech
RCOURIRR07SK00GG0001UZZZZ02B000	-*-Terminal-Medium-R-Normal–*-280-*-*-C-*-*-*
RCOURIRR07SK00PG0001QZZZZ02B000	-*-Terminal-Bold-R-Normal–*-280-*-*-C-*-DEC-DECtech
RCOURIRR07SK00PG0001UZZZZ02B000	-*-Terminal-Bold-R-Normal–*-280-*-*-C-*-*-*

F Color Conversion Routines

```
#module Color_Conversion "V01.0-000"
/*
 *
 * Facility:
 *
 *      Color_cConversion
 *
 * Abstract:
 *
 *      General usage conversion routines for HLS and HSV and RGB.
 *      The routines use the UIS conventions for Hue in HLS (centered
 *      at RED instead of BLUE).
 *
 * Environment:
 *
 *      VMS/VAX-C
 *
 * Entry Points:
 *
 *         All values are passed by reference, and F Floating.
 *
 *         Hue is always from 0 to 360, unless Saturation = 0
 *         in which case it is ignored as input, and returns a
 *         -1.0 as output.
 *
 *         All other values are expressed as a percentage from
 *         0.0 to 1.0 inclusive.
 *
 *     HSV_to_RGB( Hue, Saturation, Value, Red, Green, Blue)
 *
 *         Hue.rf.r        = Hue, from 0 to 360
 *         Saturation.rf.r = Saturation, from 0 to 1
 *         Value.rf.r      = Value, from 0 to 1
 *         Red.rf.w        = Red, from 0 to 1
 *         Green.rf.w      = Green, from 0 to 1
 *         Blue.rf.w       = Blue, from 0 to 1
 *
 *     HLS_to_RGB( Hue, Lightness, Saturation, Red, Green, Blue)
 *
 *         Hue.rf.r        = Hue, from 0 to 360
 *         Lightness.rf.r  = Lightness, from 0 to 1
 *         Saturation.rf.r = Saturation, from 0 to 1
 *         Red.rf.w        = Red, from 0 to 1
 *         Green.rf.w      = Green, from 0 to 1
 *         Blue.rf.w       = Blue, from 0 to 1
 *
 *     RGB_to_HSV( Red, Green, Blue, Hue, Saturation, Value)
 *
 *         Red.rf.r        = Red, from 0 to 1
 *         Green.rf.r      = Green, from 0 to 1
 *         Blue.rf.r       = Blue, from 0 to 1
 *         Hue.rf.w        = Hue, from 0 to 360, -1.0 if Saturation = 0.0
 *         Saturation.rf.w = Saturation, from 0 to 1
 *         Value.rf.w      = Value, from 0 to 1
 *
 *     RGB_to_HLS( Red, Green, Blue, Hue, Lightness, Saturation)
 *
 *         Red.rf.r        = Red, from 0 to 1
 *         Green.rf.r      = Green, from 0 to 1
 *         Blue.rf.r       = Blue, from 0 to 1
```

```
 *         Hue.rf.w        = Hue, from 0 to 360, -1.0 if Saturation = 0.0
 *         Lightness.rf.w  = Lightness, from 0 to 1
 *         Saturation.rf.w = Saturation, from 0 to 1
 *
 * Modification History:
 *
 */

extern void  HSV_to_RGB();
extern void  HLS_to_RGB();
extern void  RGB_to_HSV();
extern void  RGB_to_HLS();
static float VALUE();

#define mine(x,y,z)  ( ( ( (x < y) ? x:y ) < z ) ? ( (x < y) ? x:y ) : z )
#define max3(x,y,z)  ( ( ( (x > y) ? x:y ) > z ) ? ( (x > y) ? x:y ) : z )

void RGB_to_HSV ( Red, Green, Blue, Hue, Saturation, Value)

float *Red, *Green, *Blue, *Hue, *Saturation, *Value;

/*
 *  RGB_to_HSV - converts RGB values as input into HSV as output.
 *
 *  All parameters are passed by reference and are floating point.
 *
 *  RGB are read only, HSV are write only.
 *
 *  A Saturation of 0 returns -1.0 as the Hue
 *
 */

{
  float max_value, min_value,
        color_span,
        red_content, green_content, blue_content;

 /*
  *  Get the max and min values for RGB
  *
  */

  max_value = max3( *Red, *Green, *Blue);
  min_value = mine( *Red, *Green, *Blue);

 /*
  *  Value = max_value
  *
  */

  *Value = max_value;

 /*
  *  Now compute Saturation
  *
  */

  if (max_value != 0.0)
    {
      *Saturation = (max_value - min_value) / max_value;
    }
  else
    {
      *Saturation = 0.0;
    }

 /*
  *  And finally the Hue
  *
  */

  if (*Saturation != 0.0)
    {

      color_span     =  max_value - min_value;
```

```
      red_content   = (max_value - *Red)   / color_span;
      green_content = (max_value - *Green) / color_span;
      blue_content  = (max_value - *Blue)  / color_span;

      if (*Red = max_value)
        {
          *Hue = blue_content - green_content;
        }
      else
        {
          if (*Green = max_value)
            {
              *Hue = 2.0 + red_content - blue_content;
            }
          else
            {
              *Hue = 4.0 + green_content - red_content;
            }
        }

      *Hue = *Hue * 60.0;
      if (*Hue < 0.0) *Hue = *Hue + 360.0;

    }
  else
    {

     /*
      *  A Saturation of zero results in UIS$C_COLOR_UNDEFINED which
      *  is a -1.0 in floating point
      *
      */

      *Hue = -1.0;
    }
}

void HSV_to_RGB( Hue, Saturation, Value, Red, Green, Blue)

float  *Hue, *Saturation, *Value, *Red, *Green, *Blue;

/*
 *  HSV_to_RGB - converts HSV values as input into RGB as output.
 *
 *  All parameters are passed by reference and are floating point.
 *
 *  HSV are read only, RGB are write only.
 *
 */

{
  int   integer_hue;
  float fractional_hue, p, q, t, h;

  if (*Saturation == 0)
    {
     /*
      *  Strictly speaking, a Saturation of 0 means that Hue should
      *  contain UIS$C_COLOR_UNDEFINED, but the standard industry
      *  practice is to ignore Hue if the Saturation is 0.  The UIS
      *  call will signal an error if Hue is not -1.0
      *
      *  This is the anachromatic case, where R = G = B = Value.
      */

      *Red   = *Value;
      *Green = *Value;
      *Blue  = *Value;
    }
  else
    {

      h = *Hue; /* A local copy of the Hue */
```

```
        while (h <    0.0) h = h + 360.0; /* Need a positive angle */
        while (h >= 360.0) h = h - 360.0; /* Need it from 0-360, make 360 = 0 */

        h = h / 60.0; /* Make it a value between 0 - 5.999999 */

        integer_hue = h; /* Truncate Hue to a integer from 0-5 */
        fractional_hue = *Hue - (float)integer_hue; /* Get the fractional part */

        p = *Value * (1.0 - *Saturation);
        q = *Value * (1.0 - (*Saturation * fractional_hue));
        t = *Value * (1.0 - (*Saturation * (1.0 - fractional_hue)));

        switch (integer_hue)
          {
            case 0:
            case 6:

              *Red   = *Value;
              *Green = t;
              *Blue  = p;
              break;

            case 1:

              *Red   = q;
              *Green = *Value;
              *Blue  = p;
              break;

            case 2:

              *Red   = p;
              *Green = *Value;
              *Blue  = t;
              break;

            case 3:

              *Red   = p;
              *Green = q;
              *Blue  = *Value;
              break;

            case 4:

              *Red   = t;
              *Green = p;
              *Blue  = *Value;
              break;

            case 5:

              *Red   = *Value;
              *Green = p;
              *Blue  = q;
              break;
          }
      }
}

void RGB_to_HLS ( Red, Green, Blue, Hue, Lightness, Saturation)

float *Red, *Green, *Blue, *Hue, *Lightness, *Saturation;
```

```
/*
 *  RGB_to_HLS - converts RGB values as input into HLS as output.
 *
 *  All parameters are passed by reference and are floating point.
 *
 *  RGB are read only, HLS are write only.
 *
 *  A Saturation of 0 returns -1.0 as the Hue, otherwise Hue is from 0 - 360
 *
 *
 *  ** NOTE **  This routine follows the UIS convention of RED at 0°
 *    instead of the industry standard convention of locating
 *    BLUE at 0°.  To convert to industry standards, add 120°
 *    to the Hue result.
 *
 */

{
  float max_value, min_value,
        color_span,
        red_content, green_content, blue_content;

 /*
  *  Get the max and min values for RGB
  *
  */

  max_value = max3( *Red, *Green, *Blue);
  min_value = mine( *Red, *Green, *Blue);

 /*
  *  Compute Lightness
  *
  */

  *Lightness = (max_value + min_value) / 2;

  if (max_value = min_value)
    {
     /*
      *  This is Red = Green = Blue: anachromatic
      *
      *  A Saturation of zero results in UIS$C_COLOR_UNDEFINED for Hue
      *  which is a -1.0 in floating point.
      *
      */

     *Saturation =  0.0;
     *Hue        = -1.0;
    }
  else
    {
     color_span = max_value - min_value;

     /*
      *  Compute Saturation
      *
      */

     if (*Lightness < 0.5)
       {
         *Saturation = color_span / (max_value + min_value);
       }
     else
       {
         *Saturation = color_span / (2.0 - max_value - min_value);
       }

     /*
      *  Compute Hue
      *
      */
```

```
      red_content   = (max_value - *Red)   / color_span;
      green_content = (max_value - *Green) / color_span;
      blue_content  = (max_value - *Blue)  / color_span;

      if (*Red = max_value)
        {
          *Hue = blue_content - green_content;
        }
      else
        {
          if (*Green = max_value)
            {
              *Hue = 2.0 + red_content - blue_content;
            }
          else
            {
              *Hue = 4.0 + green_content - red_content;
            }
        }

      *Hue = *Hue * 60.0;

      if (*Hue < 0.0) *Hue = *Hue + 360.0;

    }
}

void HLS_to_RGB( Hue, Lightness, Saturation, Red, Green, Blue)

float  *Hue, *Lightness, *Saturation, *Red, *Green, *Blue;

/*
 *  HLS_to_RGB - converts HSV values as input into RGB as output.
 *
 *  All parameters are passed by reference and are floating point.
 *
 *  HLS are read only, RGB are write only.
 *
 *  ** NOTE **  This routine follows the UIS convention of RED at 0°
 *    instead of the industry standard convention of locating
 *    BLUE at 0°.  To convert to industry standards, the input
 *    Hue should have 120° subtracted from it.
 *
 */

{
  float m1, m2;

  if (*Lightness < 0.5)
   {
     m2 = *Lightness * (1.0 + *Saturation);
   }
  else
   {
     m2 = *Lightness + *Saturation - (*Lightness * *Saturation);
   }

  m1 = (*Lightness * 2.0) - m2;

  if (*Saturation == 0)
    {
     /*
      *  Strictly speaking, a Saturation of 0 means that Hue should
      *  contain UIS$C_COLOR_UNDEFINED, but the standard industry
      *  practice is to ignore Hue if the Saturation is 0.  The UIS
      *  routine will signal an error if Hue is not -1.0, this will not.
      *
      *  This is the anachromatic case, where R = G = B = Lightness.
      */
```

```
      *Red   = *Lightness;
      *Green = *Lightness;
      *Blue  = *Lightness;
    }
  else
    {
      *Red   = VALUE( m1, m2, *Hue + 120.0);
      *Green = VALUE( m1, m2, *Hue);
      *Blue  = VALUE( m1, m2, *Hue - 120.0);
    }
}

static float VALUE( n1, n2, Hue)

float n1, n2, Hue;

{

  while (Hue <    0.0) Hue += 360.0; /* Need a positive angle */
  while (Hue >= 360.0) Hue -= 360.0; /* Need it from 0-360 */

  if (Hue < 60.0) return(n1 + (((n2 - n1) * Hue) / 60.0));
  else
    if (Hue < 180.0) return(n2);
    else
      if (Hue < 240.0) return(n1 + (((n2 - n1) * (240.0 - Hue)) / 60.0));
      else return (n1);
}

/* End of color_conversion module */
```

G Colormap Example

The following C code example illustrates a model to allocate X11 colors that can have logical operations performed on the pixel values returned. This enables such functions as COMPLEMENT mode.

```
/*
 *  The following module will allocate X11 colors that can have logical
 *  operations done on the pixel values returned - allowing such things
 *  as COMPLEMENT mode.
 *
 */

#include <SSDEF>
#include <decw$include:Xlib.h>
#include <decw$include:Xutil.h>


/*
 *  This a virtual colormap structure.  It contains the information needed
 *  for X11 use.
 *
 */
typedef struct _vmap_struct {
    short    int  type;
    short    int  ref_count;
    long     int  size;
    Colormap id;
    long     int contig;
    unsigned long int *masks;
    long     int num_planes;
    unsigned long int pixel;
    long int num_colors;
    unsigned long int or_mask;
    long int slot_size;
    unsigned long indices[2];
    } vmap_struct ;

#define XUIS__CMAP_S_SIZE sizeof( vmap_struct)
```

Colormap Example

```
/*
 *  Routine:        create_virtual_colormap
 *
 *  Description:    Allocate colors for X11 so that logical operations can
 *     be done on the pixels.
 *
 *  Inputs:         Virtual Colormap size
 *
 *  Outputs:        A pointer to a structure containing the colormap
 *     information.  If the pointer is zero, the routine
 *     failed.
 *
 */

unsigned long int xuis_create_color_map( display_id, vmap_size)

Display  *display_id;
long int *vmap_size;

{

  Screen      *screen_id;
  Visual      *visual_id;
  vmap_struct *pCmap;

  long int    status,
              cmapSize = XUIS__CMAP_S_SIZE,
              planes,
              max,
              color,
              loop,
              cbits,
              black,
              white;

  char        *temp;

 /*
  *  Initialize to no map
  *
  */
  pCmap = 0;

 /*
  *  UIS colormaps are emulated by allocating color cells for exclusive use.
  *
  *  In reality, UIS rounds the size of the colormap up to a power of two so
  *  that all logical operators on the pixels will work.  It then allocates
  *  the map as a multiple of this rounded size.  The result is a color map
  *  which starts at a power of 2, so that the first entry's low-order bits are
  *  clear, and there is a fixed "offset" that is composed of the upper bits.
  *  all the operations mask the upper bits and operate on the low-order bits.
  *
  *  For DECwindows, it's a little tricky.  To be able to do logical operations
  *  (needed for things like complement mode) that rely on the current value
  *  of a pixel, we need to allocate colors so that the logical operation will
  *  yield a value that is valid (allocated to us).  To do this we ask for
  *  n PLANES with a single color.  Logical operations are done on the PLANE
  *  BITS and the PIXEL value is the constant (that is, pretty much the
  *  reverse of UIS).  We'll create a virtual colormap that is an array of the
  *  PIXEL value with every permutation of the PLANE MASK bits arranged so
  *  that a complement of the zero'th entry results in the pixel value in the
  *  LAST entry.
  *
  *  Note that if the system is Bitonal (single plane buffer), the array will
  *  be filled with the white pixel/black pixel values (which "should"
  *  complement to each other!).
  *
  */
  if ( (max = *vmap_size - 1) < 1)
    {
     /*
      *  Under UIS this would signal a error, the size must be greater
```

```
     *  than 1.  However, this will simple bump the plane count to 1.
     *
     */
     planes = 1;
   }
 else
   {
    /*
     *  Figure out how may planes by shifting right and incrementing the count
     *  until the value is zero.
     *
     */
     for (planes = 0; max > 0; max >>= max)
       {
         planes += 1;
       }
   }

/*
 *  Add the size of the colormap to the colormap structure (in LONGS)
 *  AND add that number of LONGWORDS for the plane arrays.  There will be a
 *  pointer to the masks which is really within the block.
 *
 */
 cmapSize += (1 << planes) * 8;

/*
 *  Create the colormap structure and initialize it.
 *
 */
 if ((status = lib$get_vm( &cmapSize, &pCmap)) == SS$_NORMAL)
   {

     pCmap->id = 0;    /* No ID yet */

     pCmap->size = cmapSize;  /* Make sure the structure has a size */
     pCmap->ref_count = 0;  /* Reference count of ZERO */

     pCmap->num_planes = planes; /* Set the plane count  */
     pCmap->num_colors = 1;  /* Use 1 color and n planes */
     pCmap->pixel = 0;    /* Clear the PIXEL cell */
     pCmap->contig = 0;  /* They do not have to be contiguous */
     pCmap->slot_size = 1 << planes; /* VMAP slot size */

    /*
     *  Calculate the address of the mask array area
     *
     */
     temp = pCmap;
     temp += XUIS__CMAP_S_SIZE + ((1 << planes) * 4);
     pCmap->masks = temp;
   }
 else
   {
    /*
     *  Error getting the virtual memory
     *
     */
     lib$signal( status);
     return ( 0);
   }

/*
 *  Get the default screen and visual for the display
 *
 */
 screen_id = XDefaultScreenOfDisplay( display_id);
 visual_id = XDefaultVisualOfScreen( screen_id);
```

Colormap Example

```
/*
 *  Use the default colormap, and allocate color cells for
 *  exclusive use...
 *
 */
pCmap->id = XDefaultColormapOfScreen( screen_id);

if ( (visual_id)->class == PseudoColor )
  {
   /*
    *  This is ONLY done for psuedo color.  We will treat
    *  all other visuals as bitonal.  This can change later,
    *  since this should also work for direct color and static
    *  grey.
    *
    */
    if ((status = XAllocColorCells( display_id,
                                    pCmap->id,
                                    pCmap->contig,
                                    pCmap->masks,
                                    pCmap->num_planes,
                                    &pCmap->pixel,
                                    pCmap->num_colors)) == 0)
      {
        return (0);
      }
    else
      {
       /*
        *  Now, we need to build our virtual colormap.
        *  This is done by building an array of index
        *  values that are all the permutations of the
        *  1 pixel and the 'n' planes.  We do it such
        *  that the array is complementary.  So when we
        *  do logical operations (like to do a complement
        *  mode write) we simply do it on the plane
        *  mask portion of the index.
        *
        *  The main loop is for every entry in the colormap
        *  rounded up to the slot size.
        *
        *  Example:
        *
        *  4 colors = 2 planes + 1 color
        *
        *  Allocation returns:  PIXEL = 1, PLANES = 10, 20 (hex)
        *
        *  index        pixel value
        *
        *    0             1
        *    1            11
        *    2            21
        *    3            31
        *
        *  XORing this with the plane mask of 30, will always
        *  end up with the complement, and a valid pixel value.
        *
        */
        for (color = 0; color < pCmap->slot_size; color += 1)
          {
           /*
            *  Start with the PIXEL and clone a copy of the
            *  current index
            *
            */
            pCmap->indices[color] = pCmap->pixel;
            cbits = color;
```

```
            /*
             *  For each plane, if the low bit of the cloned
             *  copy of the index count is set, or the plane
             *  mask bit into the pixel value.  The cloned
             *  index count is then right shifted.  The result
             *  is that for each bit in the current index value,
             *  the corresponding plane mask bit is ORed into the
             *  pixel value.
             *
             */
             for (loop = 0; loop < pCmap->num_planes; loop += 1)
               {
                 if (cbits & 1)
                   {
                     pCmap->indices[color] |= (pCmap->masks)[loop];
                   }
                 cbits >>= 1;
               }
           }
       }
     /*
      *  OR all the plane mask bits together to form a
      *  single plane mask
      *
      */
      pCmap->or_mask = 0;

      for (loop = 0; loop < pCmap->num_planes; loop += 1)
        {
          pCmap->or_mask |= (pCmap->masks)[loop];
        }
    }
  else
    {
     /*
      *  Monochrome WS, use black and white pixel!  Propagate it
      *  as the default.
      *
      */
      black = XBlackPixelOfScreen( screen_id);
      white = XWhitePixelOfScreen( screen_id);

      for (color = 0; color < pCmap->slot_size; color += 2)
        {
          pCmap->indices[color]   = black;
          pCmap->indices[color+1] = white;
        }
      pCmap->or_mask = black | white;
    }

  return ( pCmap);

}
```

H Mapping UIS Writing Modes to X11 Attributes

This appendix provides information on how to map device-independent and device-dependent writing modes to X11 attributes.

NOTE: **To document its writing mode logical operations, UIS uses a notation that is read right to left with the logical operation first. For example, read DSNA = Destination Source Not And as follows:**

(NOT Source) AND Destination

where:

D = Destination
S = Source
A = AND
N = NOT
O = OR
X = XOR

Both the UIS notation and the actual operations are explained in the following section.

UIS$C_MODE_TRAN

Transparent mode uses the function GXnoop. The logical UIS operation is D (dst). The fill type and foreground/background have no meaning.

function =	GXnoop
fill type =	FillSolid
f_pixel =	foreground
b_pixel =	background

UIS$C_MODE_COPY

Copy mode is the function GXcopy. The logical UIS operation is S (src). The fill type is Opaque Stippled and the foreground and background are normal.

function =	GXcopy
fill type =	FillOpaqueStippled
f_pixel =	foreground
b_pixel =	background

UIS$C_MODE_COMP

Complement mode is the UIS function DSX (src XOR dst). With the plane mask used as the source, the colormap is set up to enable this. Thus, you can use XOR mode in Opaque Stipple fill with the plane mask as the source.

function =	GXxor
fill type =	FillOpaqueStippled
f_pixel =	plane mask
b_pixel =	plane mask

UIS$C_MODE_COPYN

This is the UIS function SN (NOT src). It is identical to "copy," except that it has the GXInverted function.

function =	GXcopyInverted
fill type =	FillOpaqueStippled
f_pixel =	foreground
b_pixel =	background

UIS$C_MODE_OVER

In UIS, Overlay mode is the function you use to write all ONES using the secondary MASK 2 as a stencil. In X11, you accomplish this by using Fill Stippled with the GXcopy function. This uses the source as a mask and writes foreground to the destination, masking off any bits not in the foreground. (These bits remain unchanged.)

function =	GXcopy
fill type =	FillStippled
f_pixel =	foreground
b_pixel =	background

UIS$C_MODE_OVERN

This is the inverse function of Overlay mode (above). In UIS, it is described as follows: ONES use (NOT MASK_2).

function =	GXcopyInverted
fill type =	FillStippled
f_pixel =	foreground
b_pixel =	background

UIS$C_MODE_REPL

Replace mode in UIS is S (src). It is simply the copy function with opaque stipple and the correct background and foreground.

function =	GXcopy
fill type =	FillOpaqueStippled
f_pixel =	foreground
b_pixel =	background

UIS$C_MODE_REPLN

This is the inverse function of UIS$C_MODE_REPL, SN (NOT src).

function =	GXcopyInverted
fill type =	FillOpaqueStippled
f_pixel =	foreground
b_pixel =	background

UIS$C_MODE_ERAS

In UIS, this would be equivalent to write ZEROS, with the plane mask used as the virtual colormap. In X11, this is simply GXcopy with solid fill and both background and foreground set to the background pixel.

function =	GXcopy
fill type =	FillSolid
f_pixel =	background
b_pixel =	background

UIS$C_MODE_ERASN

This is the inverse of UIS$C_MODE_ERAS (ONES). The only difference is that you use the foreground pixel instead of the background pixel.

function =	GXcopy
fill type =	FillSolid
f_pixel =	foreground
b_pixel =	foreground

UIS$C_MODE_BIS

This function is DSO (src OR dst) in UIS. This is device-dependent and unless you create and use a private colormap, there is no guarantee that you will achieve the desired results.

function =	GXor
fill type =	FillStippled
f_pixel =	foreground
b_pixel =	background

UIS$C_MODE_BIC

This device-dependent mode is DSNA (NOT src AND dst) in UIS.

function =	GXandInverted
fill type =	FillStippled
f_pixel =	foreground
b_pixel =	background

UIS$C_MODE_BISN

This DSNO (NOT src OR dst) in UIS.

function =	GXorInverted
fill type =	FillStippled
f_pixel =	foreground
b_pixel =	background

UIS$C_MODE_BICN

This is DSA (src AND dst) in UIS.

function =	GXand
fill type =	FillStippled
f_pixel =	foreground
b_pixel =	background

UIS$C_MODE_XOR

This is the final device-dependent mode, DSX (src XOR dst).

function =	GXxor
fill type =	FillStippled
f_pixel =	foreground
b_pixel =	background

Index

A

B

C

D

E

G

V

W

X

Reader's Comments

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/Documentation
Digital Equipment Corporation
5 Wentworth Drive GFS/L20
Hudson, NH 03051-4929

Do Not Tear - Fold Here