DIGITAL UNIX

Writing a Graphics Device Driver and DDX for the DIGITAL UNIX X Server

Part Number: AA-R5NHA-TE

June 1997

Product Version: DIGITAL UNIX Version 4.0 or higher

This manual describes how to add support for a graphics device to the DIGITAL UNIX X Window System.

Digital Equipment Corporation
Maynard, Massachusetts

Contents

B Graphics Device ioctl Commands

C Graphics Device Driver Routines

D DDX Loadable Services Routines

Figures

Tables

About This Book

This book accompanies the X Server Developers Kit (XDK), a subset of the Device Driver Kit (DDK). This book describes the components of the DIGITAL UNIX graphics hardware architecture. It explains how to add support for a new graphics display and how to package the resulting product.

This book does not describe how to write generic X Window System device-specific code or generic UNIX® device drivers. You should have this knowledge, or access to this information, before using this kit.

Note

DIGITAL UNIX does not provide interfaces to BIOS instructions. Therefore, your hardware specifications must provide information on initializing the graphics hardware and other device-specific operations. If it does not contain this information, or if you cannot obtain this information from the hardware manufacturer, DIGITAL UNIX will not be able to support your graphics device.

The DIGITAL UNIX X Server is based on the Open Group's sample X Server (formerly distributed by the X Consortium). As the X Window System continues to evolve, interfaces between internal DIGITAL UNIX X Server components will also change. While DIGITAL makes every effort to minimize the impact of these changes, occasionally a choice needs to be made between functionality and stability. To provide customers with a full-featured product, DIGITAL may change the interfaces in future releases of DIGITAL UNIX.

Audience

This manual is for systems programmers who:

- Have a good working knowledge of the X Window System, including both conceptual and technical information
- Have a strong background in systems programming and general hardware operation

- Are familiar with graphics hardware operation, and especially with the device you want to support

Organization

This book is organized as follows:

Chapter 1	Introduction Presents an overview of the DIGITAL UNIX X Window Server architecture and how to add support for new devices to the architecture.
Chapter 2	Writing a Display Device Driver Describes how to write a device driver for a graphics board, using the generic VGA device driver as an example.
Chapter 3	Writing a DDX Describes how to write a device-dependent X (DDX) for a graphics board, using the generic DDX as a template.
Chapter 4	Writing Xserver Configuration File Entries Describes the syntax and structure of the `Xserver.conf` file and how to add support for a new graphics device or extension in this file.
Chapter 5	Building a Graphics Hardware Support Product Describes how to compile, link, and test a graphics device driver and DDX. This chapter also describes how to package the device driver and DDX for distribution to customers.
Appendix A	Graphics Device Data Structures Contains reference pages for all graphics-related data structures.
Appendix B	Graphics Device ioctl Commands Contains reference pages for all graphics-related `ioctl` commands.
Appendix C	Graphics Device Driver Routines Contains reference pages for all graphics-related routines.
Appendix D	DDX Loadable Services Routines Contains reference pages for the loadable services routines.
Appendix E	DIGITAL UNIX X Server Components Lists the components that make up the DIGITAL UNIX X Server.

Related Documentation

The printed version of the DIGITAL UNIX documentation set is color coded to help specific audiences quickly find the books that meet their needs. (You can order the printed documentation from DIGITAL.) This color coding is reinforced with the use of an icon on the spines of books. The following list describes this convention:

Audience	Icon	Color Code
General users	G	Blue
System and network administrators	S	Red
Programmers	P	Purple
Device driver writers	D	Orange
Reference page users	R	Green

Some books in the documentation set help meet the needs of several audiences. For example, the information in some system books is also used by programmers. Keep this in mind when searching for information on specific topics.

The *Documentation Overview, Glossary, and Master Index* provides information on all of the books in the DIGITAL UNIX documentation set.

Readers of this guide should be familiar with the following documents that are not part of the DIGITAL UNIX documentation set:

- Israel and Fortune, *The X Window System Server*, X Version 11, Release 5, Digital Press, 1992
- The hardware specifications for the device you want to support

The following documents contain information that pertains to writing device drivers:

- *Writing Device Drivers: Tutorial*

  This manual provides information for systems engineers who write device drivers for hardware that runs the DIGITAL UNIX operating system. Systems engineers can find information on driver concepts, device driver interfaces, kernel interfaces used by device drivers, kernel data structures, configuration of device drivers, and header files related to device drivers.

- *Writing Device Drivers: Reference*

  This manual contains descriptions of the header files, kernel support interfaces, `ioctl` commands, global variables, data structures, device

driver interfaces, and bus configuration interfaces associated with device drivers. The descriptions are formatted similarly to the DIGITAL UNIX reference pages.

- *Writing Device Drivers: Advanced Topics*

  This manual provides information on topics that are beyond the scope of the core tutorial. Systems engineers can find information on such advanced topics as kernel threads and writing device drivers in a symmetric multiprocessing (SMP) environment. The manual also contains information about writing disk device drivers.

- *System Administration*

  This manual describes how to configure, use, and maintain the DIGITAL UNIX operating system. It includes information on general day-to-day activities and tasks, changing your system configuration, and locating and eliminating sources of trouble.

  This manual is for system administrators responsible for managing the operating system. It assumes a knowledge of operating system concepts, commands, and configurations.

- *Kernel Debugging*

  This manual provides information about debugging kernels. The manual describes using the `dbx`, `kdbx`, and `kdebug` debuggers to find problems in kernel code. It also describes how to write a `kdbx` utility extension and how to create and analyze a crash dump file.

  This manual is for system administrators responsible for modifying, rebuilding, and debugging the kernel configuration. It is also for system programmers who need to debug their kernel space programs.

- *Guide to Preparing Product Kits*

  This manual describes the DIGITAL UNIX procedures for creating, installing, and managing software kits.

  This manual is intended for third-party kit developers who create software kits that can be installed using the `setld` command.

Reader's Comments

DIGITAL welcomes any comments and suggestions you have on this and other DIGITAL UNIX manuals.

You can send your comments in the following ways:

- Fax: 603-881-0120 Attn: UEG Publications, ZK03-3/Y32
- Internet electronic mail: `readers_comment@zk3.dec.com`

A Reader's Comment form is located on your system in the following location:

```
/usr/doc/readers_comment.txt
```

- Mail:

  Digital Equipment Corporation
  UEG Publications Manager
  ZK03-3/Y32
  110 Spit Brook Road
  Nashua, NH 03062-9987

  A Reader's Comment form is located in the back of each printed manual. The form is postage paid if you mail it in the United States.

Please include the following information along with your comments:

- The full title of the book and the order number. (The order number is printed on the title page of this book and on its back cover.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of DIGITAL UNIX that you are using.
- If known, the type of processor that is running the DIGITAL UNIX software.

The DIGITAL UNIX Publications group cannot respond to system problems or technical support inquiries. Please address technical questions to your local system vendor or to the appropriate DIGITAL technical support office. Information provided with the software media explains how to send problem reports to DIGITAL.

Conventions

This document uses the following conventions:

`% cat`	Boldface type in interactive examples indicates typed user input.
file	Italic (slanted) type indicates variable values, placeholders, and function argument names.
[|] { | }	In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed.

device	In syntax definitions, this typeface indicates keywords that you must type exactly as shown.
identifier	In syntax definitions, this typeface indicates variables.
. . .	In syntax definitions, a horizontal or vertical ellipsis indicates that the preceding item can be repeated one or more times.

1

Introduction

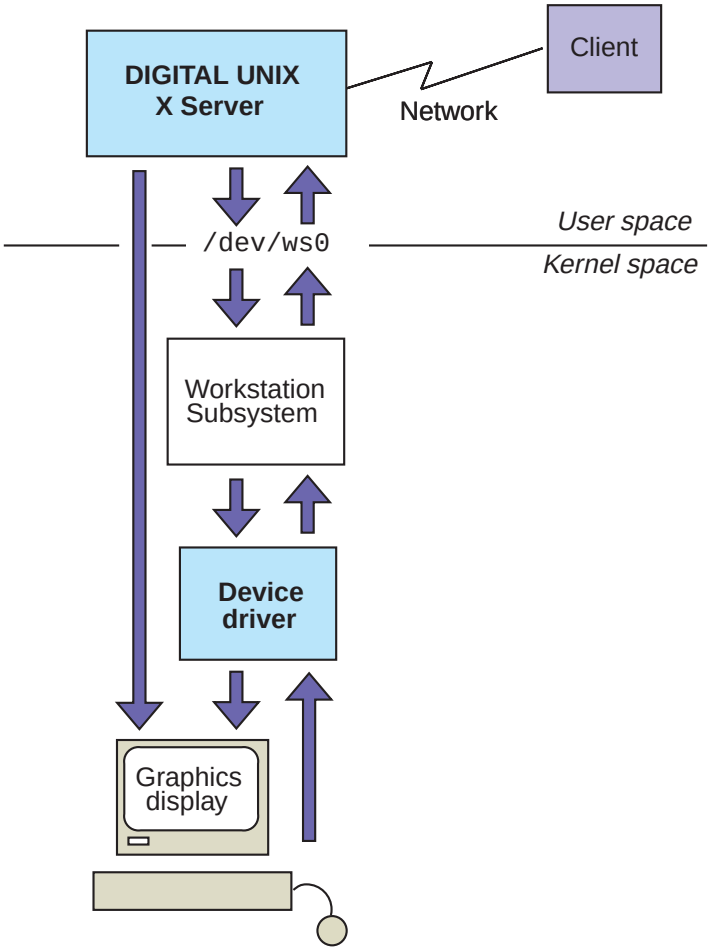
The X Server Developers Kit (XDK) is designed to help you add support for new graphics hardware to the DIGITAL UNIX X Server. The kit provides a full build environment so you can produce shared libraries and kernel code that interoperate with the DIGITAL UNIX X Window System.

DIGITAL UNIX uses a client/server architecture for communicating between an X Window System application (the client) and a graphics device connected to the system (the server). Figure 1–1 shows the client/server architecture for graphics hardware support. The Open Group defines much of this architecture — client, server, and network protocol — and the DIGITAL UNIX X Server is based on the Open Group's implementation. However, differences do exist.

Until recently, servers built from the Open Group source code (formerly distributed by the X Consortium) have provided support for specific graphics boards and preselected extensions. When building such a server, you needed to know in advance what features the server would support. You needed to decide which graphics boards to target, what extensions the server needed, and even what font renderers to use. Having selected all of these components, you would then build a single server image for a particular application. This resulted in a server that was either larger than necessary or that had limited capabilities.

The DIGITAL UNIX X Server is referred to as a loadable server because it makes use of replaceable, shared objects. Virtually all components are replaceable. The server can adapt to different hardware configurations and user needs by loading only those components that are needed, and only when they are needed.

Figure 1–1: DIGITAL UNIX Support for Graphics Hardware



ZK-1229U-AI

The architecture handles requests to send output to the graphics board, as follows:

- In user space, an X Window System application (the client) initiates the request from the user. For example, the user may want to draw a line or fill a polygon. All requests to the server go through a network interface, even if the client resides on the same physical system as the server. The network protocol defines how requests are packaged and passed between the client and server. The server unpacks the request and sends it to the appropriate device. For frame buffer devices, drawing requests usually access the frame buffer (video memory) and registers on the graphics board directly. A memory map makes

registers and memory available to the server. When the server accesses an address in the memory map, it goes through main memory to the graphics board. Some requests — such as requests to turn the screen on and off or to change the cursor shape or the color of windows — do not go directly to the device because they are not usually optimized by the hardware. These requests are handled by a common software interface. That is, they are sent to the Workstation Subsystem as `ioctl` system calls to the `/dev/ws0` device.

- In kernel space, the Workstation Subsystem invokes the appropriate device driver interface to perform the requested operation.

The architecture handles requests to accept input from the system pointer, keyboard, or other device, as follows:

- The input device notifies the device driver that an event has occurred. For example, the user has pressed a key or clicked a mouse button.
- The device driver sends the event notification to the Workstation Subsystem, which puts the event on a queue.
- The server continuously checks the queue. When it finds an event, it sends the event to the appropriate client.
- The client handles the event.

The architecture in Figure 1–1 shows how all of the components work together, but each is more complex than the figure implies. Before adding support for a new graphics device, you need to understand the DIGITAL UNIX X Server, Workstation Subsystem, and graphics display drivers in more detail.

1.1 DIGITAL UNIX X Server

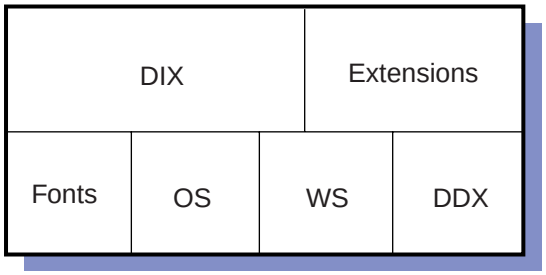
The DIGITAL UNIX X Server is dynamically configured at startup. Some components are always loaded into the server; others are loaded only when needed. This keeps the server size as small as possible.

This section describes the components of the DIGITAL UNIX X Server and how the server is dynamically configured at startup.

1.1.1 Server Components

The DIGITAL UNIX X Server is made up of several components, as shown in Figure 1–2. Each represents one or more shared libraries of routines. Most of these components also exist in the server that the Open Group distributes, and serve the same purpose as described in *The X Window System Server* (Digital Press).

Figure 1–2: DIGITAL UNIX X Server Architecture

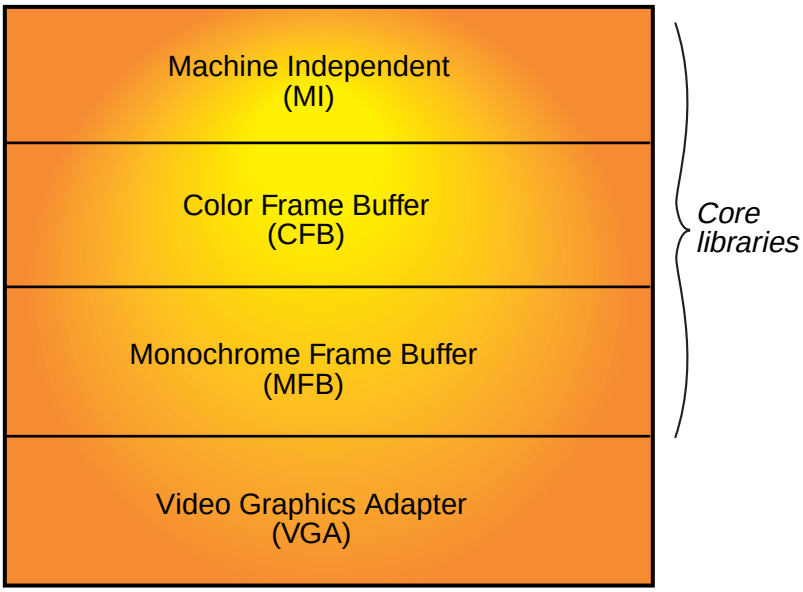


ZK-1230U-AI

Each component performs a particular function within the architecture, as follows:

- The device-independent X (DIX) component makes up the portable portion of the server. It is responsible for sending requests to the appropriate DDX or extension, or it may handle the request itself.
- Extensions add functionality to the server. Default extensions are always available in the server; deferred extensions are made available only when a client needs them. The DIGITAL UNIX X Server provides extensions for keyboard and pointer input devices.
- The fonts component retrieves fonts from font libraries. It provides the interface to the font server, which resides outside of the DIGITAL UNIX X Server.
- The operating system (OS) component performs operating system functions, such as setting up connections between the client and the server, transporting requests from the client to the server, or allocating and deallocating memory.
- The workstation subsystem component (WS) communicates with the Workstation Subsystem through the `/dev/ws0` device special file to perform common graphics operations.
- Device-dependent X (DDX) components support particular graphics devices. The library routines in a DDX handle pixmaps, regions, cursors, colormaps, screens, fonts, and graphics contexts for one type of graphics device. You add a DDX to add support for a new graphics board. DIGITAL supplies a number of core DDX libraries as shown in Figure 1–3. These libraries support all graphics devices. Other DDX libraries, such as the Video Graphics Adapter (VGA) library and the libraries that you create, call the routines in the core libraries. DIGITAL also supplies a DDX to handle input requests from the keyboard and pointer.

Figure 1–3: Core DDX Libraries



The core DDX libraries contain the following graphics functions:

– The machine-independent (MI) library contains graphics functions in their most generic form. This library implements many of the complicated algorithms associated with drawing lines and arcs. Unless your hardware provides specific support for these operations, you should call the routines in this library rather than reimplementing them yourself.

– The color frame buffer (CFB) library contains graphics functions that support color and gray-scale frame buffers (8, 16, and 32 bits/pixel).

– The monochrome frame buffer (MFB) library contains graphics functions that support black-and-white buffers (1 bit/pixel).

These library routines are not tuned to any particular graphics board, so they may not always perform as efficiently as possible. However, you can use them to quickly implement support for a new graphics device and replace them later with more efficient routines that take advantage of the hardware capabilities. The VGA library is an additional library that the DIGITAL UNIX X Server provides. It is not a core library, but it can help you provide support for a VGA graphics board.

1.1.2 Configuring the Loadable Server

Unlike the X Server that the Open Group distributes, which is a monolithic server, the DIGITAL UNIX X Server is dynamically configured at startup. Only those components that the client needs are loaded into the server. This means that the server does not take up resources that it does not need and will not use, making the DIGITAL UNIX X Server considerably smaller than its monolithic counterpart.

When the server starts up, it configures itself as follows:

1. It builds a table of all shared components that are available on the system. It uses two files to determine the components that go into the table.
   - The `loadable.c` file contains executable code that is part of the server. This code loads the DIGITAL core components, including the DIX, OS, fonts, and WS components, plus the core DDX libraries (MI, CFB, and MFB).
   - The `Xserver.conf` file is a data file that specifies all the additional components available on the system. When you add support for a new graphics board or extension, you must make an entry in the `Xserver.conf` file to define your component.
2. It loads the core components.
3. It opens the `/dev/ws0` device and queries the Workstation Subsystem for all graphics boards, loads the DDX libraries for each of those boards, and initializes them.
4. It loads the DDX libraries for the default input extensions (for the keyboard and pointer) and initializes them.
5. It loads the default extensions specified in `Xserver.conf` and initializes them. At this point, the server has the minimum number of components it needs to run. The minimum size of a server, however, can differ from one system to another, depending on its hardware configuration.
6. The server now enters its main loop and waits for requests from clients. It loads deferred extensions only if and when a client needs them. Therefore, before a client can make use of an extension, it needs to query the server. The server looks at its table of extensions. If it finds the extension, the server checks to see if the extension has been loaded. If not, it loads the extension. When the server resets, it deallocates all memory for deferred extensions. The extension is not loaded again until a client requests it.

The number of components that are loaded into the server can vary from one instance of the server to another. For example, consider a student workstation group at a university. One student may want to use a workstation for an MCAD application, using the OpenGL or PEX extension. The next student may need to do desktop publishing, using the Display Postscript extension. Yet another student may need to work with multimedia. A monolithic server could perform all of these tasks, but it would require much forethought by the programmer and a significant amount of system memory to build all of the necessary extensions into the server.

The DIGITAL UNIX X Server minimizes the server size by loading deferred extensions only when they are needed. If a student logs onto the system and runs a desktop publishing application, the server loads the Display PostScript extension. When the student logs out, the server relinquishes the memory for the extension. The next student logging onto the workstation may run a multimedia application. The server loads only the multimedia extension and relinquishes that memory when the student logs out. In this way, the server is always as small as it can be and still satisfy the needs of client applications.

The XDK supplies a collection of routines that perform loadable services. These routines form an application programming interface to the DIGITAL UNIX X Server loadable subsystem. Appendix D contains reference pages for all of the loadable services routines.

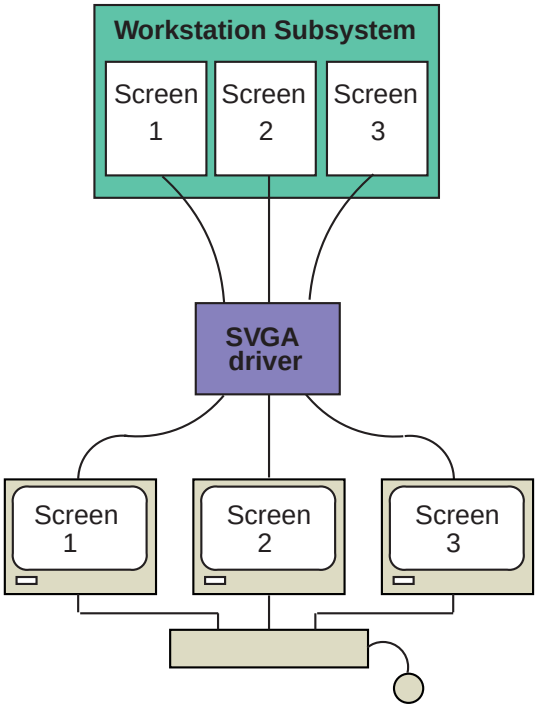
1.2 Workstation Subsystem

The Workstation Subsystem is a kernel subsystem with a device driver interface. It is not associated with any particular type of graphics hardware device, but rather provides a single interface to all graphics boards and input devices.

From user space, the server sends requests to the Workstation Subsystem by making an `ioctl` system call. The use of `ioctl` requests provides a hardware-independent interface. Adding a new hardware device does not change the interface. Using its knowledge of the devices connected to the system, the Workstation Subsystem `ioctl` interface dispatches the request to the appropriate device driver.

For example, Figure 1–4 shows a system in which three SVGA display devices are registered with the Workstation Subsystem. These three displays share the same pointer and keyboard (forming a multiheaded system). When the Workstation Subsystem receives a request for one of these displays, it calls the appropriate driver interface in the SVGA device driver.

Figure 1–4: Dispatching Interfaces to the Appropriate Device Driver



The Workstation Subsystem also handles input events from the system pointer and keyboard. It places input requests on an event queue. The server takes events from the queue and passes them to the appropriate client.

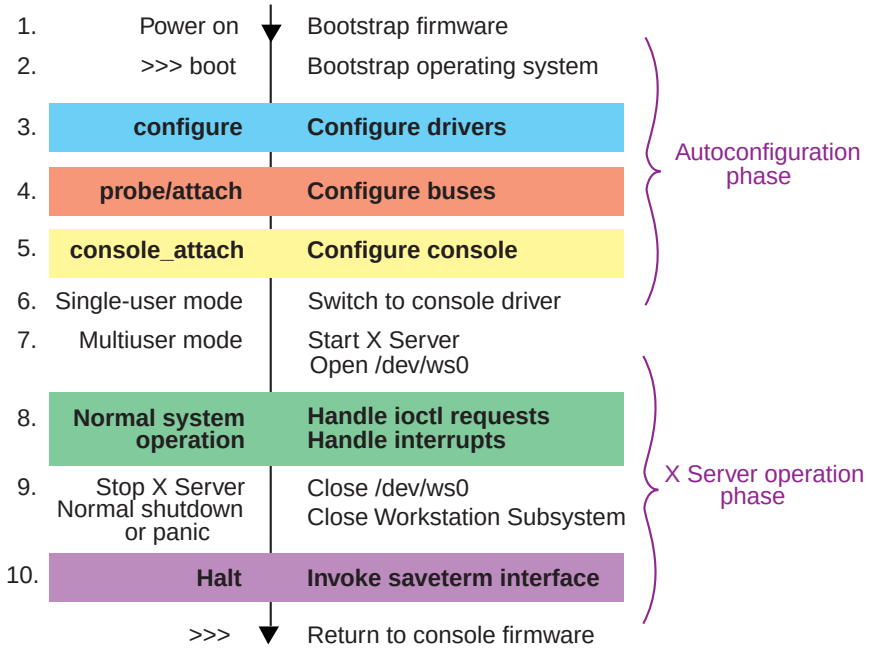
Because the Workstation Subsystem is a device driver in kernel space, it resides in the kernel even when the server is not running. Therefore, any graphics board can operate in console mode by tying into the Workstation Subsystem's console interface. An application can access a graphics display as a console terminal by issuing system calls to the Workstation Subsystem through the `/dev/console` device special file. The Workstation Subsystem dispatches the request to the appropriate device driver.

Although many requests go through the Workstation Subsystem's `ioctl` interface, other requests go directly to the device, bypassing the device driver. In this case, the DDX accesses the hardware registers and frame buffers to perform the graphics operation.

1.3 Graphics Display Device Drivers

A device driver communicates with a hardware device. For example, it sends commands to the device and returns state information from the device. Operation of a display device driver occurs in two phases. During the autoconfiguration phase, the driver configures the hardware into the operating system and provides console support. During the server operation phase, it provides support for X Window System operations and returns the graphics device to console mode when the server stops. Figure 1–5 shows the events that occur during each phase. The highlighted events show actions that the display device driver takes.

Figure 1–5: Time Line of a Running System



ZK-1232U-AI

The following list describes the events shown in the figure:

1. Power on

   When the user powers up the system, the console firmware performs basic initialization. The system displays the console prompt (>>>) and can accept some primitive commands. Console firmware code is specific to the system platform and contained on the system board, typically in a flash ROM. Using a VGA-based console card, the console firmware executes BIOS `init` and `set text mode 3` commands, which use an

internal Intel emulation facility that is not currently exported to the operating system.

2. `>>> boot`

   At the console prompt, the user issues a `boot` command, and the operating system begins the bootstrap procedure. The system uses console callbacks to render characters to the screen. Console callbacks are available only at startup to perform simple I/O operations on primary system devices.

3. Configure

   The platform code calls the driver's `configure` interface. The `configure` interface registers a callback routine to create the driver's `controller` structure and supply tunable parameters to the `/etc/sysconfigtab` database.

4. Probe/attach

   The bus code begins searching the bus for devices and calls the `probe` interface for each device driver. The driver's `probe` interface determines what devices are present on the system and whether they are able to accept requests from the user. It initializes the driver's `softc` structure — the global device driver structure. It may also initialize the driver's `controller` structure with the name of the driver's `console_attach` interface. If `probe` is successful and a graphics device is found, control returns to the bus code. The bus code calls the driver's `attach` interface for all drivers that are successfully probed. The `attach` interface can perform any noncritical configuration tasks. The `attach` interface is optional.

5. Console attach

   If the graphics console is also the system console, the kernel calls the driver's `console_attach` interface specified in the `controller` structure. For VGA-class cards or compound cards, DIGITAL provides a common VGA console driver. The `console_attach` routine for a VGA-based card need only call the `install_vga_console` routine, which registers the DIGITAL VGA console driver with the Workstation Subsystem.

Note

A serial-line console may be configured into your system, especially if you are doing initial development for a graphics card. If so, the kernel selects the serial-line driver and not the display device driver as the system console.

6. Single-user mode

   When it enters single-user mode, the system switches from console callbacks to the DIGITAL UNIX console driver. The system bootstrap procedure then either continues to multiuser mode or stays in single-user mode at the console terminal.

7. Multiuser mode

   In multiuser mode, operation of the display driver enters the second phase. This transition between the driver and the server is important because it must switch the device from console terminal operation to graphics operation, as follows:

   - When the server starts, the display driver disables output through the console driver. For example, kernel `printf` output does not go directly to the screen; it is available only from the log file in `/var/adm/messages` or through an application that redirects `/dev/console` output to a window, such as `dxconsole`.
   - The server accesses the Workstation Subsystem by opening the `/dev/ws0` device special file. If `open` is successful, the Workstation Subsystem calls the driver to initialize the hardware for graphics operations. For example, the driver initializes the device for cursor and colormap operations, and may disable the console chip or console mode. The driver's `init_screen` interface performs these initializations before the corresponding DDX executes its own initialization routines.
   - The Workstation Subsystem sets up the screen as follows:
     - Initializes the board with colormap entries.
     - Initializes the cursor.
     - Positions the cursor on the screen.
     - Turns on DPMS support, if necessary.

8. Normal system operation

   When the server starts up, the DDX creates the memory map for the server to use when directly accessing frame buffers and registers on the graphics board. While the server is running, the DDX issues `ioctl` system calls to the Workstation Subsystem for any common hardware-specific work that does not depend on the display memory. Screen, colormap, and cursor functions are usually performed by `ioctl` system calls.

9. Server stops

   The Workstation Subsystem performs several operations when the server shuts down or crashes, or when the operating system panics.

The transition from the server to the Workstation Subsystem is important because the graphics board must switch from graphics operation to console terminal operation, as follows:

- Resets all graphics screens by calling the following functions that the driver has registered with the Workstation Subsystem:
  - The `init_screen` function saves the device registers and resets the registers to their initial values.
  - The `clear_screen` function removes all graphics output from the screen.
  - The `cursor_on_off(off)` function removes the cursor from the screen.
  - The `video_off` function accesses the board to turn the screen off.
  - The `close` function returns the board to a state where it can resume console operation.
- Resets the console screen.

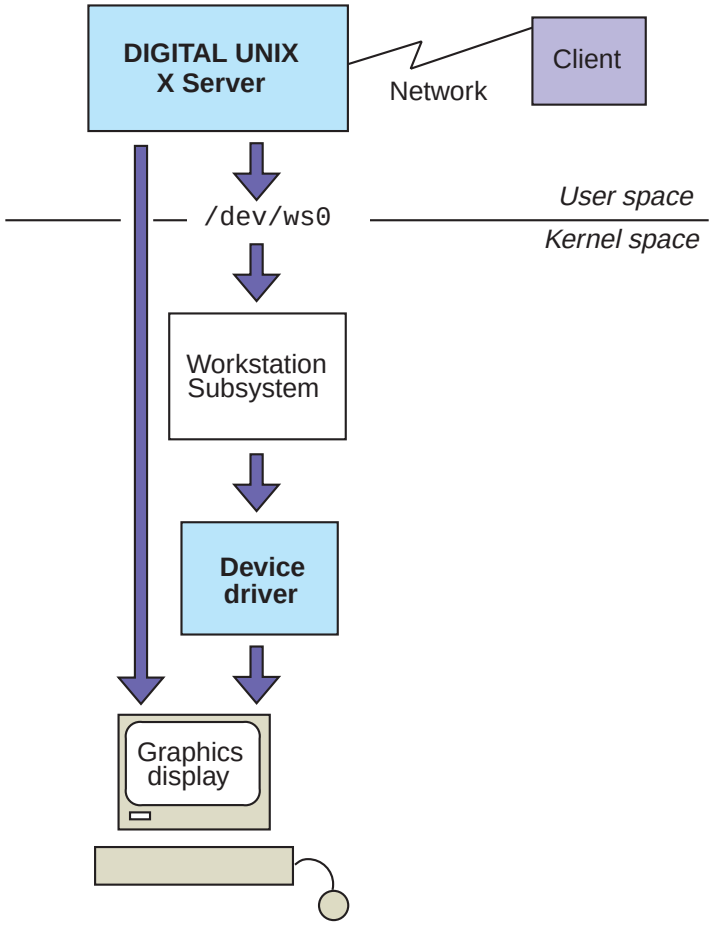
10. Halt

    When the user presses the hardware system halt button (for example, to stop a system hang and force a crash dump), usually no action is needed. However, if a separate chip is used for console text (as is the case for many of DIGITAL's Open3D controllers), some action may be required to reenable the console chip. The driver can register a `saveterm` function, which the kernel calls at this time to perform the necessary operations.

1.4 Adding Support for a New Graphics Board

Adding support for a new graphics board is not a trivial undertaking, although the DIGITAL UNIX X Server provides many libraries of routines to make the job easier. You must add code to the architecture at two points in the architecture, as shown in Figure 1–6. You add a device driver in kernel space, and you add a DDX in user space. When you are satisfied with your device driver and DDX, you can package them onto a diskette, CD–ROM, or tape, and ship this product kit to your customers.

Figure 1–6: Adding Graphics Device Support



ZK-1228U-AI

1.4.1 Adding a Device Driver

In kernel space, you write a device driver, which consists of the following interfaces:

- The `configure` interface is called during system configuration to register controller and device information associated with the device driver. DIGITAL has made the `configure` interface as standard as `read`, `write`, `open`, and `close`.
- The `probe` and `attach` interfaces are called at autoconfiguration time after the `configure` interface has completed. These interfaces determine what devices are connected to the system. In addition, they set up the addresses and data structures needed for performing

memory mapping. They register each device with the Workstation Subsystem and set up any interrupt handlers that the device driver needs. If any of these operations must succeed for the device to operate, the `probe` interface must perform them. The `attach` interface does not cause device initialization to succeed or fail.

- Interrupt-handler interfaces are called whenever an interrupt event occurs. Interrupt-handlers are optional for graphics devices, depending on the hardware. For example, the vertical retrace interrupt needs to be used on some hardware to change the cursor shape or the colormap with no visible side-effects. Most VGA boards, however, do not need to use this interrupt because they buffer such events and perform them at an appropriate time.
- Interfaces registered with the Workstation Subsystem to handle `ioctl` requests perform screen, cursor, and colormap functions. If you are providing support for special hardware that performs functions beyond the `ioctl` requests defined by DIGITAL UNIX, you can either implement device-specific `ioctl` interfaces or support these functions through the DDX and memory map.

You compile and link the device driver as a kernel subsystem and make it known to the kernel through the `/etc/sysconfigtab` database.

1.4.2 Adding a DDX

In user space, you add a DDX to the server to perform device-dependent operations, such as drawing a line or filling a polygon. At a minimum, you must write an initialization routine. The initalization routine creates a data structure to represent the display and name the DDX routines that perform display functions.

You can use routines in the core DDX libraries to implement these functions. If the device you want to add is a VGA device, you can use the VGA library routines to implement most or all of these functions. However, to take advantage of your hardware capabilities, you will want to write your own routines to perform some graphics functions.

All hardware devices have strengths and weaknesses. The routines that DIGITAL UNIX supplies will work on most or all graphics boards. However, as a DDX writer, you must look at the hardware and its capabilities, then write routines that take advantage of those capabilities, replacing as many routines as necessary to get the best performance. The DDX for the example VGA board implements device-specific routines to create and destroy windows, but it uses machine-independent routines in the MI library to create and destroy regions.

You compile and link the DDX routines into a shared library, then add the library to the `Xserver.conf` file so that the server can load it at startup.

1.4.3 Packaging the Driver and DDX

When you are satisfied with the performance of your display device driver and DDX, you can create a kit on tape, CD–ROM, or diskette, as described in *Guide to Preparing Product Kits*.

2

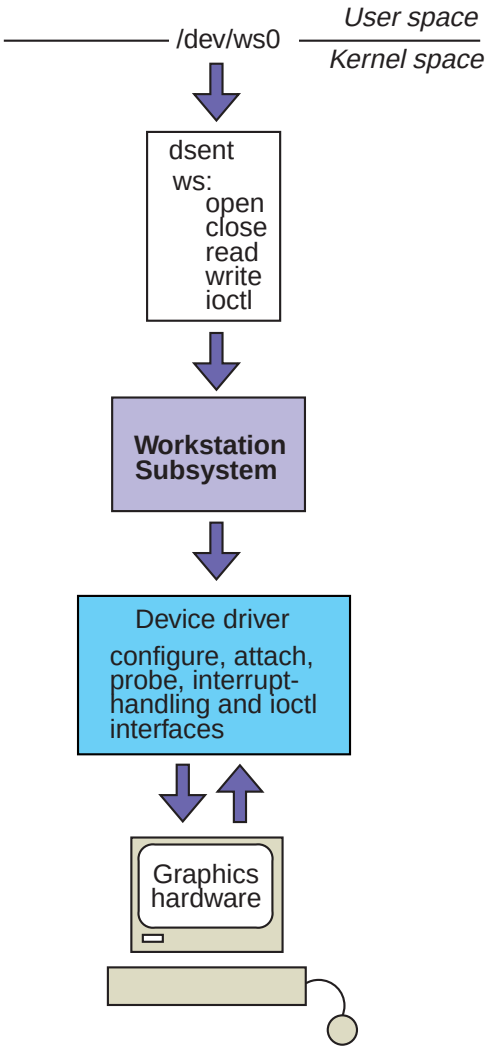
Writing a Display Device Driver

Graphics devices are supported by two components in kernel space:

- The Workstation Subsystem provides a single common interface between the client and all graphics devices. It is a device driver that follows the model described in *Writing Device Drivers: Tutorial*. The Workstation Subsystem registers its `open`, `close`, `read`, `write`, and `ioctl` interfaces with the kernel. All requests to graphics devices go through the Workstation Subsystem.
- The device driver provides routines to perform graphics functions for a specific graphics board. The Workstation Subsystem calls these routines when it receives an `ioctl` request. The graphics device driver does not register these routines with the kernel; it registers them with the Workstation Subsystem. In addition to these registered routines, the device driver provides interfaces to configure the driver into the system, probe the buses for the devices it manages, and attach the devices to the hardware topology. These interfaces get called during system startup. During driver configuration, the graphics device driver defines the memory map, which makes the device registers and frame buffers available directly to the DDX. If necessary, it can define interrupt-handling interfaces, though this is uncommon for SVGA graphics drivers. A display device driver does not need to contain support for the mouse and keyboard. The Workstation Subsystem and DIGITAL-supplied input graphics drivers provide support for primary input devices.

Figure 2–1 shows the model used for implementing graphics device drivers.

Figure 2–1: Graphics Device Driver Model



Writing a graphics device driver involves the following tasks:

- Declaring data structures to describe the graphics board and to hold I/O handles for memory mapping
- Implementing a `configure` interface and a configuration callback routine to configure the driver into the system
- Implementing a `probe` and, optionally, an `attach` interface to configure the graphics board into the system

- Implementing a `console_attach` interface, if the graphics device can also act as the console terminal when the X Window System is not running
- Implementing routines to handle interrupts, if necessary
- Implementing routines to perform graphics functions when the Workstation Subsystem receives an `ioctl` request

This chapter describes how to perform each of these tasks, using a sample graphics device driver called myvga. The sample driver is included with the X Developers Kit. If this is the first time you have implemented a graphics device driver, you can use the myvga driver as the basis for your VGA-compliant driver.

Note

In this release, the Workstation Subsystem uses a funnel to force execution of the device driver onto a single CPU. This means that the device driver does not have to lock any resources to function in a multiprocessor environment. However, to be compatible with future releases of the DIGITAL UNIX X Server, you should make your driver SMP safe, as described in *Writing Device Drivers: Advanced Topics*.

2.1 Data Structure Declarations

A graphics device driver must declare the following kinds of data structures:

- Data structures to describe the characteristics of the graphics board. The Workstation Subsystem defines the format of these data structures, which the device driver includes in its own device-specific data structure.
- I/O handles to allow the DDX to access the graphics hardware through memory mapping. The kernel defines the format of I/O handles, which the device driver includes in its own device-specific data structure.

2.1.1 Defining the Characteristics of the Graphics Board

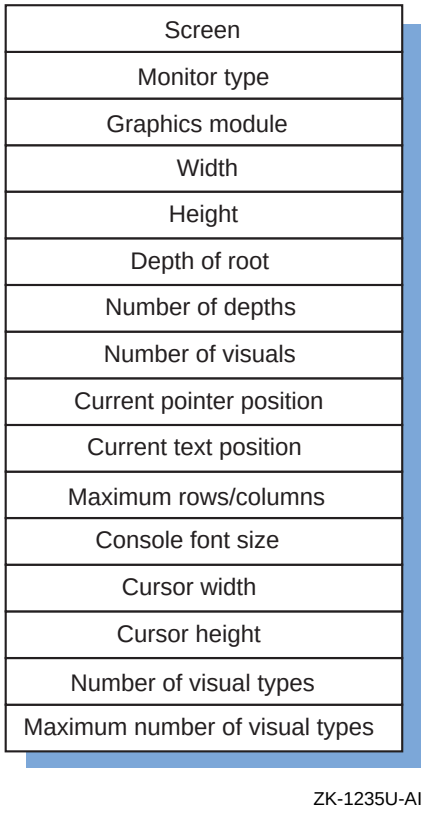
The Workstation Subsystem defines the following data structures for screen, depth, and visual descriptors, and structures for screen, colormap, and cursor functions:

- Screen descriptor

  The `ws_screen_descriptor` data structure defines characteristics of the screen display, such as the monitor type, height and width of the

display, and the current coordinates of the cursor. This data structure is defined in `/usr/sys/include/sys/workstation.h` and included in the driver's device-specific data structure. Figure 2–2 shows the format of this data structure.

Figure 2–2: Format of a Screen Descriptor

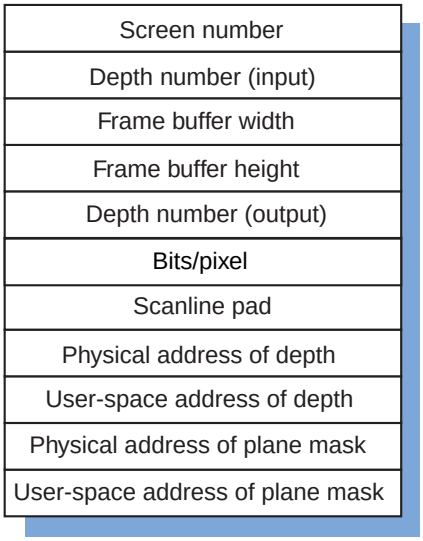


ZK-1235U-AI

- Depth descriptor

  The `ws_depth_descriptor` data structure defines the depth of the root window, including charactistics such as the frame buffer width and height in pixels. This data structure is defined in `/sys/workstation.h` and included in the driver's device-specific data structure. Figure 2–3 shows the format of this data structure.

Figure 2–3: Format of a Depth Descriptor

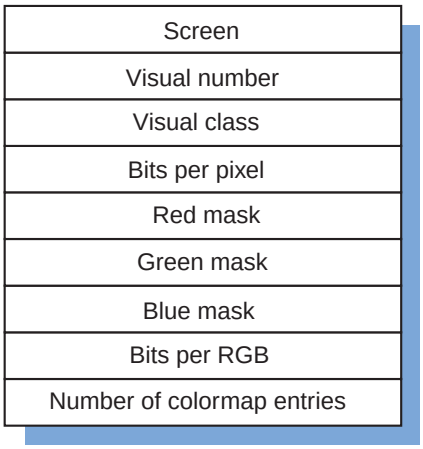


ZK-1237U-AI

- Visual descriptor

  The `ws_visual_descriptor` data structure defines the visual characteristics of the display, such as the visual class and the number of colormap entries. This data structure is defined in `/sys/workstation.h` and included in the driver's device-specific data structure. Figure 2–4 shows the format of this data structure.

Figure 2–4: Format of a Visual Descriptor

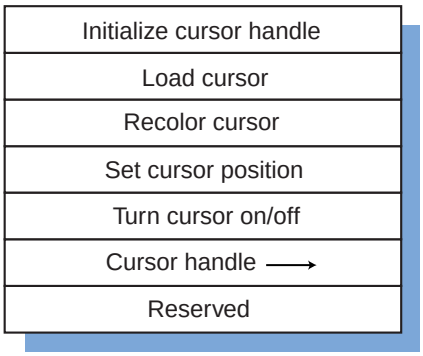


ZK-1238U-AI

- Cursor functions

  The device driver must define routines to initialize and load the cursor, to set the cursor color and position, and to turn the cursor on and off. You initialize the `ws_cursor_functions` data structure with the names of these routines so that the Workstation Subsystem can dispatch `ioctl` requests to them when necessary. The Workstation Subsystem defines the `ws_cursor_functions` data structure in `/sys/wsdevice.h`, and the driver includes it in the device-specific data structure. Figure 2–5 shows the format of this data structure.

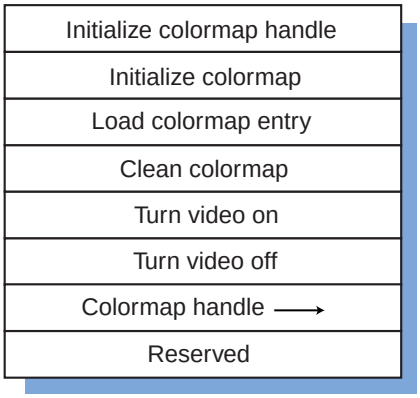
Figure 2–5: Cursor Function Definition



- Colormap functions

  The device driver must define routines to manage the colormap, such as initializing and loading the colormap, and turning the video on and off. You initialize the `ws_color_map_functions` data structure with the names of these routines so that the Workstation Subsystem can dispatch `ioctl` requests to them when necessary. The Workstation Subsystem defines the `ws_color_map_functions` data structure in `/sys/wsdevice.h`, and the driver includes it in the device-specific data structure. Figure 2–6 shows the format of this data structure.

Figure 2–6: Colormap Function Definition

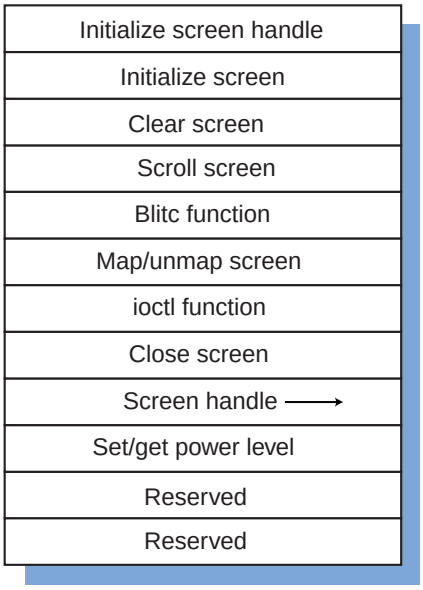


ZK-1240U-AI

- Screen functions

  The device driver must define routines to manage the screen, such as scrolling, clearing, and mapping the screen. You initialize the `ws_screen_functions` data structure with the names of these routines so that the Workstation Subsystem can dispatch `ioctl` requests to them when necessary. The Workstation Subsystem defines the `ws_screen_functions` data structure in `/usr/sys/include/sys/workstation.h`, and the driver includes it in the device-specific data structure. Figure 2–7 show the format of this data structure.

Figure 2–7: Screen Function Definition



ZK-1241U-AI

The display device driver includes these structures in a device-specific data structure that describes the particular graphics board. For example, the myvga device driver defines a `myvga_type` data structure that includes the Workstation Subsystem structures plus additional information about the graphics board, as follows:

```
struct myvga_type {
   ws_screen_descriptor screen;
   ws_depth_descriptor depth[MYVGA_NDEPTHS];
   ws_visual_descriptor visual[MYVGA_NVISUALS];
   ws_screen_functions sf;
   ws_color_map_functions cmf;
   ws_cursor_functions cf;
   unsigned int state;
   short ctlr_type;
   short bus_type;
   pid_t mapped_pid;
   unsigned short attribute;
   unsigned short unit;
   unsigned short board_id;
   short min_dirty;
   short max_dirty;
   short x_hot;
   short y_hot;
   ws_color_cell cursor_fg;
   ws_color_cell cursor_bg;
```

```
    unsigned long mem_phys;
    io_handle_t mem_handle;
    unsigned long cursor_offset;
    unsigned int mem_size;
    u_int bits[256];
    struct myvga_color_cell cells[256];
    ihandler_id_t *intr_handle;
    io_handle_t iobase;
    io_handle_t membase;
    void *orig_state;
    void *new_state;
};
```

The driver's `probe` interface creates and initializes these structures as described in Section 2.3.1.

2.1.2 Defining I/O Handles and Macros

An I/O handle is a data structure of type `io_handle_t`, which the driver uses to access the frame buffer (video memory) and control status registers on the graphics board. The driver declares one I/O handle to contain the base address of the hardware I/O registers, and one to contain the base address of the hardware memory. The driver's `probe` interface initializes these handles.

The myvga device driver declares I/O handles as part of the `myvga_type` structure, as follows:

```
struct myvga_type {
⋮
    io_handle_t iobase;
    io_handle_t membase;
⋮
};
```

Using the read and write macros that DIGITAL provides, a driver usually defines macros to read and write data to and from I/O handles at a specified offset from their base addresses. For example, the myvga driver defines an `INB` macro to read a byte, an `OUTB` macro to write a byte, and an `OUTW` macro to write a word at the specified offset:

```
#define INB(a) READ_BUS_D8((a)+scp->iobase)
#define OUTB(a,v) {WRITE_BUS_D8((a)+scp->iobase, (v)); mb();}
#define OUTW(a,v) {WRITE_BUS_D16((a)+scp->iobase, (v)); mb();}
```

Defining similar macros for your graphics board can make it easier to read and write to and from the hardware. In Section 2.3.1, the `probe` interface uses these macros to test whether the hardware is operating correctly.

To access specific offsets within the I/O registers and memory, the driver can define constants for each hardware offset that it requires. The myvga driver defines the following offsets:

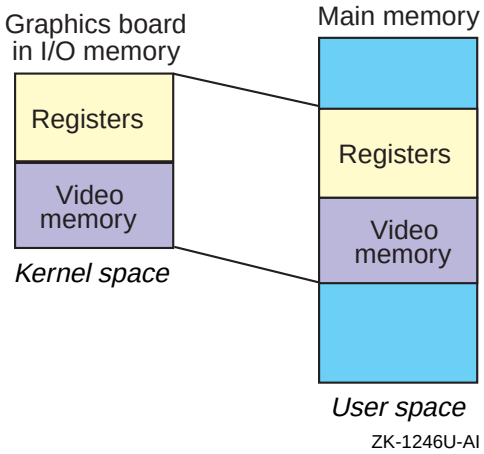
```
#define MYVGA_CRTC_ADDRESS       0x3D4
#define MYVGA_CRTC_DATA          0x3D5

#define MYVGA_SEQ_ADDRESS        0x3C4
#define MYVGA_SEQ_DATA           0x3C5

#define MYVGA_PEL_MASK           0x3C6
#define MYVGA_PEL_ADDR_WMODE     0x3C8
#define MYVGA_PEL_DATA           0x3C9
```

You need to consult your hardware specification to determine the correct offsets for the registers and memory locations on your graphics board.

You may also want to let the DDX access hardware registers and memory directly. This is often more efficient than sending `ioctl` requests through the Workstation Subsystem, especially for operations that the hardware optimizes.

To make registers and memory available to the DDX code, you create a memory map that places the I/O handle addresses in main memory. When the DDX accesses an address in the memory map, it goes through main memory to the hardware. Figure 2–8 shows a memory map that maps the addresses of hardware registers and video memory into system memory.

Figure 2–8: Memory Map of Registers and Frame Buffers



The driver's `map_unmap_screen` routine initializes the memory map. The DDX calls this routine by making a `MAP_SCREEN_AT_DEPTH` `ioctl` request during its initialization routine.

2.2 Configuring the Graphics Board

During system startup, the kernel calls the driver's `configure` interface to configure the driver into the system. However, the system may not be ready to configure the driver at that time. For example, virtual memory or the file system may not be available until later in the startup procedure. If the driver needs these resources, it must register a configuration callback routine to be called later in the startup procedure.

Therefore, to configure a display driver into the system, you supply both a `configure` interface and a configuration callback routine.

2.2.1 The configure Interface

The startup procedure is divided into stages, and a device driver can register to have its configuration callback routine get called at any of those stages. A display device driver should be configured after virtual memory becomes available, at the `CFG_PT_PRECONFIG` configuration callback point. For example:

```
int
myvga_configure(cfg_op_t op,
                caddr_t indata,
                ulong indatalen,
                caddr_t outdata,
                ulong outdatalen)
{
   int ret = -1;
⋮
   switch (op) {
    case CFG_OP_CONFIGURE:
⋮
      ret = register_callback(myvga_do_configuration,
                              CFG_PT_PRECONFIG,
                              CFG_ORD_NOMINAL,
                              0L);
      break;
```

Prior to calling `register_callback`, the driver may need to perform other operations, such as checking the device attributes table. You must determine what other configuration operations the driver needs to perform.

At run time, the `sysconfig` utility can call the driver's `configure` interface to query attributes from the device attribute table, or to reconfigure or unconfigure the driver. If the driver performs no special process for these configuration options, simply provide an empty `case` statement for each option. For example:

```
    case CFG_OP_QUERY:
      break;

    case CFG_OP_RECONFIGURE:
      break;

    case CFG_OP_UNCONFIGURE:
      break;

    default:
      return EINVAL;
   }

  return 0;
}
```

Unless the user specifies an invalid configuration option, the `configure` interface should always succeed.

For more information on writing a `configure` interface, see *Writing Device Drivers: Tutorial*.

2.2.2 Configuration Callback Routine

The configuration callback routine is responsible for configuring the driver into the system. The device driver is represented in the hardware topology by a `controller` structure. To create a `controller` structure for the device driver, you create a `controller_config` structure and pass this structure to the `create_controller_structure` kernel routine.

The configuration callback routine should initialize one `controller_config` structure for each controller installed on the system. The driver supplies the subsystem name, the types of bus that it can support, and the name of the device driver. For example:

```
void
myvga_do_configuration()
{

   extern struct driver myvgadriver;
   struct controller_config tmp_ctlr_register;
   int status = -1, i;

   for(i = 0; i < myvga_num_installed; i++) {
      tmp_ctlr_register.revision = CTLR_CONFIG_REVISION;
      strcpy(tmp_ctlr_register.subsystem_name, MYVGA_NAME);
      strcpy(tmp_ctlr_register.bus_name, DRIVER_WILDNAME);
      tmp_ctlr_register.devdriver = &myvgadriver;
```

The routine then passes the initialized structure to the `create_controller_struct` routine, which creates the `controller` structure with the information that the configuration callback routine supplies, plus information known internally to the system. For example:

```
      status = create_controller_struct(&tmp_ctlr_register);
```

If `create_controller_struct` cannot create the structure, the callback routine sets a flag to indicate the failure, then it exits.

```
    if(status != ESUCCESS) {
      if(status != ESUCCESS) {
         printf("myvga_do_configuration: Unable to link ctlr\n");
         (void)cfgmgr_set_status(MYVGA_NAME);
         myvga_failed_config = TRUE;
         break;
      }
   }
}
```

The configuration callback routine does not need to create `device` structures, as described in *Writing Device Drivers: Tutorial*, because the Workstation Subsystem creates them.

2.3 Performing Device Autoconfiguration

The driver's `probe` and `attach` interfaces perform device autoconfiguration. That is, they determine what devices are present on the system and whether the devices are able to accept requests from the user.

When deciding which tasks the `probe` interface should perform and which tasks the `attach` interface should perform, keep in mind that autoconfiguration fails if the `probe` interface fails, but autoconfiguration does not fail if the `attach` interface fails. Therefore, you should put critical operations in the `probe` interface and noncritical operations in the `attach` interface. For example, you cannot access a device's I/O registers if the I/O handles were not created during system startup. Because this operation determines whether the device is successfully autoconfigured into the system, you should always place it in the `probe` interface.

The version of the myvga device driver described in this chapter performs noncritical tasks in the `attach` interface. The version of the driver shipped on the XDK places all of the autoconfiguration code in the `probe` interface and provides no `attach` interface. Both are acceptable coding practices.

2.3.1 The probe Interface

The kernel calls the driver's `probe` interface during system startup to determine what devices are connected to the system. The `probe` interface for a graphics board performs the following tasks:

- Allocates an array of device-specific structures. This array, called a `softc` structure, becomes part of the hardware topology — a representation of the system hardware, which the system keeps in memory while it is up and running.
- Initializes I/O handles to the base addresses of the I/O registers and frame buffer on the graphics board.
- Tests the device to make sure it is operating correctly.
- Allocates device-specific data structures, if necessary.
- Initializes the `softc` structure.
- Registers all graphics display devices with the Workstation Subsystem.
- Enables interrupts, if the device is interrupt-driven. (Most DIGITAL VGA drivers do not use interrupts.)

The sections that follow describe these tasks.

2.3.1.1 Allocating the softc Structure

The `softc` structure is an array of structures that define the graphics boards configured into the system. For example, the `myvga_softc` structure is an array of `myvga_type` structures, one for each graphics controller configured on the system. The `myvgaprobe` interface allocates memory for the `softc` structure by calling the `MALLOC` macro. For example:

```
struct  myvga_type   *myvga_softc[MAX_MYVGA_CONTROLLERS];

int
myvgaprobe(vm_offset_t addr,
           register struct controller *ctlr)
{
   register struct myvga_type *scp;
⋮
   MALLOC(myvga_softc[ctlr->ctlr_num],
          struct myvga_type *,
          sizeof(struct myvga_type),
          NULL, NULL);

   if (myvga_softc[ctlr->ctlr_num] ==
          (struct myvga_type *)NULL) {
      printf("myvgaprobe: Unable to MALLOC softc[%d]\n",
```

```
            ctlr->ctlr_num);
      return 0;
   }

   scp = myvga_softc[ctlr->ctlr_num];
```

The `scp` variable contains the current `myvga_type` structure so that `probe` can access the information in that structure more easily.

2.3.1.2 Initializing I/O Handles

The `probe` interface is responsible for creating the I/O handles that access the hardware registers and frame buffers. To obtain the addresses of the bus I/O address space and memory space, call the `busphys_to_iohandle` kernel interface and specify the bus physical address, the address space type (register or memory), and the `controller` structure associated with the graphics board. (The kernel passes the `controller` structure as an argument to the `probe` interface.)

Creating I/O handles for registers and frame buffers is different for different bus types. For example, some controllers allow registers to be mapped to bus memory space. (The VGA standard does not use this capability.) If your graphics display device driver can operate on multiple buses, you can place the code that creates I/O handles in a `switch` statement. For example:

```
   switch(ctlr->bus_hd->bus_type){
    case BUS_PCI:
⋮
    case BUS_ISA:
⋮
    case BUS_EISA:
⋮
    default:
      printf("myvgaprobe: Unsupported bus type: %d\n",
             ctlr->bus_hd->bus_type);
      FREE(scp, M_DEVBUF);
      return 0;
   }
```

If you are going to provide VGA console support, you must also create the following I/O handles:

```
  VGA_io_base_handle = busphys_to_iohandle(0, BUS_IO, ctlr);
  VGA_mem_base_handle = busphys_to_iohandle(0, BUS_MEMORY, ctlr);
```

2.3.1.3 Testing the Graphics Board

The probe interface is responsible for making sure the graphics board is operating correctly. For any graphics board, you can perform a simple read/write test to make sure you can access the I/O registers and memory. The myvga device driver calls its INB and OUTB macros defined in Section 2.1.2 to test whether it can read and write to the board, as follows:

```
  OUTB(MYVGA_CRTC_ADDRESS, 0x0a); start = INB(MYVGA_CRTC_DATA);
  OUTB(MYVGA_CRTC_ADDRESS, 0x0b); end   = INB(MYVGA_CRTC_DATA);

  OUTB(MYVGA_CRTC_ADDRESS, 0x0a); OUTB(MYVGA_CRTC_DATA, VAL1);
  OUTB(MYVGA_CRTC_ADDRESS, 0x0b); OUTB(MYVGA_CRTC_DATA, VAL2);

  OUTB(MYVGA_CRTC_ADDRESS, 0x0a); data1 = INB(MYVGA_CRTC_DATA);
  OUTB(MYVGA_CRTC_ADDRESS, 0x0b); data2 = INB(MYVGA_CRTC_DATA);

  OUTB(MYVGA_CRTC_ADDRESS, 0x0a); OUTB(MYVGA_CRTC_DATA, start);
  OUTB(MYVGA_CRTC_ADDRESS, 0x0b); OUTB(MYVGA_CRTC_DATA, end);
```

If the driver reads invalid data from the board, it clears a flag to indicate that probe failed.

```
   if (data1 != VAL1 || data2 != VAL2) {
       myvga_softc[ctlr->ctlr_num]->ctlr_type = -1;
       printf("myvga0: generic vga probe failed\n");
       FREE(scp, M_DEVBUF);
       return(0);
}
```

Your driver may need to perform special tests in its probe routine to ensure that the device operates correctly, depending on the hardware requirements. For a PCI bus, for example, you might read the vendor ID and device ID from the board or check that the base class code and subclass codes are correct. You would also place these tests within a case statement, as described in Section 2.3.1.2.

2.3.1.4 Allocating Device-Specific Data Structures

A driver may need to define additional data structures for its own use. For example, the driver may need to save the graphics state — the contents of registers and other information — before changing from graphics mode to console mode. Using the information in the data structure, the driver can restore the board to its original state when it returns to graphics mode. For example:

```
   MALLOC(scp->orig_state, vgaHWRec *, sizeof (vgaHWRec),
          M_HWINTR, (M_NOWAIT | M_ZERO));
   if (!scp->orig_state) {
```

```
   printf("myvgaprobe: failed malloc of register structs\n");
   FREE(scp,M_DEVBUF);
   return(0);
}

MALLOC(scp->new_state, vgaHWRec *, sizeof (vgaHWRec),
       M_HWINTR, (M_NOWAIT | M_ZERO));
if (!scp->new_state) {
   printf("myvgaprobe: failed malloc of register structs\n");
   FREE(scp,M_DEVBUF);
   return(0);
}
```

Your driver may require different data structures. If the success of autoconfiguration depends on these data structures, the driver should allocate them when the probe interface executes.

2.3.1.5 Initializing the softc Structure

Every driver must register with the Workstation Subsystem so that the Workstation Subsystem can dispatch workstation requests to the appropriate device driver entry point. To do this registration, you need to supply information to the Workstation Subsystem about the graphics board and the routines that perform graphics functions. Section 2.1.1 shows the data structures that contain this information.

The myvga device driver's probe routine calls a subroutine to initialize the data structure, as follows:

- ws_screen_descriptor

  The device driver intializes the ws_screen_descriptor data structure with the characteristics of the screen.

```
int myvga_fill_softc(register struct
                     myvga_type *scp,
                     struct controller *ctlr)
{
⋮
   scp->screen.screen = ctlr->ctlr_num;
   scp->screen.monitor_type = tmp_monitor_type;
   strcpy(scp->screen.moduleID,"VGA      ");
   scp->screen.width = 640;
   scp->screen.height = 480;
   scp->screen.root_depth = 0;
   scp->screen.allowed_depths = 1;
   scp->screen.nvisuals = 1;
   scp->screen.x = 0;
   scp->screen.y = 0;
   scp->screen.row = 0;
```

```
scp->screen.col =0;
scp->screen.max_row = 24;
scp->screen.max_col = 80;
scp->screen.f_width = 8;
scp->screen.f_height = 16;
scp->screen.cursor_width = 32;
scp->screen.cursor_height = 32;
scp->screen.min_installed_maps = 1;
scp->screen.max_installed_maps = 1;
```

- `ws_depth_descriptor`

  The device driver initializes this data structure with the characteristics of the screen depth.

```
scp->depth[0].screen = 0;
scp->depth[0].which_depth = 0;
scp->depth[0].fb_width = 640;
scp->depth[0].fb_height = 480;
scp->depth[0].depth = 8;
scp->depth[0].bits_per_pixel = 8;
scp->depth[0].scanline_pad = 32;
scp->depth[0].physaddr = 0;
scp->depth[0].pixmap = 0;
scp->depth[0].plane_mask_phys = 0;
scp->depth[0].plane_mask = 0;
```

- `ws_visual_descriptor`

  The device driver initializes this data structure with the characteristics of the screen visuals.

```
scp->visual[0].screen = 0;
scp->visual[0].which_visual = 0;
scp->visual[0].screen_class = PseudoColor;
scp->visual[0].depth = 8;
scp->visual[0].red_mask = 0;
scp->visual[0].green_mask = 0;
scp->visual[0].blue_mask = 0;
scp->visual[0].bits_per_rgb = 8;
scp->visual[0].color_map_entries = 256;
```

- `ws_cursor_functions`

  The display device driver initializes this data structure with the names of its cursor routines.

```
scp->cf.init_cursor_handle = myvga_init_cursor_handle;
scp->cf.load_cursor = myvga_load_cursor;
scp->cf.recolor_cursor = myvga_recolor_cursor;
scp->cf.set_cursor_position = myvga_set_cursor_position;
scp->cf.cursor_on_off = myvga_cursor_on_off;
scp->cf.cursor_private = NULL;
```

- `ws_color_map_functions`

  The display device driver initializes this data structure with the names of its colormap routines.

```
scp->cmf.init_colormap_handle =
        myvga_init_color_map_handle;
scp->cmf.init_color_map = myvga_init_color_map;
scp->cmf.load_color_map_entry =
        myvga_load_color_map_entry_6bit;
scp->cmf.clean_color_map = myvga_clean_color_map;
scp->cmf.video_on = myvga_video_on;
scp->cmf.video_off = myvga_video_off;
scp->cmf.cmap_private = NULL;
```

- `ws_screen_functions`

  The display device driver initializes this data structure with the names of its screen routines.

```
scp->sf.init_screen_handle = myvga_init_screen_handle;
scp->sf.init_screen = myvga_init_screen;
scp->sf.clear_screen = myvga_clear_screen;
scp->sf.scroll_screen = myvga_scroll_screen;
scp->sf.blitc = myvga_blitc;
scp->sf.map_unmap_screen = myvga_map_unmap_screen;
scp->sf.ioctl = myvga_ioctl;
scp->sf.close = myvga_close;
scp->sf.set_get_power_level = NULL;
scp->sf.screen_private = NULL;
scp->sf.screen_private2 = NULL;
```

The driver's `probe` interface must also initialize the structure with pointers to screen, cursor, and colormap handles. These are internal pointers that the Workstation Subsystem passes to the graphics functions. They speed the performance of graphics operations and let you write and use chip-specific drivers for those functions. For example, you could write a module for a common cursor chip and use it with several different graphics board drivers.

The driver intializes the structure with these handles, as follows:

```
scp->sf.screen_handle =
   (*(scp->sf.init_screen_handle))
   ((caddr_t)myvga_softc, ctlr->physaddr, ctlr->ctlr_num, 0);
scp->cf.cursor_handle =
   (*(scp->cf.init_cursor_handle))
   ((caddr_t)myvga_softc, ctlr->physaddr, ctlr->ctlr_num, 0);
scp->cmf.colormap_handle =
   (*(scp->cmf.init_colormap_handle))
   ((caddr_t)myvga_softc, ctlr->physaddr, ctlr->ctlr_num, 0);
```

2.3.1.6 Registering with the Workstation Subsystem

The driver registers with the Workstation Subsystem by calling the `ws_register_screen` routine and passing pointers to the initialized Workstation Subsystem structures. For example:

```
status = ws_register_screen(&scp->screen,
                            scp->visual,
                            scp->depth,
                            &scp->sf,
                            &scp->cmf,
                            &scp->cf,
                            ctlr);
if (status == -1) {
   printf("myvga driver: could not register screen\n");
   FREE(scp, M_DEVBUF);
   return 0;
}
```

The driver cannot continue the autoconfiguration process for this device if it cannot register with the Workstation Subsystem. The driver checks the return value from `ws_register_screen` and exits with an error status if the registration fails.

2.3.1.7 Setting the Global Display Type for sizer

The `sizer` utility displays information about the system, including information about the devices connected to it. To make information about your graphics board available to `sizer`, you need to initialize global variables with the workstation display units and display type that the `–wu` and `–wt` options return. For example:

```
if (ws_display_units < 8) {
    ws_display_type = (ws_display_type << 8) | WS_DTYPE;
    ws_display_units = (ws_display_units << 1) | 1;
}
```

See `sizer`(8) for more information about the `sizer` utility.

2.3.1.8 Enabling Interrupt Handlers

The `probe` interface registers interrupts by first initializing `handler_intr_info` and `ihandler_t` structures. The `handler_intr_info` structure contains information about the interrupt handlers for the device controllers connected to a bus, such as the controller number, interrupt handler name, and driver type (controller or bus adapter). The `ihandler_t` structure contains information associated with device driver interrupt handling.

```
struct handler_intr_info handler_info;
ihandler_t handler;
   .
   .
   .
if (myvga_intr_enable) {
   handler.ih_bus = ctlr->bus_hd;
   handler_info.configuration_st = (caddr_t)ctlr;
   handler_info.config_type = CONTROLLER_CONFIG_TYPE;
   handler_info.intr = myvgaintr;
   handler_info.param = (caddr_t)ctlr->ctlr_num;
   handler.ih_bus_info = (char *)&handler_info;
```

The driver then calls the handler_add interface, passing the address of the ihandler_t structure, as follows:

```
scp->intr_handle = handler_add(&handler);
```

This interface registers a device driver's interrupt handler with the bus that dispatches the interrupt.

If the handler_add interface is unsuccessful, the driver unregisters the handler. If the interface is successful, the driver calls the handler_enable interface. If handler_enable fails, the driver deletes the interrupt handler and exits. For example:

```
if (scp->intr_handle == NULL) {
   FREE(scp,M_DEVBUF);
   return 0;
}
if (handler_enable(scp->intr_handle) != 0) {
   handler_del(scp->intr_handle);
   FREE(scp,M_DEVBUF);
   return 0;
}
}
```

Interrupts are optional for many graphics boards. If that is the case for your board, you do not need to exit if handler_add or handler_enable fails. Instead, you may want to set a flag to indicate that interrupts are not enabled, then continue with the autoconfiguration procedure.

For further information on interrupt-handler data structures and interfaces, see *Writing Device Drivers: Tutorial*.

2.3.2 The attach Interface

The device driver can perform any noncritical autoconfiguration tasks when the attach interface executes. For example, if the VGA graphics display can also be used as the console terminal when the X Window System is not running, the driver needs to set the console_attach member of the

controller structure to the name of the driver's console_attach interface. You can do this in the attach interface, as follows:

```
ctlr->console_attach = (caddr_t)myvga_console_attach;
```

Section 2.4 describes how to write the console_attach interface for VGA drivers.

Before autoconfiguration of the device ends, the driver should display information about the graphics board. For example:

```
printf(" %dx%d %s\n", scp->screen.width,
       scp->screen.height,
       scp->screen.moduleID);
printf("%s%d: MYVGA example driver - version %s\n",
       ctlr->ctlr_name, ctlr->ctlr_num, myvga_version);
```

These messages can be important for identifying the board if problems occur. The myvga device driver displays the screen height, width, and module ID, plus the controller name and number. Without this information, there may be no way to track down a graphics board that has not been properly configured.

2.4 Supporting the VGA Console Device

When a VGA graphics monitor is the system terminal and the X Window System is not running, console support allows a user to log in as root on the graphics monitor. User programs needing console support open /dev/console. For the graphics console, this device sends requests to the Workstation Subsystem, which dispatches the request to the appropriate driver interface.

DIGITAL supplies a VGA console driver that any VGA display driver can use for console support. To make this driver available to a graphics board, the display driver supplies a console_attach interface that calls the install_vga_console routine. The install_vga_console routine registers the VGA console driver interfaces with the Workstation Subsystem for this graphics board.

The following example shows the console_attach interface for the myvga device driver. The vga_use_orig_state flag indicates that the driver should use the current board VGA register state to set text mode 3. This overrides the default mode 3 settings.

```
int myvga_console_attach(struct controller *ctlr)
{
  int status = -1;  /* Failure */
```

```
  vga_use_orig_state = 1;
  status = install_vga_console(ctlr);

  if (status == -1) {
    printf("myvga: could not install vga console support\n");
  }

  return(status);
}
```

If the driver needs to reinitialize the graphics adapter when the user presses the hardware system halt button, the `console_attach` interface should also register a `saveterm` interface. This action is not usually needed unless the graphics board has a separate chip for console text.

The display driver registers a `saveterm` interface by calling the `drvr_register_saveterm` routine from its `console_attach` interface. For example, if the myvga device driver needs to provide a `saveterm` interface, it can register the interface as follows:

```
int myvga_console_attach(struct controller *ctlr)
{
  int status = -1;

  vga_use_orig_state = 1;
  status = install_vga_console(ctlr);

  if (status != -1) {
      drvr_register_saveterm(enable_vga,
                             (caddr_t)vgap->sf.screen_handle,
                             DRVR_REGISTER);
  }

  if (status == -1) {
      printf("myvga: could not install console support\n");
  }

  return(status);
}
```

The kernel calls the `saveterm` interface before it halts the system so that the driver can reenable the console (system firmware).

2.5 Handling Interrupts

Interrupt-handler interfaces are called whenever an interrupt event occurs. Interrupt handlers are optional for graphics devices, depending on the hardware. For example, on some boards the vertical retrace interrupt is the best time to change the cursor shape or the colormap with no visible side

effects. Most VGA boards do not need to use this interrupt because they buffer such events and perform them at an appropriate time.

Interrupt-handling interfaces for graphics boards are no different from the interrupt-handling interfaces described in *Writing Device Drivers: Tutorial*. Refer to that book for information on how to write them.

2.6 Handling Workstation ioctl Requests

User programs send commands to a device in one of two ways:

- As direct requests to the hardware registers and memory
- As `ioctl` requests that are dispatched to the driver through the Workstation Subsystem

When designing a display device driver, you need to determine which operations the DDX can perform directly and which operations the `ioctl` requests and driver routines can perform. For example, the DDX can perform most cursor and colormap operations if they do not need to be serviced by interrupts.

While it is faster and more convenient to access the graphics board through the DDX, there is more risk to I/O space. Requests from user space do not use kernel routines. If a problem occurs, such as a machine check, it is not easy to debug. Furthermore, a device driver is more likely to maintain binary and source compatibility from one release to the next because DIGITAL defines the interfaces, not the Open Group. You must weigh the tradeoffs between speed, risks to I/O space, and compatibility of binary and source code when deciding whether to provide graphics support through the DDX or through the device driver.

DIGITAL defines `ioctl` commands for common graphics functions, such as screen, cursor, and colormap functions. You need to supply routines to respond to these commands and register them with the Workstation Subsystem when the driver's `probe` interface executes. The routine can be a stub if the DDX or the console driver handles the function. That is, the routine can simply exit with a success status.

Table 2–1, Table 2–2, and Table 2–3 list the graphics functions that `ioctl` commands handle.

Table 2–1: Screen Functions

Function	Description
`clear_screen`	Clears the screen when running in console mode. The display driver should implement this function as a stub.
`close`	Puts the board into a state where the console driver can resume operation when the server closes or the system panics. The VGA console driver restores the graphics display to text mode, but you must implement any board-specific steps that your driver should take. If you need to reset the board when the user presses the halt button, you need to provide a `saveterm` interface in addition to a `close` interface. This is often necessary when the board uses a separate chip for console operation.
`get_set_power_level`	Gets or sets the power level for the Digital Power Management System (DPMS).
`init_screen`	Initializes the graphics board for graphics operations.
`init_screen_handle`	Initializes a driver-specific pointer, which the Workstation Subsystem passes to the driver's screen functions.
`ioctl`	Supports an optional adapter `ioctl` command. The display driver should implement this function as a stub if it does not define an optional `ioctl` command.
`map_unmap_screen`	Maps the hardware frame buffer and I/O registers to user address space, which makes them directly accessible by the DDX. The driver should call the `ws_map_region` routine to map these physical addresses to user space.
`scroll_screen`	Scrolls the console screen. The display driver should implement this function as a stub.

Table 2–2: Cursor Functions

Command	Description
`cursor_on_off`	Turns the cursor on and off.
`init_cursor_handle`	Initializes a pointer to the cursor for a particular graphics board. The Workstation Subsystem passes this pointer to the driver's cursor functions.
`load_cursor`	Loads the specified cursor and sets the cursor position on the display screen.
`recolor_cursor`	Recolors the foreground and background of the cursor.
`set_cursor_position`	Moves the cursor to a specified location on the screen.

Table 2–3: Colormap Functions

Command	Description
`clean_color_map`	Fixes any "dirty" colormap entries.
`init_color_map`	Sets the colormap to a state so that console output can be read on the screen.
`init_colormap_handle`	Initializes a pointer to a colormap for the specified graphics board. The Workstation Subsystem passes this handle to the driver's colormap functions.
`load_color_map_entry`	Loads colormap entries into the graphics board memory.
`video_off`	Turns video off. The screen saver uses this function.
`video_on`	Turns video on. The screen saver uses this function.

3

Writing a DDX

A DDX is a shared library that adds server support for a graphics device. Each type of graphics device needs its own DDX. However, DDXs can share routines, making implementation of a new DDX simpler than implementing a complete set of routines for each type of device. At first, you can supply only an initialization routine to get a new device up and running.

DDX routines can access hardware I/O registers and memory directly, if the driver provides access to them. The DDX uses this method for operations that the hardware optimizes, such as writing pixels on the screen. DDX routines can also issue `ioctl` system calls through the Workstation Subsystem to the graphics device driver. The DDX uses this method for common graphics operations, such as changing the shape of the cursor.

The following shared libraries are available with the DIGITAL UNIX X Server. Any DDX can call routines in these libraries.

- The machine-independent (MI) library contains graphics functions in their most generic form. This library implements many of the complicated algorithms associated with drawing lines and arcs. Unless your hardware provides specific support for these operations, you should call the routines in this library, or similar routines in the CFB and MFB libraries, rather than reimplementing them yourself.
- The color frame buffer (CFB) library contains graphics functions that support color and gray-scale frame buffers (8 bits/pixel). This library is also compiled for 16 and 32 bits/pixel.
- The monochrome frame buffer (MFB) library contains graphics functions that support black-and-white frame buffers (1 bit/pixel).
- The VGA library contains VGA graphical functions that are not performance intensive and, therefore, not frequently optimized by accelerated graphics adapters. DIGITAL UNIX supports VGA-compliant graphics adapters through the use of standard VGA registers. By using the routines in this library, most VGA cards can run at least minimally.

A complete generic VGA DDX is also available that makes use of the VGA library to support a VGA-compliant graphics display device. If this is the first time you have implemented a DDX for a VGA device, you can use the generic VGA DDX as a template and modify it to support your device's particular hardware specifications. Often, you can use the generic VGA

DDX with little modification to quickly get a new VGA device up and running, then customize it to optimize performance. For example, if your hardware provides special support for high-performance stippling, you would write stippling code that takes advantage of this feature by accessing registers beyond those defined by the VGA DDX.

A DDX consists of the following components:

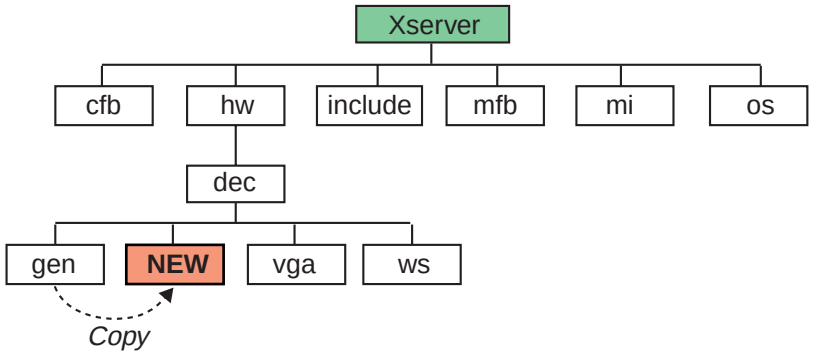
- Include files that define device-specific data structures and macros for screens, visuals, depths, and registers
- An initialization routine that allocates storage for the device-specific data structures and initializes those structures to default values
- Routines for each special-purpose graphics procedure, that is, procedures that are not handled by existing DDX routines

This chapter describes how to write a VGA-compliant DDX that is based on the generic VGA DDX.

3.1 Using the Generic VGA DDX

Figure 3–1 shows the portion of the `Xserver` directory hierarchy that you access when writing a DDX, including the shared libraries and the DDXs for the graphics devices that DIGITAL UNIX supports. The generic VGA DDX, which you use as a template, is also stored in this directory hierarchy. All directories in this hierarchy contain both source (`.c` and `.h`) files and object (`.o`) modules.

Figure 3–1: DDX Directories



The DDX directories are organized under the `Xserver` directory, as follows:

- `./Xserver/cfb`

  Contains the color frame buffer (CFB) library. Your DDX can call routines in this library to perform color graphics operations.

- `./Xserver/hw/dec`

  Contains the DIGITAL DDX code for all of the graphics cards that DIGITAL UNIX supports, including:

  – `./Xserver/hw/dec/gen`, which contains the routines and include files specific to the generic VGA DDX

  – `./Xserver/hw/dec/vga`, which contains the routines and include files that all VGA devices can use

  – `./Xserver/hw/dec/ws`, which contains the routines and include files that interface with the Workstation Subsystem in kernel space

- `./Xserver/include`

  Contains generic include files that all DDXs require. These include files define common data structures for objects such as screens, visuals, and depths.

- `./Xserver/mfb`

  Contains the monochrome frame buffer (MFB) library. Your DDX can call routines in this library to perform single plane (black-and-white) graphics operations.

- `./Xserver/mi`

  Contains the machine-independent (MI) library. Your DDX can call routines in this library to perform drawing operations that are not optimized by the hardware, the CFB library, or the MFB library.

- `./Xserver/os`

  Contains the operating system-specific (OS) library. Your DDX can call routines in this library to perform operations specific to DIGITAL UNIX.

To use the generic VGA DDX as a template for a new VGA DDX:

1. Create a directory for your new DDX. Place this directory under `./Xserver/hw/dec`. Figure 3–1 shows this directory as `NEW`.
2. Copy the source files from `./Xserver/hw/dec/gen` to the directory dedicated to your DDX.
3. Change the prefixes of all source files, device-specific data structures, and routines to reflect the name of your DDX. The generic VGA DDX uses the prefix `gen`. You can use a global search and replace operation.

3.2 Defining the Characteristics of the Screen

Three data structures describe the characteristics of any graphics display device:

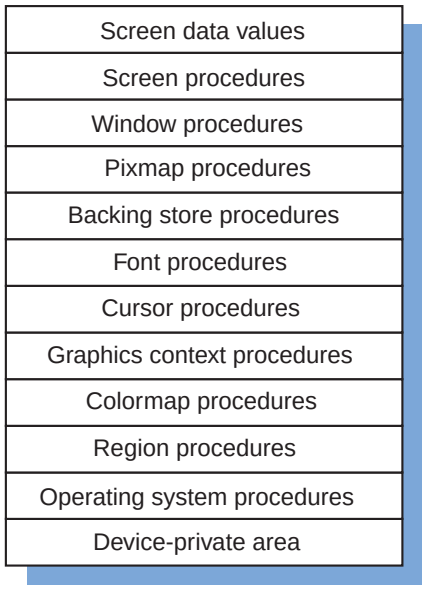
- The ScreenRec structure defines the screen itself, including the screen size and the routines that perform graphics functions.
- The VisualRec structure defines the range of colors available on the screen.
- The DepthRec structure describes the depths supported by the screen, including the number of bits/pixel that represent a color on the screen.

The following sections describe these structures in more detail. For a complete description, see *The X Window System Server*.

3.2.1 The ScreenRec Structure

The ScreenRec structure defines the screen itself, including the screen size and the routines that perform graphics operations. This structure is similar in content and purpose to the ws_screen_descriptor that the display device driver defines. The DDX dynamically allocates one ScreenRec structure for each of its screens when the server starts up, and initializes the structure as described in Section 3.4. Figure 3–2 shows the basic format of this structure.

Figure 3–2: Format of the ScreenRec structure



ZK-1242U-AI

The ScreenRec structure contains the following information:

- Screen data values define the screen height and width, default colormap, the number of depths and visuals, and other characteristcs of the screen.
- Screen procedures specify the routines that perform screen operations, such as turning the screen saver on and off or freeing the resources used by the screen.
- Window procedures specify the routines that perform window operations such as creating and destroying or setting the position of a window.
- Pixmap procedures specify the routines that create and destroy graphics objects, or pixmaps.
- Backing store procedures specify the routines that restore, clear, and expose different areas of a window.
- Font procedures specify the routines that manage fonts.
- Cursor procedures specify the routines that perform cursor functions, such as displaying the cursor and setting its position on the screen.
- Graphics context procedures specify the routines that store information for graphics output.
- Colormap procedures specify the routines that install and destroy the arrays of colors that are available on a screen.
- Region procedures specify the routines that create, copy, destroy, and perform set operations on areas of a screen.
- Operating system procedures specify the routines that perform operations that are specific to the DIGITAL UNIX operating system.
- Device-private area is space that the DDX can use for its own purpose.

3.2.2 The VisualRec Structure

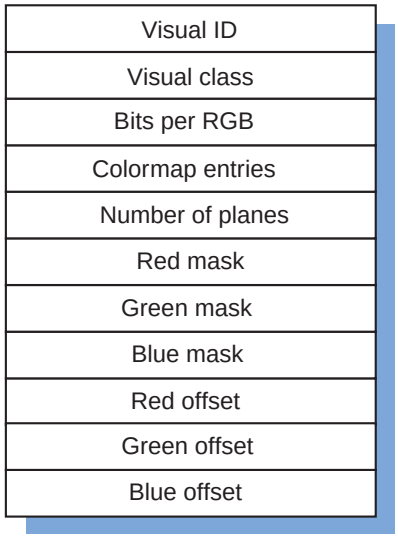
Visuals define the range of colors that can be displayed on the screen. There are several visual classes:

- `StaticGray` uses 1 bit/pixel to represent black and white.
- `GrayScale` and `StaticColor` represent colors or shades of gray in 4 bits/pixel. These visual classes can define up to 16 colors or shades of gray, which are stored in a colormap in graphics hardware memory.
- `PseudoColor` represents colors in 8 bits/pixel. This visual class can define up to 256 colors, which are stored in a colormap in the graphics hardware memory.
- `TrueColor` represents colors in three 24-bit masks — one each for red, green, and blue primary color. Using this visual class, you can display

over 16,000,000 colors. The colors are not stored in a colormap; they are mixed when graphics objects are displayed.

The `VisualRec` structure defines the characteristics of a single visual class. Figure 3–3 shows the format of this structure.

Figure 3–3: Format of the VisualRec Structure



ZK-1243U-AI

The `VisualRec` structure contains the following information:

- The visual ID contains the client ID for the colormap associated with the visual class. The X Window System initializes this structure member to the next available ID.
- The visual class identifies the type of visual, either `StaticGray`, `GrayScale`, `StaticColor`, `PseudoColor`, or `TrueColor`.
- The bits per RGB indicate how many significant bits in a color cell are used to store the red, green, and blue primary colors.
- Colormap entries are used only for `GrayScale`, `StaticColor`, and `PseudoColor` visual classes. This structure member indicates the number of entries in a colormap for this visual class.
- Number of planes indicates the number of bits per pixel.
- The `PseudoColor` visual class uses the red, green, and blue masks to indicate the valid bits that can appear in the red, green, and blue fields of a pixel value.

- The `TrueColor` visual class uses the red, green, and blue offsets to indicate the range within the 24-bit address that contains the color and intensity for a pixel.

If a structure member is not used for a particular visual class, you should set it to zero.

The DDX defines each visual class that it can support in an array of `VisualRec` structures. For example, the generic VGA DDX can support static gray, static gray scale, static color, and pseudocolor. It defines a static array of `VisualRec` structures, as follows:

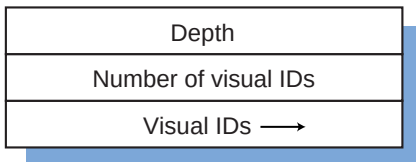
```
static VisualRec proto_visuals4[] = {
    {0, StaticGray, 6, 16, 4, 0, 0, 0, 0, 0},
    {0, GrayScale, 6, 16, 4, 0, 0, 0, 0, 0},
    {0, StaticColor, 6, 16, 4, 0xe0, 0x1c, 0x03, 5, 2, 0},
    {0, PseudoColor, 6, 16, 4, 0, 0, 0, 0, 0, 0}
};
```

The generic VGA DDX uses the `proto_visuals` array to set up a static structure for each visual class that it can support. When the server starts up, the DDX picks one visual class and uses the prototype structure for that class to initialize a dynamic visual structure.

3.2.3 The DepthRec Structure

The depth of a screen or pixmap is defined as the number of bits/pixel. The depth is defined in a `DepthRec` structure. Figure 3–4 shows the format of this structure.

Figure 3–4: Format of the DepthRec Structure



ZK-1245U-AI

The `DepthRec` structure contains the following information:

- The depth is the number of bits per pixel value.
- The number of visual IDs corresponds to the number of visual classes supported by the depth.
- The visual ID pointer is the address of an array that contains the client ID of each visual class that the depth supports.

The following array defines two DepthRec structures. The first is for a black-and-white screen with a depth of 1; no visual IDs are needed in this case. The second structure is for a depth of 4, with the NUMVISUALS constant defining the number of colors supported by that depth, and VIDs defining a pointer to an array of visual IDs for those colors.

```
static DepthRec depths4[] = {
  /* depth      numVid          vids */
  {1,           0,              NULL},
  {4,           NUMVISUALS,     VIDs}
};
```

3.3 Defining Registers and Frame Buffers

When a display device driver is initialized during system startup, it creates I/O handles to the base addresses of registers and memory on the graphics board. It defines individual registers and frame buffers as offsets from those base addresses. The DDX defines a similar set of objects to describe the registers and memory that it needs to access on the graphics board.

Section 2.1.2 shows how the myvga device driver defines two registers as offsets from the base address of an I/O handle. The DDX defines these registers also, supplying the same offsets, as follows:

```
#define SEQ_INDEX       0x3C4
#define SEQ_DATA        0x3C5

#define CRTC_INDEX      0x3D4
#define CRTC_DATA       0x3D5
```

A graphics device can have many registers. Consult your hardware specification to determine the correct offsets for the registers and memory locations on your graphics board.

Often, the DDX needs to initialize a register to some predefined set of values, for example, to set graphics modes. The DDX may be able to define an array that contains those values and write the array to the register in a single step, if the hardware allows. For example, the DDX for the generic VGA device can initialize the CRTC register to the following values by writing the VGA_CRTC array to the register:

```
#define CRTC_COUNT      25

unsigned char VGA_CRTC[CRTC_COUNT] =
                {0x5f, 0x4f, 0x50, 0x82, 0x54, 0x80, 0x0b, 0x3e,
                0x00, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x59,
                0xea, 0x8c, 0xdf, 0x28, 0x00, 0xe7, 0x04, 0xe3,
                0xff};
```

You must be careful when initializing registers in this way, especially on accelerated boards. The DDX may not be able to access a register until vertical refresh (vsync) occurs or it may need to wait until bus activity or direct memory access (DMA) stops. Consult your hardware specification for information on how to access registers.

When the Alpha processor detects two write operations in a row to the same location, it optimizes the sequence by ignoring the first write. This optimization is called “merging writes.” In normal system memory, this is fine as only the second value is usually needed. However, when the memory represents a hardware register, this is not a desirable optimization. If the two writes represent two commands to the graphics board, only the second command will reach the board.

To prevent the Alpha processor from doing this optimization, you must perform a memory barrier after every write or before every read, or both, to make sure all the registers on the board are flushed and all commands are executed. For more information on memory barriers, see the *Alpha Architecture Manual*.

Memory barriers are expensive as they lock up the CPU while each register is written and all write buffers are flushed. This may be unavoidable when performing writes, though some optimization may be possible. For example, if you know that you need to set registers 1, 2, and 3 with data, and that the instruction starts when you touch register 4, you do not need to perform a memory barrier until all four registers are set.

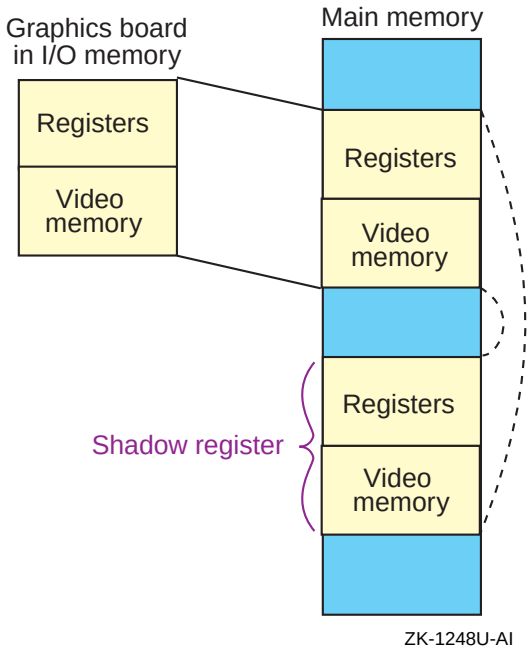
When reading register values, you can avoid memory barriers altogether by using a shadow register. A shadow register contains the current values stored in each register. Whenever you need to read the value of a register, read it from the shadow register to avoid going to the hardware and locking up the CPU.

The generic VGA DDX defines a shadow register as follows:

```
typedef struct _genShadowReg {
    unsigned int   map_mask;
    unsigned int   sr_reg;
    unsigned int   enable_sr;
    unsigned int   func_sel;
    unsigned int   read_map;
    unsigned int   rw_mode;
    unsigned int   bitmask;
    unsigned long  FrameBufferAddress;
    unsigned int   MemorySize;
    unsigned int   OffScreenLines;
    unsigned int   VideoMode;
    void           (* InstallColormap)();
} genShadowRegRec, *genShadowRegPtr;
```

Figure 3–5 shows the memory map that the display driver creates during system startup, and the shadow register that the DDX creates in system memory. Every time the DDX writes a value to the hardware, it also writes the value to the shadow register.

Figure 3–5: Memory Map and Shadow Registers



A DDX usually defines macros to write data to hardware registers and their corresponding members of a shadow register. Using a macro ensures that both the register and the shadow register always contain the same information. For example, the following macro writes a value to the SEQ register:

```
#define MAP_MASK(a)  {                  \
    unsigned char value = 0x0f & (a);   \
    if (value != pShadow->map_mask) {   \
        OUTP(SEQ_INDEX, 0x02);          \
        OUTP(SEQ_DATA, (value));        \
        pShadow->map_mask = (value);    \
    }                                   \
}
```

3.4 Writing a DDX Initialization Routine

A static server is a single, monolithic server in which all modules are always loaded, regardless of whether the user needs them during any

server invocation. DIGITAL UNIX uses a static server for debugging purposes only. Your DDX may also need to operate within a static server environment.

A loadable server is made up of shared libraries that get loaded based on the devices connected to the system and on information in the `Xserver.conf` file. This makes the server as small as possible for any server invocation. DIGITAL UNIX usually uses the loadable server.

To operate within a static server environment, the DDX needs to provide a `ScreenInit` routine to perform initialization when the server starts up. To operate within a loadable server environment, the DDX needs to provide a `ScreenInit_base` routine. Both routines take the following arguments:

- `index`

  Specifies the screen number. The server assigns a screen number to the screen as it encounters screens at startup, starting at 0.

- `pScreen`

  Specifies a pointer to a `ScreenRec` structure for this graphics display. The initialization routine allocates and initializes this structure.

- `argc`

  Specifies the number of arguments the user included in the command line to start the server.

- `argv`

  Specifies a pointer to an array that contains the command line arguments. The DDX can read this array but cannot modify it.

The difference between `ScreenInit` and `ScreenInit_base` is in how they are invoked:

- In a static server environment, the DDX does not need to be loaded; it is part of the server image. When the server detects a graphics device on the system, it calls the `ScreenInit` routine for that device. This routine is responsible for performing initializations required by the particular device type. A VGA-compliant DDX, for example, can call `vgaScreenInit_base` to perform common VGA initializations, then call its own `ScreenInit_base` routine to perform any device-specific initializations that it requires.
- In a loadable server environment, the `vgaScreenInit_base` is invoked during server startup because it is specified in the `Xserver.conf` file. The `vgaScreenInit_base` routine performs the common VGA initializations, then calls the `ScreenInit_base` routine for each type of VGA device detected on the system. The device-specific

ScreenInit_base routine needs to perform only device-specific initializations.

For example, the generic VGA DDX defines a ScreenInit routine that performs common VGA initializations by calling vgaScreenInit_base, then performs DDX-specific initializations by calling genScreenInit_base, as follows:

```
Bool genScreenInit(int index,
     ScreenPtr pScreen,
                int argc,
                char **argv)
{
    int status;

    status = vgaScreenInit_base(index, pScreen, argc, argv);
    if (status == FALSE)
        return status;
    return genScreenInit_base(index, pScreen, argc, argv);
}
```

The following sections describe the DDX-specific initializations that the ScreenInit_base routine performs.

3.4.1 Setting Up the Screen Private Area

Every DDX has a private area within the ScreenRec structure in which to store any information it needs. For example, the generic VGA DDX stores plane mask and depth information. The device-independent portion of the server is unable to access the data within the private area.

The generic DDX can have only one index into its private area, one server generation, and one visual, even if the system consists of multiple screens (as in a multiheaded system). The generic VGA DDX uses the following if statement to ensure that only one instance of each of these objects is created:

```
    if (genGeneration != serverGeneration) {
        genScreenPrivateIndex = AllocateScreenPrivateIndex();
        if (genScreenPrivateIndex < 0)
            return FALSE;
        genGeneration = serverGeneration;

       /* Set up visual IDs */
       visuals = proto_visuals4 + StaticColor;
       for (i = 0; i < NUMVISUALS; i++) {
    visuals[i].vid = FakeClientID(0);
           VIDs[i] = visuals[i].vid;
       }
    }
```

This example uses the StaticColor visual class only. You may want to check the value of the –vclass command line option to see if the user has specified a visual class. The FakeClientID routine assigns a client ID to the visual class. Passing the argument 0 (zero) causes the routine to return the next available ID.

After it has an index and a visual class, the DDX can allocate memory for its private area. The DDX should call xalloc to allocate the memory, then initialize the ScreenRec structure with the address of the private area. For example:

```
genActiveVgaPriv =
   (vgaScreenPrivPtr) xalloc (sizeof (vgaScreenPrivRec));
if (!genActiveVgaPriv)
    return FALSE;

pScreen->devPrivates[genScreenPrivateIndex].ptr =
(pointer)genActiveVgaPriv;
```

3.4.2 Creating the Memory Map

In kernel space, the display device driver creates I/O handles for the hardware registers and allocates memory when its probe interface executes at system startup. However, it does not map those I/O handles to system memory until the DDX requests it. To create the memory map, the DDX issues ioctl system calls, which get passed through the Workstation Subsystem to the display driver. The genInitScreen_base routine invokes the following subroutine to create the memory map, passing its screen index as an argument:

```
genInitVGA(index);
```

Within the genInitVGA subroutine, the DDX and the display driver share information in two data structures that the display driver defines:

- ws_map_control

  The DDX and display driver should coordinate to determine what information they need to share in this data structure. For the generic VGA DDX, this data structure contains the ID of the screen, the depth to which the screen is mapped, and a flag that indicates the current state of the screen.

- ws_depth_descriptor

  This data structure contains information that describes the depths that the screen supports, including the addresses of memory and registers in both kernel space and user space.

To map memory and registers, the DDX makes a MAP_SCREEN_AT_DEPTH ioctl system call, passing a ws_map_control structure to the driver.

This ioctl command is handled by the driver's map_unmap_screen routine, which creates the memory map. For example, the genInitVGA routine allocates a ws_map_control structure and passes the structure to the display driver, as follows:

```
if (map == NULL)
    map = (ws_map_control *) Xcalloc(sizeof(map));
if (!init_flags[index]) {
    map->map_unmap = MAP_SCREEN;
    map->screen = index;
    if ((err=ioctl(wsFd, MAP_SCREEN_AT_DEPTH, map))) {
         perror("Failed to map screen at depth");
        exit (1);
    }
    init_flags[index] = 1;
}
```

The init_flags array contains a flag that indicates whether the memory has been mapped already, in case this routine is called at a later time.

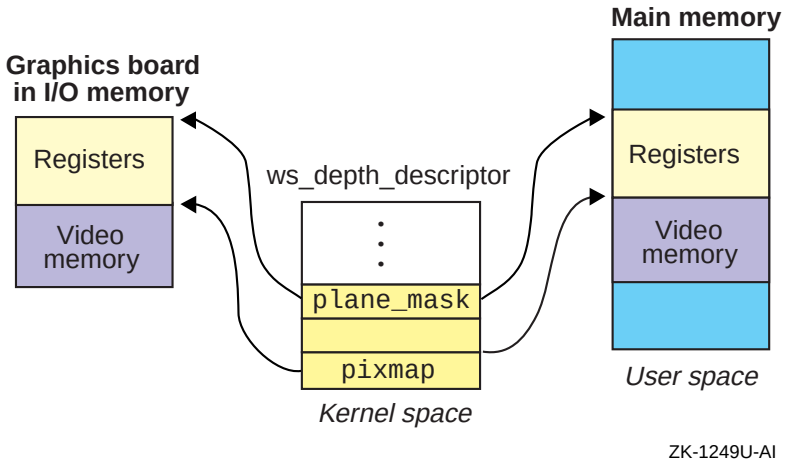
The DDX makes a GET_DEPTH_INFO ioctl system call to retrieve depth information from the device, as follows:

```
if (dpth == NULL)
    dpth = (ws_depth_descriptor *)
     Xcalloc(sizeof(ws_depth_descriptor));
dpth->screen = index;
if ((err=ioctl(wsFd, GET_DEPTH_INFO, dpth))) {
    perror("Failed to get depth info");
    exit (1);
}
```

When the genInitVGA subroutine returns, the ws_depth_descriptor contains the base addresses of the memory and I/O registers, as shown in Figure 3–6. The DDX initialization routine can save these addresses in its private area. For example:

```
genActiveVgaPriv->vgaregs = (pointer) dpth->plane_mask;
genActiveVgaPriv->firstScreenMem = (pointer) dpth->pixmap;
```

Figure 3–6: Mapping Memory Through the Depth Descriptor



3.4.3 Allocating and Initializing the Shadow Register

The shadow register stores the current values of hardware registers so that the DDX can avoid reading from the hardware, which slows performance. If you define a shadow register in the DDX, place a pointer to the shadow register in its private area. The DDX's `ScreenInit_base` routine performs this task.

The generic VGA DDX uses `xalloc` to allocate the shadow register, and it initializes the private area with a pointer to the structure, as follows:

```
pShadow = (genShadowRegPtr) xalloc(sizeof (genShadowRegRec));
if (!pShadow)
    return FALSE;

genActiveVgaPriv->avail = (pointer) pShadow;
```

After it allocates memory for the structure, the generic VGA DDX initializes the shadow register with initial values that ensure that the structure will be updated the first time the DDX writes to the registers. It then calls macros, such as the `MAP_MASK` macro defined in Section 3.3, to write initial values to the registers. For example:

```
pShadow->FrameBufferAddress =
             (unsigned long) genActiveVgaPriv->firstScreenMem;
pShadow->map_mask = 0xFF;
pShadow->sr_reg     = 0xFF;
pShadow->enable_sr  = 0xFF;
pShadow->func_sel   = 0xFF;
pShadow->read_map   = 0xFF;
```

```
pShadow->rw_mode    = 0xFF;
pShadow->bitmask    = 0xFF;

MAP_MASK(0);
SR_REG(0);
ENABLE_SR(0);
FUNC_SEL(0);
READ_MAP(0);
WR_MODE(0);
BITMASK(0);
```

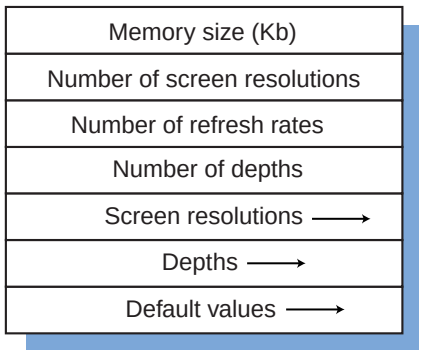
3.4.4 Setting the Screen Mode

When users start up the server, they can use command line arguments to specify a resolution, refresh rate, or depth for the screen. These values may or may not be valid for the graphics device. To pick the valid values that are closest to those requested by the user, the DDX should call the `LS_GetVideoMode` routine. This is a convenience routine that DIGITAL UNIX provides as part of the loadable X Server library.

The `LS_GetVideoMode` routine compares the values requested by the user against a data structure that specifies all possible resolutions, refresh rates, and depths that are valid for the graphics device. It returns the valid value that is closest to the one the user requested. If the user does not specify command line arguments, it returns default values, which are also defined in the structure.

Before the DDX can call the `LS_GetVideoMode` routine, you must define a `ValidModeRec` structure and initialize it with the appropriate values for the device. Figure 3–7 shows the format of the `ValidModeRec` structure.

Figure 3–7: Format of the ValidModeRec Structure



ZK-1244U-AI

The generic VGA DDX defines the `ValidModeRec` structure globally along with constants for valid values. The `genWidHt` array defines the valid

width and height values (the generic VGA DDX supports only 640x480 pixels). The genVsyncs array defines the valid refresh rates (the generic VGA DDX supports a refresh rate of 60). The genDepth array defines the valid depths (the generic VGA DDX supports a depth of four). The genIndices array defines the default values. For example:

```
#define RESOLUTIONS sizeof genWidHt / sizeof genWidHt[0]
#define REFRESHES   sizeof genVsyncs / sizeof genVsyncs[0]
#define DEPTHS   sizeof genDepth / sizeof genDepth[0]

wid_ht genWidHt[] = {
    {640,  480}
};
short genVsyncs[] = {60};
short genDepth[] = {4};
short genIndices[DEPTHS][RESOLUTIONS][REFRESHES] =
 {m640_AT_60 | DEF_MODE | DEF_VSYNC | DEF_DEPTH};

ValidModeRec genModes = {0, RESOLUTIONS, REFRESHES, DEPTHS,
        genWidHt, genVsyncs, genDepth, genIndices[0][0]}
```

The one piece of information in this data structure that the DDX does not define globally is the total memory size. The initialization routine must initialize this structure member before calling the LS_GetVideoMode routine. To determine the total memory size, the DDX should query the graphics board. Because this command is hardware specific, the generic VGA DDX uses a hardcoded value of 512 Kb. You should replace this code with the appropriate command for your graphics board, as described in your hardware specifications.

The generic VGA DDX initializes the total memory size and calls the LS_GetVideoMode routine as follows.

```
  mem_size = QUART_MEG;
  genModes.mem_kilos = 512;

  pShadow->VideoMode = LS_GetVideoMode(pScreen, &genModes);
```

3.4.5 Providing Information for Cursor Handling

The DDX and the display driver share information in the ws_screen_descriptor structure about the screen height and width so that they can manage the cursor. The display device driver initializes the structure when its probe interface executes. However, users may change the screen size when they invoke the server with the –screen option. Therefore, the DDX must initialize the ws_screen_descriptor structure with the correct screen information to make sure the driver has the same values.

The DDX issues a SET_SCREEN_INFO ioctl command to pass the information to the device driver, which initializes its own copy of the structure in kernel space. For example:

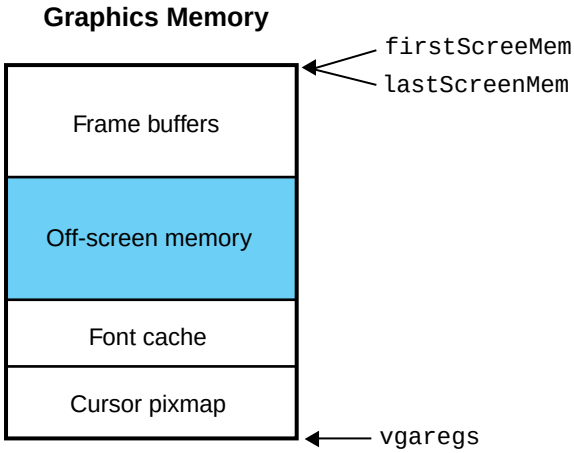
```
void genSetInfo(ScreenPtr pScreen, int index)
{
    ws_screen_descriptor wsd;

    wsd.screen = index;
    wsd.width = pScreen->width;
    wsd.height = pScreen->height;
    ioctl(wsFd, SET_SCREEN_INFO, &wsd);
    screenDesc[index].width = pScreen->width;
    screenDesc[index].height = pScreen->height;
}
```

The sizer utility also uses this structure when it displays information about the system configuration. For example, when the user invokes sizer with the –gr option, the utility displays the size and resolution of each screen on the system. When the user invokes sizer with the –gt option, the utility displays the device driver ID string for each graphics screen.

3.4.6 Calculating Memory Offsets

The DDX can reserve areas of memory on the graphics board for frame buffers and other purposes. Many VGA DDXs also reserve space for the cursor pixmap and font cache. The memory that remains is the off-screen memory, which the DDX can use for any purpose. For example, some DDXs cache pixmaps in off-screen memory, where the pixmaps can be directly updated. Figure 3–8 shows a typical layout of graphics memory, including space for frame buffers, off-screen memory, a font cache, and cursor pixmap.

Figure 3–8: Off-Screen Memory



ZK-1250U-AI

The generic VGA DDX calculates the size of frame buffers and off-screen memory, then saves these sizes in the shadow register, as follows:

```
mem_size = ldexp(262144, mem_size);
  off_screen = 2 * mem_size / pScreen->width - pScreen->height;
  pShadow->MemorySize = mem_size;
  pShadow->OffScreenLines = off_screen;
```

3.4.7 Providing Information for the xdpyinfo Utility

The `xdpyinfo` utility displays information about the server and its capabilities, such as the kinds of screens and visuals that are available. The DDX is responsible for providing information to `xdpyinfo` about the screen width, screen height, and root depth.

For example, the generic VGA DDX provides information to `xdpyinfo` based on the following calculations:

```
  if (screenArgs[screen_num].flags & ARG_DPIX)
      dpix = screenArgs[screen_num].dpix;
  else
      dpix =  ((pScreen->width * 254) +
               (wsp->screenDesc->monitor_type.mm_width * 5)) /
               (wsp->screenDesc->monitor_type.mm_width * 10);

  if (screenArgs[screen_num].flags & ARG_DPIY)
      dpiy = screenArgs[screen_num].dpiy;
  else
      dpiy =  ((pScreen->height * 254) +
               (wsp->screenDesc->monitor_type.mm_height * 5)) /
```

```
                (wsp->screenDesc->monitor_type.mm_height * 10);

  pScreen->rootDepth = 4;
```

The generic DDX either finds the screen width in a `ScreenArgsRec` structure or calculates the width from information in the `ScreenRec` and `ws_screen_descriptor` structure. It returns the screen height in a similar way and hardcodes the root depth to the number 4.

3.4.8 Initializing the Graphics Hardware

The DDX initializes the graphics hardware by performing whatever steps are necessary to make the hardware ready to use. For example, the generic VGA DDX starts the sequencer, sets the color mode, unlocks I/O registers, sets the display modes, and gives the display access to the color palette. When the hardware has been initialized, it is no longer in text mode; it is in graphics mode.

Note

On systems that interface to BIOS instructions, not all of these initializations are necessary. A single BIOS instruction can initialize the graphics modes. However, Digital UNIX does not interface to BIOS instructions. You must rely on your hardware specifications for the steps to initialize the hardware.

3.4.9 Initializing the ScreenRec Structure

The `vgaScreenInit_base` routine initializes the `ScreenRec` structure with screen data values and the names of routines to perform VGA-compliant graphics functions. Screen data values define the characteristics of the screen, such as height and width. You will probably want to supply data values that more accurately describe the characteristics of your graphics board. You may also want to replace some of the VGA-compliant routines with routines that optimize performance for your board.

The following sections describe the members of the `ScreenRec` structure that hold screen data values and routine names.

3.4.9.1 Screen Data Values

The `ScreenRec` structure contains members that describe the screen height and width, number of depths, colormap and backing store support, information on visual classes, and the address of device-specific functions.

The generic VGA DDX initializes these structure members to values that apply to most VGA-compliant graphics devices. You should set the following structure members to values that more accurately reflect your hardware capabilities:

```
pScreen->mmWidth =
    ((pScreen->width * 254L)+ (dpix * 5)) / (dpix * 10);
pScreen->mmHeight =
    ((pScreen->height * 254L) + (dpiy * 5)) / (dpiy * 10);

pScreen->numDepths = NUMDEPTHS;
pScreen->allowedDepths = depths;

pScreen->minInstalledCmaps = 1;
pScreen->maxInstalledCmaps = 1;
pScreen->backingStoreSupport = Always;
pScreen->saveUnderSupport = Always;

pScreen->numVisuals = NUMVISUALS;
pScreen->visuals = visuals;
pScreen->rootVisual = visuals->vid;

pScreen->devPrivates[vgaScreenPrivateIndex].ptr =
    (pointer) &genDrawFuncs;
```

3.4.9.2 Screen Procedures

Table 3–1 shows the screen procedures that you need to define in the `ScreenRec` structure, including the default values that the `vgaScreenInit_base` routine supplies.

Table 3–1: Screen Procedures

Procedure	Description
`CloseScreen`	Closes the screen and frees resources that the screen uses.
	VGA default: None. (The DDX must provide this routine.)
`QueryBestSize`	Returns the largest possible size for a cursor, tile, or stipple.
	VGA default: `mfbQueryBestSize`
`SaveScreen`	Turns the screen saver on or off, or forces the screen saver to change modes.
	VGA default: `colorSaveScreen`
`GetImage`	Extracts a portion of a drawable image and stores it in an `XImage` structure.

Table 3–1: Screen Procedures (cont.)

	VGA default: `vgaGetImage`
`GetSpans`	Extracts a number of lines from the drawable image and stores them in a buffer.
	VGA default: `vgaGetSpans`
`PointerNonInterestBox`	Called by the DIX, indicates whether pointer events can be ignored.
	VGA default: `colorPointerNonInterestBox`
`SourceValidate`	Ensures that a drawable image is ready to be copied.
	VGA default: NULL pointer

The generic VGA DDX defines only a `CloseScreen` procedure. All other procedures default to the values that `vgaScreenInit_base` provides. For example:

```
pScreen->CloseScreen = genCloseScreen;
```

3.4.9.3 Cursor Procedures

Table 3–2 shows the cursor procedures that you need to define in the `ScreenRec` structure.

Table 3–2: Cursor Procedures

Procedure	Description
`ConstrainCursor`	Forces a cursor to stay within a given area.
	VGA default: None
`CursorLimits`	Returns the physical constraints a cursor would have if `ConstrainCursor` was called with a specified area.
	VGA default: None
`DisplayCursor`	Displays the cursor on the screen.
	VGA default: None
`RealizeCursor`	Allocates and initializes device-specific resources for a cursor before displaying the cursor on the screen.
	VGA default: None
`UnrealizeCursor`	Deallocates the resources for a cursor.

Table 3–2: Cursor Procedures (cont.)

Procedure	Description
	VGA default: None
RecolorCursor	Changes the color of a cursor.
	VGA default: None
SetCursorPosition	Moves the cursor to the specified position on the screen.
	VGA default: None

Because the .PN vgaScreenInit_base routine does not define cursor procedures, the generic VGA DDX must define these procedures, as follows:

```
pScreen->DisplayCursor     = colorDisplayCursor;
pScreen->RecolorCursor     = colorRecolorCursor;
pScreen->ConstrainCursor   = colorConstrainCursor;
pScreen->CursorLimits      = colorCursorLimits;
pScreen->RealizeCursor     = colorRealizeCursor;
pScreen->UnrealizeCursor   = colorUnrealizeCursor;
pScreen->SetCursorPosition = colorSetCursorPosition;
```

3.4.9.4 Colormap Procedures

Table 3–3 shows the colormap procedures that you need to define in the ScreenRec structure.

Table 3–3: Colormap Procedures

Procedure	Description
CreateColormap	Allocates resources for a new colormap.
	VGA default: vgaCreateColormap
DestroyColormap	Deallocates resources for a colormap that is to be destroyed.
	VGA default: vgaDestroyColormap
InstallColormap	Changes the mapping of pixel values to colors that match a new colormap.
	VGA default: None
UninstallColormap	Removes a colormap and replaces it with the default colormap for the screen.
	VGA default: None
ListInstalledColormaps	Returns the numbers and resource IDs of all colormaps installed on the screen.

Table 3–3: Colormap Procedures (cont.)

Procedure	Description
	VGA default: None
StoreColors	Stores one or more colors in a colormap.
	VGA default: None
ResolveColor	Adjusts the server's universal color values to values that are appropriate for the screen.
	VGA default: vgaResolveColor

Although the vgaScreenInit_base routine defines colormap procedures, the generic VGA DDX defines its own procedures to fill in colormap values. When it initializes the ScreenRec structure with the names of these procedures, it replaces the VGA procedures. For example:

```
pScreen->CreateColormap = genCreateColormap;
pShadow->InstallColormap = pScreen->InstallColormap;
pScreen->InstallColormap = genInstallColormap;
```

3.4.9.5 Window Procedures

Table 3–4 shows the window procedures that you need to define in the ScreenRec structure.

Table 3–4: Window Procedures

Procedure	Description
CreateWindow	Creates a new window.
	VGA default: vgaCreateWindow
DestroyWindow	Frees resources associated with a window that is about to be destroyed.
	VGA default: vgaDestroyWindow
PositionWindow	Updates data structures to a new window position prior to moving the window to that position.
	VGA default: vgaPositionWindow
ChangeWindowAttributes	Updates window attributes and any resources that depend on those attributes.
	VGA default: vgaChangeWindowAttributes
RealizeWindow	Allocates and initializes resources a window needs if it is going to be mapped to the screen.

Table 3–4: Window Procedures (cont.)

Procedure	Description
	VGA default: `vgaMapWindow`
`UnrealizeWindow`	Deallocates the resources that `RealizeWindow` allocates when a window is unmapped.
	VGA default: `vgaUnmapWindow`
`ValidateTree`	Computes the border clip region and window clip region for every marked window in a window tree when a portion of the tree changes.
	VGA default: `miValidateTree`
`PostValidateTree`	Performs device-dependent operations on a window tree that has been validated.
	VGA default: NULL pointer
`WindowExposures`	Paints the exposed areas of any window that has been validated.
	VGA default: `miWindowExposures`
`PaintWindowBackground`	Paints the exposed areas of a window background.
	VGA default: `vgaPaintWindow`
`PaintWindowBorder`	Paints the exposed areas of a window border.
	VGA default: `vgaPaintWindow`
`CopyWindow`	Copies an exposed area from a window to another place on the screen.
	VGA default: `vgaCopyWindow`
`ClearToBackground`	Draws the window background or generates exposure events for a portion of a window.
	VGA default: `miClearToBackground`
`ClipNotify`	Adjusts the window's hardware clipping resources when the clip list changes.
	VGA default: None

The generic VGA DDX does not initialize these members of the `ScreenRec` structure; it accepts the VGA-compliant procedures that `vgaScreenInit_base` defines.

3.4.9.6 Pixmap Procedures

Table 3–5 shows the pixmap procedures that you need to define in the ScreenRec structure.

Table 3–5: Pixmap Procedures

Procedure	Description
CreatePixmap	Allocates a PixmapRec structure for a new pixmap.
	VGA default: vgaCreatePixmap
DestroyPixmap	Deallocates the resources for a pixmap.
	VGA default: vgaDestroyPixmap

The generic VGA DDX does not initialize these members of the ScreenRec structure; it accepts the VGA-compliant procedures that vgaScreenInit_base defines.

3.4.9.7 Backing Store Procedures

Table 3–6 shows the backing store procedures that you need to define in the ScreenRec structure.

Table 3–6: Backing Store Procedures

Procedure	Description
SaveDoomedAreas	Saves a portion of a window that is about to be obscured or moved and places it in a backing store pixmap.
	VGA default: Null pointer
RestoreAreas	Recovers an area of a window that was previously saved to a backing store pixmap.
	VGA default: Null pointer
ExposeCopy	Copies an area from a backing store pixmap, moves it to another place on the screen, and exposes any portion of the area that is no longer obscured.
	VGA default: Null pointer
TranslateBackingStore	Allocates a new backing store pixmap, if necessary, when a window is resized or moved.
	VGA default: Null pointer
ClearBackingStore	Clears an area of a backing store pixmap.

Table 3–6: Backing Store Procedures (cont.)

Procedure	Description
	VGA default: Null pointer
DrawGuarantee	Specifies whether any upcoming graphics events will be clipped to the window's border clip region.
	VGA default: Null pointer

The generic VGA DDX does not initialize these members of the ScreenRec structure; it accepts the VGA-compliant procedures that vgaScreenInit_base defines.

3.4.9.8 Font Procedures

Table 3–7 shows the font procedures that you need to define in the ScreenRec structure.

Table 3–7: Font Procedures

Procedure	Description
RealizeFont	Initializes any font-specific data structures for a screen when a font is opened.
	VGA default: vgaRealizeFont
UnrealizeFont	Frees any resources that RealizeFont allocates.
	VGA default: vgaUnrealizeFont

The generic VGA DDX does not initialize these members of the ScreenRec structure; it accepts the VGA-compliant procedures that vgaScreenInit_base defines.

3.4.9.9 Graphics Context Procedure

Table 3–8 shows the graphics context procedure that you need to define in the ScreenRec structure.

Table 3–8: Graphics Context Procedure

Procedure	Description
CreateGC	Allocates resources needed for a new graphics context.
	VGA default: vgaCreateGC

The generic VGA DDX does not initialize these members of the ScreenRec structure; it accepts the VGA-compliant procedures that vgaScreenInit_base defines.

3.4.9.10 Region Procedures

Table 3–9 shows the region procedures that you need to define in the ScreenRec structure.

Table 3–9: Region Procedures

Procedure	Description
RegionCreate	Allocates and initializes a RegionRec structure for a new region.
	VGA default: miRegionCreate
RegionInit	Initializes the RegionRec structure for an existing region.
	VGA default: miRegionInit
RegionCopy	Copies the contents of one region to another.
	VGA default: miRegionCopy
RegionDestroy	Deallocates the resources for a region.
	VGA default: miRegionDestroy
RegionUninit	Deallocates resources for a statically allocated region but does not deallocate its RegionRec structure.
	VGA default: miRegionUninit
Intersect	Returns the intersection of two regions.
	VGA default: miIntersect
Union	Returns the union of two regions.
	VGA default: miUnion
Subtract	Returns the difference between two regions.
	VGA default: miSubtract
Inverse	Returns the inverse of a region.
	VGA default: miInverse
RegionReset	Changes the extents of a region.
	VGA default: miRegionReset
TranslateRegion	Moves a region to a new location on the screen.
	VGA default: miTranslateRegion

Table 3–9: Region Procedures (cont.)

Procedure	Description
RectIn	Determines whether a rectangle falls within a region.
	VGA default: miRectIn
PointInRegion	Determines whether a point falls within a region.
	VGA default: miPointInRegion
RegionNotEmpty	Returns true if a specified region is not empty.
	VGA default: miRegionNotEmpty
RegionEmpty	Returns true if a specified region is empty.
	VGA default: miRegionEmpty
RegionExtents	Returns the smallest rectangle that encloses a specified region.
	VGA default: miRegionExtents
RegionAppend	Concatenates two regions.
	VGA default: miRegionAppend
RegionValidate	Creates a single region from several screen areas concatenated by multiple calls to RegionAppend.
	VGA default: miRegionValidate
BitmapToRegion	Converts a bitmap to a region.
	VGA default: mfbBitmapToRegion
RectsToRegion	Creates a region from several rectangles.
	VGA default: miRectsToRegion
SendGraphicsExpose	Sends a GraphicsExpose or NoExpose event to the client for a specified region.
	VGA default: miSendGraphicsExpose

The generic VGA DDX does not initialize these members of the ScreenRec structure; it accepts the VGA-compliant procedures that vgaScreenInit_base defines.

3.4.9.11 Operating System Procedures

Table 3–10 shows the operating system procedures that you need to define in the ScreenRec structure. The DDX does not usually replace these routines.

Table 3–10: Operating System Procedures

Procedure	Description
BlockHandler	VGA default: NoopDDA
WakeupHandler	VGA default: NoopDDA
ModifyPixmapHeader	VGA default: miModifyPixmapHeader
CreateScreenResources	VGA default: miCreateScreenResources
MarkWindow	VGA default: miMarkWindow
MarkOverlappedWindows	VGA default: miMarkOverlappedWindows
ChangeSaveUnder	VGA default: miChangeSaveUnder
PostChangeSaveUnder	VGA default: miPostChangeSaveUnder
MoveWindow	VGA default: miMoveWindow
ResizeWindow	VGA default: miResizeWindoW
GetLayerWindow	VGA default: miGetLayerWindow
HandleExposures	VGA default: miHandleExposures
ReparentWindow	VGA default: NULL pointer
ChangeBorderWidth	VGA default: miChangeBorderWidth
SetShape	VGA default: miSetShape
MarkUnrealizedWindow	VGA default: miMarkUnrealizedWindow

The generic VGA DDX does not initialize these members of the ScreenRec structure; it accepts the VGA-compliant procedures that vgaScreenInit_base defines.

3.4.10 Creating a Default Colormap

Every screen needs a default colormap that defines the colors of the window and its border. (The DDX may create other colormaps, if users request them.) Every colormap is associated with a client ID, which the server assigns. The DDX can call the FakeClientID routine to assign an ID to the colormap. You should store the client ID in the defColormap member of the ScreenRec structure.

A VGA-compliant DDX can then call the vgaCreateColormap routine to allocate and initialize the default colormap for the screen. For example, the generic VGA DDX creates its default colormap as follows:

```
pScreen->defColormap = FakeClientID(0);
```

```
    if (!vgaCreateDefColormap(pScreen))
        return FALSE;
```

3.4.11 Performing Machine-Independent Initializations

After the common VGA and DDX-specific initializations have been performed and the hardware has been set to graphics mode, only a few machine-independent initializations remain.

The DDX needs to call the `miInitializeBackingStore` routine to specify the names of the routines it wants to use for backing store operations. These routines are specified in a `miBSFuncRec` structure that you define globally. For example, the generic VGA DDX defines the structure as follows:

```
static miBSFuncRec genBSFuncRec = {
  (void (*)()) vgaSaveAreas,    SaveAreas
  (void (*)()) vgaRestoreAreas, RestoreAreas
  (void (*)()) 0,               SetClipmaskRgn
  (PixmapPtr (*)()) 0,          GetImagePixmap
  (PixmapPtr (*)()) 0,          GetSpansPixmap
};
```

The DDX passes this data structure and a pointer to its `ScreenRec` structure to `miInitializeBackingStore`.

```
  miInitializeBackingStore(pScreen, &genBSFuncRec);
```

The DDX should perform the following additional machine-independent initializations as well:

```
  miScreenDevPrivateInit(pScreen, 0,
                         genActiveVgaPriv->firstScreenMem);

  {
  extern Bool wsScreenClose();
  wsScreenPrivate *wsp = (wsScreenPrivate *)
            pScreen->devPrivates[wsScreenPrivateIndex].ptr;

  /* Wrap screen close routine to avoid memory leak */
  wsp->CloseScreen = pScreen->CloseScreen;
  pScreen->CloseScreen = wsScreenClose;

  D_GENERICDrawInit(pScreen, wsp->screenDesc->screen);
  }
```

4

Writing Xserver Configuration File Entries

You define all of the devices and extensions that can be configured into the server in the `/var/X11/Xserver.conf` file. When the server starts up, it uses this file to determine which components to load.

The configuration file is divided into the following sections:

- Device configuration section, which contains entries for all graphics display devices
- Extension configuration section, which contains entries for all default and deferred extensions that are not input extensions
- Font renderer configuration section, which contains entries for all font renderers
- Input configuration section, which contains entries for all input extensions

You can also use the configuration file to define default command line arguments and to specify alternate paths for loadable libraries.

You can make entries in each section recursive so that components are tightly tied to or dependent on other components. For example, a DDX based on the generic DDX is dependent on the VGA DDX. Therefore, it is defined as a sublibrary of VGA. When loading, the server loads the VGA library, then its sublibraries. If it cannot load the VGA library, the server does not load the sublibraries.

This chapter describes the syntax of the entries you make in each section of the `/var/X11/Xserver.conf` file.

4.1 Device Configuration Section

Display device entries define the name and shared library for each DDX. All entries have the following syntax:

```
device <
   < library_name library_filename init_routine { device_name |
      < sublibrary_name sublibrary_filename init_routine device_name > }
         ⋮
   >
>
```

You supply the following parameters:

library_name
: Specifies the name of the library. For example, the library name for the VGA DDX library is `_dec_vga`.

library_filename
: Specifies the file name of the library, and consists of the `lib` prefix followed by the *library_name* parameter and the `.so` extension. For example, the library file name for the VGA DDX is `lib_dec_vga.so`.

init_routine
: Specifies the name of the initialization routine to call when this component is loaded into the server.

device_name
: Specifies the name of the device as it is known to the device driver. The `sizer –gt` command returns this name.

Individual components may have multiple sublibraries. Each device entry and subentry is searched for matching device names that the system returns. For example, the following `device` section defines the VGA and related graphics displays — QVision and ATI64:

```
device <
  < _dec_vga lib_dec_vga.so vgaScreenInit
     < _dec_triton lib_dec_triton.so pwgaScreenInit_base QVision >
     < _dec_ati64 lib_dec_ati64.so atiScreenInit_base ATI64 >
  >
>
```

In this example, the VGA device DDX library file is `lib_dec_vga.so`, and the initialization routine is `vgaScreenInit`. If the server detects a VGA device on the system, it loads `lib_dec_vga.so` and calls `vgaScreenInit`. The `vgaScreenInit` initialization routine identifies the specific VGA display device that is present — QVision or ATI64. The routine obtains a list of sublibraries and searches them for the device name. If it finds a match, it loads the library and calls the initialization routine.

4.2 Extension Configuration Section

The extension defines the name and library for all extensions, except input extensions. Entries in this section have the following syntax:

```
extensions <
   < library_name library_filename init_routine [ extension_name ]
     [ <sublibrary_name sublibrary_filename init_routine
     [ extension_name ]> ]
        ⋮
   >
>
```

You supply the following parameters for each extension:

library_name
: Specifies the name of the library.

library_filename
: Specifies the file name for the library, and consists of the `lib` prefix followed by the *library_name* parameter and the `.so` extension.

init_routine
: Specifies the name of the initialization routine to call when this component is loaded into the server.

extension_name
: Specifies the name of the extension. If you do not specify an *extension_name* parameter, the extension is treated as a default extension; it is loaded whenever the server starts up. If you specify an *extension_name* parameter, the extension is treated as a deferred extension; it is not loaded until a client requests it.

The following example defines two deferred extensions:

```
extensions <
  < extSync    libextSync.so   SyncExtensionInit  SYNC >
  < extxtest   libextxtest.so  XTestExtensionInit XTEST >
```

The following extension lists hardware-specific modules as sublibraries. If this extension does not find a device handler that matches one of the devices available on the system (in this case, PMAG-RO or PMAG-JA), it does not get loaded, and it lets the client know that the extension is not available.

```
    < xv  libxv.so   XvExtensionInit  XVideo
       < _dec_xv_tx lib_dec_xv_tx.so XvropScreenInit PMAG-RO >
       < _dec_xv_tx lib_dec_xv_tx.so XvropScreenInit PMAG-JA >
```

4.3 Font Renderer Configuration Section

Font renderers load fonts into memory and decode font characters for display on the screen. Font renderer definitions have the following syntax:

```
font_renderers <
   < library_name library_filename init_routine [ renderer_name ]
      [<sublibrary_name sublibrary_filename init_routine
      [ renderer_name ] > ]
         ⋮
   >
>
```

You supply the following parameters for each font renderer:

library_name
: Specifies the name of the library.

library_filename
: Specifies the file name for the library, and consists of the `lib` prefix followed by the *library_name* parameter and the `.so` extension.

init_routine
: Specifies the name of the initialization routine to call when this component is loaded into the server.

extension_name
: Specifies the name of the font renderer, but no renderers to date take advantage of this parameter.

The following example defines three font renderers:

```
font_renderers <
  < fr_fs      libfr_fs.so      fs_register_fpe_functions >
  < fr_Speedo  libfr_Speedo.so  SpeedoRegisterFontFileFunctions >
  < fr_Type1   libfr_Type1.so   Type1RegisterFontFileFunctions >
>
```

4.4 Input Configuration Section

Input extensions provide support for input devices other than the system pointer and keyboard. For example, input extensions add support for pointers or dial and button boxes.

Input extension definitions have the following syntax:

```
input <
   < library_name library_filename init_routine [ input_name ]
      [ < sublibrary_name sublibrary_filename init_routine
      [ input_name ] > ]
         ⋮
   >
>
```

You supply the following parameters for each input extension:

library_name
: Specifies the name of the library.

library_filename
: Specifies the file name for the library, and consists of the `lib` prefix followed by the *library_name* parameter and the `.so` extension.

init_routine
: Specifies the name of the initialization routine to call when this component is loaded into the server.

input_name
: Specifies the name of the input extension.

The following example specifies a dial and button box on `/dev/tty01`. When the input extension is initialized, it opens the `lib_dec_xi_pcm.so` shared library and calls the `XiPcmInit` initialization routine.

```
input <
  < _dec_xi_pcm lib_dec_xi_pcm.so XiPcmInit /dev/tty01 >
>
```

4.5 Other Uses of the Configuration File

The configuration file can specify an alternate loadable library path for the server to search during startup. This is especially useful when you want to keep your shared libraries in a directory separate from those shipped with the XDK.

Alternate library path definitions have the following syntax:

library_path < *path* [*:path* ...] >

You supply the following parameters for each library path:

path
: Specifies the directory path that the server should search. You can specify more than one path by separating the directory specifications with a colon. The directories are searched in the order in which they appear in the entry.

For example, the following entry specifies two paths to search for shared libraries. The server first searches `/newserver/fonts/lib/font` for shared libraries. If it cannot find the library in that directory, it searches `/usr/shlib`.

```
library_path < /newserver/fonts/lib/font:/usr/shlib >
```

Finally, you may sometimes need to modify the command line to supply default command line arguments. The `args` section of the configuration file lets you define default arguments so that the casual user does not need to be concerned with them.

Default command line argument definitions have the following syntax:

```
args < option default_value
       ⋮
>
```

You supply the following parameters for each command line argument:

option
: Specifies the command line option, including the leading dash if it is part of the syntax.

default_value
: Specifies the value assigned to the option.

For example, the following entry uses the `–vclass0` command line argument to set the default screen visual to static gray by default:

```
args <
  -vclass0 StaticGray
>
```

5

Building a Graphics Hardware Support Product

A graphics hardware support product consists of a device driver in kernel space and a DDX in user space, plus supporting files and file fragments. After you have produced the source code for these components, you must perform several steps to build a fully functional hardware support product:

1. Build the display device driver.

   You create file fragments, compile and link the device driver into a single binary module, then configure the driver into the kernel. At this point, you can perform some initial testing to see that the driver operates without the DDX.

2. Compile and link the DDX into a shared library.

   The X Server Developers Kit (XDK) provides scripts to help you build the DDX as part of the static server and as a shared library to run with the loadable server. When you have built the DDX, you can test both the DDX and the driver to see that they operate within the X Window System.

3. Package the driver, DDX, and supporting file fragments.

   You create subsets of both the DDX and the display driver and copy them onto a kit, which you can distribute to users.

This chapter describes how to build a graphics hardware support product, using the generic VGA display driver and DDX as an example.

5.1 Building the Display Device Driver

A statically configured driver is part of the kernel and loaded along with the kernel at system startup. The driver is configured along the boot path. A dynamically configured driver is loaded into the driver after system startup. It is configured with the `sysconfig` utility at run time. A graphics display driver must be configured statically.

You can configure a single binary module either statically or dynamically. You create the single binary module for the graphics display driver and configure it into the kernel as described in the following sections. For more information on building device drivers, see *Writing Device Drivers: Tutorial*.

5.1.1 Producing a Single Binary Module

To produce a single binary module:

1. Create a directory to contain the driver product files.

   ```
   # mkdir /usr/sys/io/MYVGA100
   ```

2. Copy the driver product files to the new directory.

   ```
   # cd /usr/sys/io/MYVGA100
   # cp /usr/sys/dev/myvga.c .
   # cp /usr/sys/dev/myvga.h .
   ```

3. Create a `files` file fragment.

   ```
   /usr/sys/io/MYVGA100
   # vi files
   ```

```
# This is the files file fragment for the /dec/myvga driver
# used to produce the single binary module.
#
MODULE/STATIC/myvga        standard Binary
io/MYVGA100/myvga.c        module   myvga
```

ZK-1251U-AI

The `files` file fragment provides information about the location of the driver, how the driver should be loaded, and the format of the driver files (source or binary). The `doconfig` utility uses this information to build the driver.

4. Create a `BINARY.list` file.

   ```
   # cd /usr/sys/conf
   # vi BINARY.list
   ```

```
/usr/sys/io/MYVGA100:
```

ZK-1252U-AI

The `BINARY.list` file specifies rules for building the driver into the `BINARY Makefile` file. The `sourceconfig` program uses the information in this file.

5. Create a sysconfigtab file fragment.

```
# cd /usr/sys/io/MYVGA100
# vi sysconfigtab
```

```
myvga:
  Subsystem_Description = myvga device driver
  Module_Config_Name = myvga
  MYVGA_Developer_Debug = 1
  Num_Units_Installed = 1
  Interrupt_Enable = 1
  PCI_Option = PCI_SE_Rev - 0x210, Vendor_Id - 0x1002, Device_Id - 0x4158,
     Rev - 0, Base - 0, Sub - 0, Pif - 0 Sub_Vid - 0, Sub_Did - 0,
     Vid_Mo_Flag - 1, Did_Mo_Flag - 1, Rev_Mo_Flag - 0, Base_Mo_Flag - 0,
     Sub_Mo_Flag - 0, Pif_Mo_Flag - 0, Sub_Vid_Mo_Flag - 0,
     Sub_Did_Mo_Flag - 0, Driver_Name - myvga, Type - C, Adpt_Config - N
  EISA_Option = Board_Id - ISA0099, Function_Name - Null, Driver_Name - myvga,
     Type - C, Adpt_Config - N
  ISA_Option = Board_Id - Null, Function_Name - 'MYVGA' , Driver_Name = myvga,
     Type - C, Adpt_Config - N
```

ZK-1253U-AI

The sysconfigtab file fragment contains device special file information, bus option data, and physical contiguous memory usage information for the driver. The sysconfig utility uses the information in this file when configuring the driver into the system or when returning information to the user about the driver.

6. Run the sourceconfig program.

```
# cd /usr/sys/conf
# ./sourceconfig BINARY
```

The sourceconfig program creates a Makefile file that includes your driver in the kernel. The utility uses the BINARY.list file to locate the files file fragment for your driver. It uses information in the files file fragment to create your driver's entry in the Makefile file.

7. Run the make program.

```
# cd /usr/sys/BINARY
# make myvga.mod
```

This program compiles the driver source files and produces a binary object (.mod) file.

8. Create a kernel configuration development area.

```
# cd /usr/sys/conf
# doconfig
*** KERNEL CONFIGURATION AND BUILD PROCEDURE ***
```

```
Enter a name for the kernel configuration file. [ALF]: MYVGA

Do you want to edit the configuration file? (y/n) [n] no
```

The doconfig utility creates a configuration area for building a new kernel that includes your display driver. Give this kernel a name other than your current kernel configuration until you have tested that your driver works correctly.

9. Run the sysconfigdb utility.

```
# cd /usr/sys/io/MYVGA100
# sysconfigdb -a -f sysconfigtab myvga
```

This utility adds the sysconfigtab file fragment to the /etc/sysconfigtab database on the system.

5.1.2 Statically Configuring a Single Binary Module

To statically configure a single binary module:

1. Edit or create the *NAME*.list file, where *NAME* is the name of the kernel you are building. DIGITAL recommends that you use the name of your driver. For example:

```
# cd /usr/sys/conf
# vi MYVGA.list
```

```
/usr/sys/io/MYVGA100:
```

ZK-1252U-AI

2. Run the doconfig program.

```
# cd /usr/sys/conf
# doconfig -c MYVGA
Do you want to edit the configuration file? (y/n) [n] no
```

When used with the –c option, the doconfig utility uses the configuration file that you specify to build the kernel.

3. Copy the new kernel to the root directory.

```
# cd /
# cp /usr/sys/MYVGA/vmunix /vmunix.myvga
```

5.2 Testing and Debugging the Driver

After you build the driver and configure it into the kernel, you can test that it operates as the console terminal. If problems arise, you have several options for debugging the driver.

To test the driver, you either start up the system in single-user mode or disable the X Window System. Either way, the graphics display runs in console mode and does not require the DDX. The examples in this section start up the system in single-user mode.

To disable the X Window System, move the `/sbin/rc3.d/S95xlogin` file to some other location. (You must have `root` privileges to do this.) For example:

```
# mv /sbin/rc3.d/S95xlogin /sbin/rc3.d/xS95xlogin
```

When the X Window System is disabled in this way, the startup procedure brings up the system in multiuser mode, but does not start the X Window System. A regular login prompt appears instead of the login window.

5.2.1 Testing the Driver with genvmunix

The generic `vmunix` kernel contains the VGA device driver that DIGITAL supports. You should test that the VGA console support code operates with your VGA graphics board, as follows:

```
# shutdown -h now
>>> boot -fl s -fi "genvmunix"
⋮
>
```

In most cases, the VGA console routines should operate on your graphics board. If the single-user mode prompt does not appear, the VGA console support code that DIGITAL provides is not compatible with your graphics board. To fix this problem, you need to determine why the VGA text cannot display on your graphics board. You may need to change the `vga_orig_state` and replace the VGA console routines that DIGITAL supplies, as needed.

5.2.2 Testing the Driver with the Newly Configured Kernel

When you know that your driver operates with `genvmunix`, test the driver with the kernel you created in Section 5.1.2:

```
> shutdown -h now
>>> boot -fl s -fi "vmunix.myvga"
```

As the system starts up, you should see the `printf` statements from your driver indicating that the device has been probed and attached. If these messages do not appear:

- Check that the driver appears in the list of objects in `/usr/sys/BINARY/Makefile`. If not, create a new `Makefile` file, build the single binary module, and reconfigure the driver.
- Check that the driver has been added to the `sysconfigtab` database. For example, to see if the generic VGA display driver was configured into the database, you would enter the command:

  ```
  > sysconfigdb -l myvga
  ```

  If the utility does not display your driver's entry in the database, you should rerun the `sysconfigdb` utility and reconfigure the driver.
- Look in the kernel binary file for occurrences of the message strings generated by your driver. For example:

  ```
  > strings /vmunix.myvga | grep "MYVGA"
  ```

  If these strings do not appear, the driver is not configured into the kernel. Repeat the steps in Section 5.1.1 and Section 5.1.2.

When you can successfully start up the system, run the `sizer` utility to see that the driver returns the appropriate information for your graphics board. For example, the `–gt` option returns the graphics board ID; the `–gr` option returns the screen size, in pixels:

```
> sizer -gt
VGA

> sizer -gr
640X480
```

In this example, `VGA` indicates the generic VGA driver. Your driver would return a different board ID.

5.2.3 Debugging the Driver Code

If you discover problems in the driver code, several debugging options are available:

- Analyze the crash dump file.

  If the system crashes, reboot the system with the old kernel and examine the `core` file. You can use the debugger (`dbx`) to examine the state of the system when the panic occurred. (See *Kernel Debugging*.)

- Add debugging messages to the driver.

  Adding a `printf` statement at every entry and exit point of your driver routines can show you which interfaces are causing the problem. You can turn these statements on and off with compiler definitions. For example:

```
int
myvgaattach(ctlr)
    register struct controller *ctlr;
{
⋮
#ifdef DEBUG_PRINT_ENTRY
    DPRINTF("myvgaattach(0x%lx): entry\n", (u_long)ctlr);
#endif /* DEBUG_PRINT_ENTRY */
⋮
```

  Compile the driver with the `–DDEBUG_PRINT_ENTRY` option and reconfigure it into the kernel.

- Set a debug attribute in the device attribute table to turn the debugging messages on and off.

  Using the device attribute table lets you turn debugging messages on and off without recompiling and reconfiguring the driver into the kernel. For example, the generic VGA driver defines a static variable called `myvga_developer_debug` and inititalizes it to 0 (turning debugging off by default). The variable would then be defined in the device attribute table, as follows:

```
static int myvga_developer_debug = 0;

cfg_subsys_attr_t myvga_attributes[] = {
      .
      .
      .
  {"MYVGA_Developer_Debug", CFG_ATTR_INTTYPE,
      CFG_OP_CONFIGURE | CFG_OP_QUERY | CFG_OP_RECONFIGURE,
      (caddr_t)&myvga_developer_debug, 0, 1, 0},
⋮
};
```

  The `CFG_OP_RECONFIGURE` operation code lets you change the value of the variable with the following command;

```
> sysconfig -r myvga debug=1
```

  The `CFG_OP_QUERY` operation code lets you look at the current value of the variable with the following command:

```
> sysconfig -Q myvga
myvga:
debug -   1
```

Within the driver code, the value of the variable determines whether the driver displays debugging messages. For example:

```
if (myvga_developer_debug) {
    DPRINTF("myvgaattach(0x%lx): entry\n", (u_long)ctlr);
}
```

- Run the driver with the kernel debugger.

  If none of the preceding debugging techniques helps you track down the cause of the problem, you can run the kernel debugger as described in *Kernel Debugging*. The kernel debugger lets you set breakpoints, step through the driver code, and examine the values of variables.

When the driver runs successfully in console mode, you build the DDX and test the driver and the DDX with the X Window System.

5.3 Building the DDX Library

The following tools should be installed on your system before you can build a DDX library. You invoke some of these tools directly. Others are called by the `Makefile` file, which creates the library for you.

- `imake` (`/usr/bin/X11/imake`)

  You use the `imake` command extensively within the X source tree to create `Makefile` files for the entire X Window System as well as for the individual shared libraries that go into the system. The build procedure issues an `imake` command at the beginning of a full build. This guarantees the `imake` command will run on the build machine. DIGITAL UNIX provides the appropriate `imake` rules for creating DDX `Makefile` files. Reference pages are available on line for more information about the `imake` command.

- `make` (`/usr/bin/make`)

  You use the `make` command to compile and link individual DDXs or the entire X Window System. The `make` command updates the target based on whether the target's dependencies have been modified since the last build of the target or if the target itself does not exist. For example, the following command creates a `Makefile` file based on the rules defined in the `Imakefile` file in the current directory:

  ```
  % make Makefile
  ```

  The following command removes all the object files previously built using the `Makefile` file in the current directory:

```
% make clean
```

The following command creates the `include` files defined in the current `Makefile` file:

```
% make includes
```

The following command performs the dependency list commands specified in the current `Makefile` file:

```
% make depend
```

- C compiler (`/usr/bin/cc`)

  The `Makefile` file that you run to build the DDX uses the C compiler extensively for compiling source files. Reference pages are available on line for more information about the C compiler options.

- Link editor (`/usr/bin/ld`)

  The link editor links extended `coff` object files. The `ld` command combines several object files into one, performs relocation, resolves external symbols, builds tables and relocation information (for run-time linkage for shared links), and supports symbol table information for symbolic debugging. By default, the link editor uses the `–non_shared` option, which produces static executables, and the output object created does not use any shared objects during execution. The `–shared` option produces a shared object, including all of the tables for run-time linking and for resolving references to other specified shared objects. The object created may be used by the linker to make dynamic executables. For more information about the link editor, refer to the online reference pages.

- `yacc` (`/usr/bin/yacc`)

  The `yacc` command converts a context-free grammar specification into a set of tables for a simple automaton that executes an LR(1) parsing algorithm. The build procedure uses `yacc` to create the grammar rules for the `Xserver.conf` file.

- `lex` (`/usr/bin/lex`)

  The `lex` command uses the rules and actions contained in the grammar file to generate a program, `lex.yy.c`, which can be compiled with the `cc` command. That program can then receive input, break the input into the logical pieces defined by the grammar rules, and run program fragments contained in the actions in file `/usr/bin/ar`. The `ar` utility creates and maintains groups of files combined into a single archive file. Generally, you use this utility to create and update library files that the link editor uses. The build procedure uses `lex` to create the program to parse the `Xserver.conf` file.

The following sections describe several build procedures — for the entire X Window System, for the static server, and for the loadable server.

5.3.1 Building the X Window System

You need to build the entire X Window System when you first install the XDK. You need to build it again any time you upgrade the DIGITAL UNIX operating system or install or patch any of the build tools.

The build procedure described in this section produces two versions of the DIGITAL UNIX X Server:

- `Xserver/Xdec`

  This is the static server. It is a single, monolithic server that contains all DDXs and extensions that are present on the system. This version of the server is built from static libraries.

- `Xserver/loadable/Xdec`

  This is the loadable server, which is substantially smaller than `Xserver/Xdec`. This version of the server is built from shared libraries. Only the DDXs specified in the `Xserver.conf` file are loaded into the server at startup; deferred extensions are not loaded until the user needs them.

The Open Group `README` file is located in the `/usr/X11R6/xc/config/cf` directory and describes the template files and the default variables that the X Window System uses. There are four types of files in the `/usr/X11R6/xc/config/cf` directory:

- `cf` files contain operating system and platform-specific definitions, such as the operating system name. The DIGITAL UNIX `cf` file is called `osf1.cf`.
- `tmpl` files define operating system and platform-specific system build parameters. The DIGITAL UNIX `tmpl` file is called `Dec.tmpl`.
- `rules` files define operating system and platform-specific system build rules. The DIGITAL UNIX `rules` file is called `Dec.rules`.
- `def` files define host-specific customizations. The `def` file shipped with the XDK is called `site.def`.

The source tree build uses the rules and definitions defined in these files. All `Imakefile` files refer to the directories in the source tree by using the `TOPDIR` and `CURDIR` environment variables, so that they can refer to directories by their relative pathnames. Therefore, if you move the source tree, you do not have to change the `Imakefile` files.

To build the X Window System:

1. Install the XDK source kit as described in the cover letter.
2. Change the site definition file to run within your build environment. You may copy the `site.def.installed` file, supplied in the `config` directory, to `site.def`.

```
% cd /usr/X11R6/xc/config
% cp site.def.installed site.def
```

3. Build the source tree with the following command:

```
% cd /usr/X11R6/xc
% make "-k -f Makefile.ini BOOTSTRAPCFLAGS=-D__alpha" World
```

Because there are no `Imakefile` files in the `xc` and `contrib` directories, you must build both of these directories. You can use the following build script to build all of the source trees for the X Window System. Running this script replaces steps 2 and 3.

```
#!/bin/sh
# Script name: bldXIKit
# Use this script to build the X Implementors Kit

CURRENT_DIR=$1
FLAGS=" -k -f Makefile.ini BOOTSTRAPCFLAGS=-D__alpha"
if [ ! -f $CURRENT_DIR/xc/config/cf/site.def.orig ]
then
    mv $CURRENT_DIR/xc/config/cf/site.def \
        $CURRENT_DIR/xc/config/cf/site.def.orig
    cp $CURRENT_DIR/xc/config/cf/site.def.installed \
        $CURRENT_DIR/xc/config/cf/site.def
fi

for i in xc contrib ; \
do \
    (cd $CURRENT_DIR/$i ; \
    echo "making World in $CURRENT_DIR/$i..."; \
    make $FLAGS CDEBUGFLAGS= World); \
done

echo " "
echo "==== Make World of X build complete ======"
echo " "
```

To run this script, specify the name of the target directory of the source kit and the name of a log file. For example, if the source kit were installed in the `/usr/X11R6` directory and you were running the script on April 18, 1997, you would enter the following command:

```
% ./bldXIKit /usr/X11R6 > & ./build.log.04-18-97
```

This build procedure can take several hours to complete.

5.3.2 Building the DDX into the Static Server

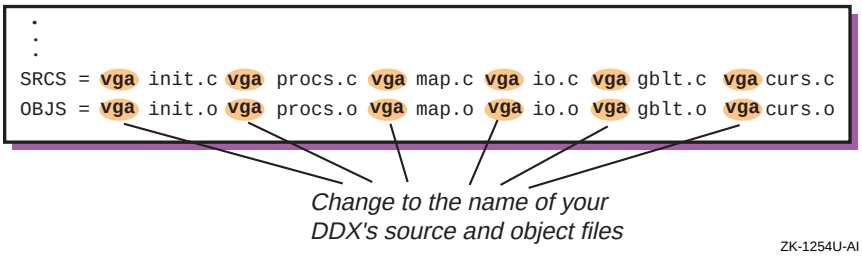
To build the DDX into the static server:

1. Create an `Imakefile` file, which defines the rules for the build procedure. The simplest way to create an `Imakefile` file is to copy one and modify it to suit your DDX. For example, to create an `Imakefile` file for a DDX that is based on the VGA DDX, you would copy the `Imakefile` file for the VGA DDX, as follows:

```
% cd Xserver/hw/dec/NEW
% cp ../vga/Imakefile .
```

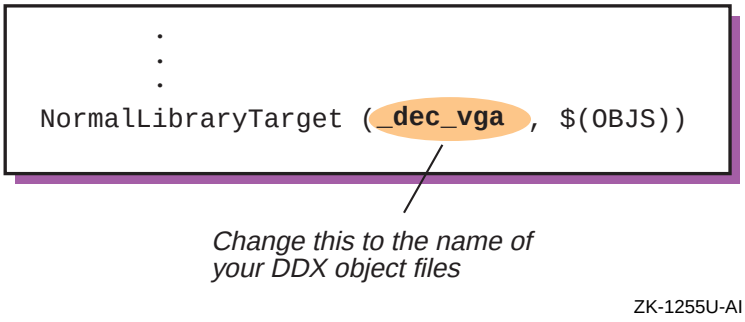
2. Modify the SRCS and OBJS definitions to point to your DDX's source and object files.

```
% vi Imakefile
```



ZK-1254U-AI

3. Modify the file's `NormalLibraryTarget` rule to refer to your DDX's object files.



ZK-1255U-AI

4. Create the `Makefile` file for the DDX.

```
% make Makefile
```

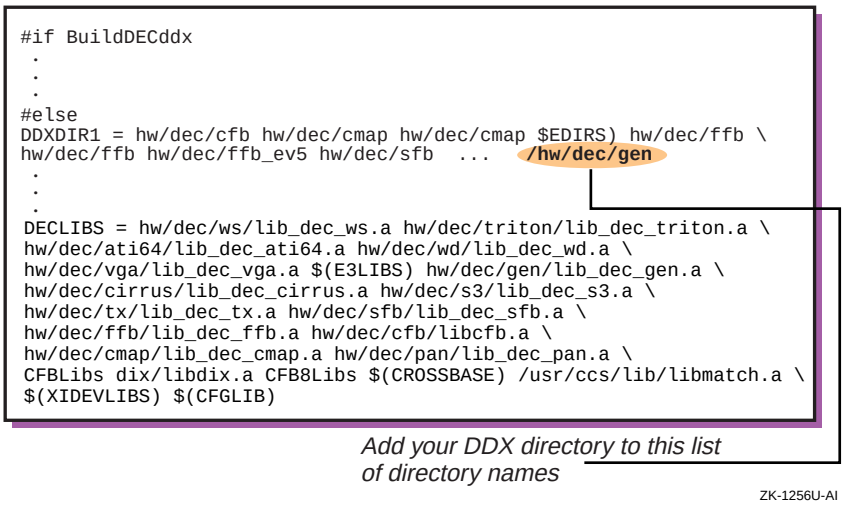
5. Build the static DDX library.

```
% make
```

This `make` command creates an object file with the `.o` extension.

6. Move up to the top-level directory and add your DDX directory name to the `Imakefile` file for the static server.

```
% cd /usr/X11R6/xc
% cd ../../..
% vi Imakefile
```



Note that `BuildDECddx` is defined as `NO` in the `site.def` file, so be sure to add your directory name to the `#else` portion of the `#if` statement.

7. Create the `Makefile` file for the static server.

```
% make Makefile
```

8. Compile and link the static server.

```
% make loadXdec
```

5.3.3 Building the DDX into the Loadable Server

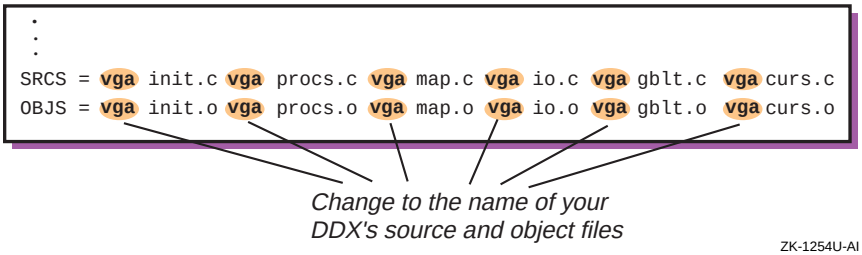
To build the DDX into the loadable server:

1. Create an `Imakefile` file, which defines the rules for the build procedure. The simplest way to create an `Imakefile` file for your DDX is to copy an existing `Imakefile` file and modify it to suit your DDX. For example, to create an `Imakefile` file for a DDX that is based on the VGA DDX, you would copy the `Imakefile` file for the VGA DDX, as follows:

```
% cd Xserver/hw/dec/NEW
% cp ../vga/Imakefile .
```

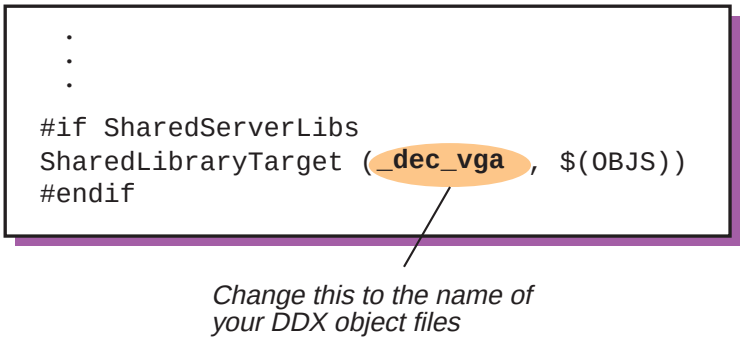
2. Modify the SRCS and OBJS definitions to point to your DDX's source and object files.

```
% vi Imakefile
```



ZK-1254U-AI

3. Modify the file's SharedLibraryTarget rule to refer to your DDX's object files.



ZK-1257U-AI

Note that SharedServerLibs is defined as YES in /usr/X11R6/xc/config/cf/Project.tmpl.

4. Create the make file for the DDX.

```
% make Makefile
```

5. Build the shared library.

```
% make
```

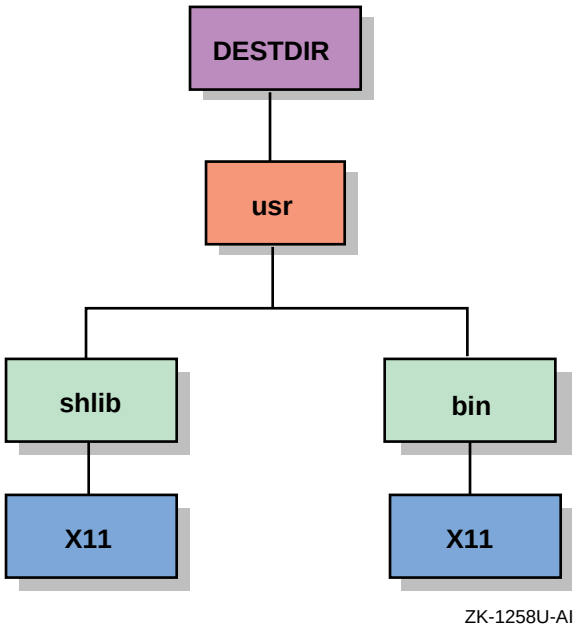
This make command creates a shared library file with the .so extension.

5.4 Testing and Debugging the DDX

You can run the static server from the build directory. You cannot run the loadable server directly from its build directory because the library search path is not set to the correct directory path. Therefore, after you have successfully built the DDX, you need to install the DDX in an output area and set the library path to search that area, as described in the sections that follow.

Figure 5–1 shows the directory structure of the install output area after you have created the output area.

Figure 5–1: Install Output Area



The loadable server is located in `/usr/X11R6/install/usr/bin/X11/Xdec`, and the link `/usr/X11R6/install/usr/bin/X11/X` points to it.

5.4.1 Running the Static Server

To run the static server, execute the static server image.

```
% /usr/X11R6/xc/programs/Xserver/Xdec
```

5.4.2 Running the Loadable Server

To run the loadable server:

1. Create an output area for your version of the loadable server. For example:

   ```
   % mkdir /usr/X11R6/install
   ```

2. Run the `Makefile` files in the `contrib` and `xc` directories.

   ```
   % cd contrib
   % make -k install DESTDIR=/usr/X11R6/install
   % cd xc
   % make -k install DESTDIR=/usr/X11R6/install
   ```

   The `Makefile` files derive the name of the output area based on the value of the `DESTDIR` variable. If you do not define `DESTDIR`, they install the DDX in subdirectories of the root directory (`/`).

   Warning

   If you do not define `DESTDIR`, the `Makefile` files overwrite the installed DIGITAL UNIX X Window System binary distribution.

3. Edit the `Xserver.conf` file and modify the `LD_LIBRARY` path to search the install output area rather than the default library path, `/usr/shlib/X11:/usr/shlib`. For example, to set the search path to the install output area `/usr/X11R6/install`, add the following entry to `Xserver.conf`:

   ```
   % vi Xserver.conf
   ```

   ```
   library_path <
        /usr/X11R6/install/usr/shlib/X11:/usr/shlib
   >
   ```

   ZK-1259U-AI

4. Run the loadable server.

   ```
   % /usr/X11R6/install/usr/bin/X11/Xdec \
   -config /usr/X11R6/install/usr/var/X11/Xserver.conf
   ```

5.4.3 Testing the DDX and Driver

You should run several tests to determine whether the DDX and the driver run correctly.

- Look for the basketweave pattern on the screen — a sign that the X Window System is running.
- Bring up an Xterm window.
- Run the Window Manager application.
- Run a simple client application, such as `ico`.
- Run programs that change the cursor and colormap, such as the Session Manager, to make sure the cursor and colors display properly.
- Test key transition points, such as starting and stopping the server and pressing the halt button.
- Run the `xset` utility to set various screen parameters, as follows:
  - Use `xset –p` to set pixel color values.
  - Use `xset –s` to set screen saver parameters.
  - Use `xset +dpms` to turn DPMS on, and `xset -dpms` to turn DPMS off.
- Run the `x11perf` benchmark program.

  ```
  % xc/programs/x11perf -v1.3 > datafile
  ```

  This program takes 4.5 hours to complete. You then run the `Xmark` utility against `datafile` to analyze the results.

  ```
  % Xmark datafile
  ```

  The numbers that `Xmark` returns depends on your graphics card. Following are some typical numbers:

Cirrus	1
QVision	2
S3Trio64	5
ATI64	7
TGA	15

When the DDX and driver pass these initial tests, you should run the test suite that the XDK supplies. The tests in the suite exercise the entire X Window System. When the server you have built passes these tests, with the exception of any DrawArc failures, you can be assured that your hardware support product is fully integrated into the system.

To build and run the test suite:

1. Set up the test suite environment by editing an environment file, such as `.cshrc`, as follows:

   ```
   % vi $HOME/.cshrc
   ```

   ```
   setenv TET_ROOT /usr/X11R6/src/xc/test/xsuite
   set path=($TET_ROOT/bin $TET_ROOT/xtest/bin $path)
   >
   ```

   ZK-1260U-AI

2. Create a link from the test suite directory to `/vsw`.

   ```
   % cd /usr/X11R6/src/xc/test/xsuite
   % ln -s /usr/X11R6/src/xc/test/xsuite /vsw
   ```

3. Modify the file `/usr/X11R6/src/xc/test/xsuite/xtest/tetbuild.cfg` to specify how to build the test suite on your system.

   ```
   % cd xtest
   % vi tetbuild.cfg
   ```

   ```
   TET_ROOT=/vsw
   SYSINC=-I/usr/include/X11 -I/usr/include
   LDFLAGS=
   DEPHEADERS=/usr/include/X11/Xlib.h
   XTESTFONTDIR=${TET_ROOT} /xtest
   XP_DEFINES=-DUNIXCONN -DDNETCONN
   ```

   ZK-1261U-AI

4. Run `xdpyinfo` and note the values returned for the number of screens, depths, dimensions (in millimeters), vendor release number, and motion buffer size.

5. Modify the file `/usr/X11R6/src/xc/test/xsuite/xtest/tetexec.cfg` to define the environment on which the tests will run. The following example shows the information returned by `xdpyinfo` in bold letters. Information that you must provide is shown in bold italic letters.

   ```
   % vi tetexec.cfg
   ```

```
TET_EXEC_IN_PLACE=True
XT_ALT_SCREEN=1
XT_FONTPATH=/vsw/xtest/font,/usr/lib/X11/fonts/misc\,/usr/lib/X11
XXT_VISUAL_CLASSES=StaticGray(8) GrayScale(8) StaticColor(8)
                   PseudoColor(8) TrueColor(8) DirectColor(8)
XT_FONTPATH_GOOD=/usr/lib/X11/fonts/100dpi,/usr/lib/X11/fonts/75dpi
XT_SCREEN_COUNT= 2
XT_PIXMAP_DEPTHS=1 8 12 24
XT_HEIGHT_MM= 254
XT_WIDTH_MM= 325
XT_SERVER_VENDOR=DECWINDOWS Digital Equipment Corporation Digital UNIX V4.0
XT_VENDOR_RELEASE= 1
XT_DISPLAYHOST= mynode.domain.com
XT_DECNET=No
XT_LOCAL=Yes
XT_FONTDIR=/vsw/xtest/fonts/
XT_DISPLAYMOTIONBUFFERSIZE= 100
```

ZK-1262U-AI

6. Build the test suite.

```
% cd fonts% make comp_pcf% cd ..% make% cd xtest% tcc -
b% sh build_Mtests
```

7. Run the server and the test suite.

```
% /usr/X11R6/install/usr/bin/X11/Xdec \
-config /usr/X11R6/install/usr/var/X11/Xserver.conf \
-ac -pn
% tcc -e -xtest all &
```

The test suite exercises stippling and drawing code and lets you know if the DDX does not read and write pixmaps correctly. The full test suite can take 1.5 hours to complete. It notifies you of the name and location of the journal file that it creates to log its progress. When the test suite is completed, you should examine the journal file or run the `rpt` command to produce a summary of the test results. To run `rpt`, supply the name of the journal file. For example:

```
% rpt results/0003/journal
```

The following is a typical journal file. You should expect four failures each in the `drwarc` and `drwarcs` test areas.

VSW Area Name	TOTALS	Pass		Unres		UnSupp		NIU
	Asserts		Fail		UnIni		UnTest	
/test/CH06/drwarc	102	70	4	0	0	2	1	25
/test/CH06/drwarcs	112	79	4	0	0	2	2	25
CH06 Totals	1522	1200	8	0	0	46	67	201
TOTALS	3924	3130	8	0	0	123	398	265

5.4.4 Debugging the DDX

If you have problems with the DDX, you can run the server with the dbx debugger. The debugger lets you set breakpoints, step through the program code, and examine the contents of variables to pinpoint the cause of the problem. However, the default compiler options for building both the static and loadable servers strip information from the images that is useful to the debugger. Therefore, you must recompile the source files that you want to debug before you run the debugger.

Move to the directory that contains the DDX you want to debug, and recompile the library. For example:

```
% cd /usr/X11R6/xc/programs/Xserver/hw/dec/gen
% make clean
% make CDEBUGFLAGS=-g LDOPTIONS=-warning_unresolved
```

5.4.4.1 Debugging the Static Server

To debug the static server:

1. Create a /.dbxinit file that lists the source directories for the debugger to place in its search path, sets a breakpoint at the server main loop, and runs the server.

   ```
   % vi /.dbxinit
   ```

   ```
   use /usr/X11R6/xc/programs/Xserver/dix
   /usr/X11R6/xc/programs/mi
   /usr/X11R6/xc/programs/Xext
   /usr/X11R6/xc/programs/Xserver/hw/dec/gen
   /usr/X11R6/xc/programs/Xserver/hw/dec/vga

   Stop in main
   r -pn
   ```

   ZK-1263U-AI

   The –pn option forces the server to continue even if it cannot connect to all of its well-known sockets.

2. Start the debugging session for the static server.

   ```
   % /usr/bin/dbx /usr/X11R6/src/xc/programs/Xserver/Xdec
   ```

5.4.4.2 Debugging the Loadable Server

The loadable server loads a DDX when it finds a device that matches the DDX. You cannot set a breakpoint for a DDX function until its shared library is loaded and its symbols are resolved.

If the server crashes during startup, it should leave behind a `core` file, either in the current working directory, in the root directory (`/`), or in `/usr/X11R6/install/usr/shlib/X11`. Run `dbx` on the `core` file to determine where the segmentation fault occurred.

```
% /usr/bin/dbx /usr/X11R6/install/usr/bin/X11/X /core
```

Use `dbx` commands to get a traceback of the routines that caused the crash. If you cannot get enough information from the traceback or register dump, you need to debug the DDX code. For example, if the failure is in the `genScreenInit_base` routine, you would create the following `./dbxinit` file:

```
% vi ./dbxinit
```

```
use /usr/X11R6/xc/programs/Xserver/dix
/usr/X11R6/xc/programs/mi
/usr/X11R6/xc/programs/Xext
/usr/X11R6/xc/programs/Xserver/hw/dec/gen
/usr/X11R6/xc/programs/Xserver/hw/dec/vga

Stop in main
r -config /usr/X11R6/install/usr/var/X11/Xserver.conf
stop in CallDixMain
c
stop in LoadAndInitDeviceModule
c
stop in AddScreen
c
```

ZK-1264U-AI

This `dbxinit` file sets a breakpoint at `LoadAndInitDeviceModule` because that routine loads all DDXs and finds their DDX initialization routines. It sets a breakpoint at the `AddScreen` routine because that routine invokes the DDX initialization routines. After the DDX that you want to debug has been loaded, you can set breakpoints at any of its entry points.

5.5 Packaging the Kit

When you are satisfied with the quality and performance of your display device driver and DDX, you can create a kit on tape, CD–ROM, or diskette, which you can then distribute to customers. The kit consists of two layered products — a kernel product for the display driver and a user product for the DDX. Table 5–1 summarizes the contents of the distribution kit.

Table 5–1: Graphics Device Driver and DDX Kit

Device Driver	DDX
Driver `.mod` file	DDX `.so` file
`sysconfigtab` file fragment	
Driver subset control program	DDX subset control program

The *Guide to Preparing Product Kits* describes how to create subsets and produce distribution media for all layered products, including kits for device drivers and foreign devices.

A

Graphics Device Data Structures

The reference pages in this appendix describe the data structures that a graphics display device driver and a DDX use. Each reference page for a data structure may include the following information:

Name

This section lists the name of the structure along with a description of its purpose.

Include File

This section shows the header file, including the path where the structure is defined.

Synopsis

This section shows one of the following:

- C structure definition

  This is shown when you need to understand or initialize all of the members of the structure.

- Structure member table

  This is shown when you need to understand or reference some of the members of the structure.

Members

This section provides a description of each member of the structure.

Description

This section provides more details about the purpose of the data structure.

Related Information

This section lists related reference pages where you can find additional information.

DepthRec

NAME

DepthRec – Used by the DDX to describe one depth that a screen supports

INCLUDE FILE

/usr/X11R6/xc/programs/Xserver/include/scrnintstr.h

SYNOPSIS

```
typedef struct _Depth {
    unsigned char       depth;
    short               numVids;
    VisualID            *vids;
} DepthRec;
```

MEMBERS

depth
: Specifies the number of bits/pixel for the depth.

numVids
: Specifies the number of visuals that the depth supports.

vids
: Specifies a pointer to an array of client IDs for each visual.

DESCRIPTION

The DepthRec structure defines one depth that a screen supports. The DDX defines an array of DepthRec structures — one for each supported depth.

RELATED INFORMATION

Appendix A, Data Structures: ScreenRec, VisualRec

NAME

LS_LibraryReq – May be referenced by the DDX to determine the default and configured libraries

INCLUDE FILE

/usr/X11R6/xc/x11/Xserver/loadable.h

SYNOPSIS

Member Name	Data Type
LibName	char *
LibFileName	char *
ProcName	char *
DeviceName	char *
SubLibs	struct LS_LibraryReq
NumSubLibs	int
_OpenLibIndex	int

MEMBERS

LibName
Specifies the symbolic name of the library.

LibFileName
Specifies the library file name.

ProcName
Specifies the name of the library's initialization procedure.

DeviceName
Specifies the device module ID.

SubLibs
Specifies the sublibraries of this library.

NumSubLibs
Specifies the number of sublibraries in SubLibs.

OpenLibIndex
Specifies a quick lookup index to this library in a list of open libraries.

DESCRIPTION

The LS_LibraryReq structure is an opaque data structure that contains information about the libraries that are loaded, or can be loaded, into the server.

RELATED INFORMATION

Appendix D, Loadable Services Routines: LS_GetDeviceName, LS_GetInitProc, LS_GetInitProcName, LS_GetLibFileName, LS_GetLibName, LS_GetLibraryReqByDeviceName, LS_GetLibraryReqByExtension, LS_GetLibraryReqByLibName, LS_GetSubLibList, LS_IsLibraryInited, LS_LoadLibraryReqs, LS_MarkForUnloadLibraryReqs, LS_UnLoadLibraryReqs

NAME

ScreenRec – Used by the DDX to describe the graphics display characteristics

INCLUDE FILE

/usr/X11R6/xc/programs/Xserver/include/scrnintstr.h

SYNOPSIS

```
typedef struct _Screen {
    int                 myNum;
    ATOM                id;
    short               width, height;
    short               mmWidth, mmHeight;
    short               numDepths;
    unsigned char       rootDepth;
    DepthPtr            allowedDepths;
    unsigned long       rootVisual;
    unsigned long       defColormap;
    short               minInstalledCmaps, maxInstalledCmaps;
    char                backingStoreSupport, saveUnderSupport;
    unsigned long       whitePixel, blackPixel;
    unsigned long       rgf;
    GCPtr               GCperDepth[MAXFORMATS+1];
    PixmapPtr           PixmapPerDepth[1];
    pointer             devPrivate;
    short               numVisuals;
    VisualPtr           visuals;
    int                 WindowPrivateLen;
    unsigned            *WindowPrivateSizes;
    unsigned            totalWindowSize;
    int                 GCPrivateLen;
    unsigned            *GCPrivateSizes;
    unsigned            totalGCSize;

    /* Random screen procedures */

    CloseScreenProcPtr          CloseScreen;
    QueryBestSizeProcPtr        QueryBestSize;
    SaveScreenProcPtr           SaveScreen;
    GetImageProcPtr             GetImage;
    GetSpansProcPtr             GetSpans;
```

ScreenRec

```
PointerNonInterestBoxProcPtr PointerNonInterestBox;
SourceValidateProcPtr        SourceValidate;

/* Window Procedures */

CreateWindowProcPtr           CreateWindow;
DestroyWindowProcPtr          DestroyWindow;
PositionWindowProcPtr         PositionWindow;
ChangeWindowAttributesProcPtr ChangeWindowAttributes;
RealizeWindowProcPtr          RealizeWindow;
UnrealizeWindowProcPtr        UnrealizeWindow;
ValidateTreeProcPtr           ValidateTree;
PostValidateTreeProcPtr       PostValidateTree;
WindowExposuresProcPtr        WindowExposures;
PaintWindowBackgroundProcPtr  PaintWindowBackground;
PaintWindowBorderProcPtr      PaintWindowBorder;
CopyWindowProcPtr             CopyWindow;
ClearToBackgroundProcPtr      ClearToBackground;
ClipNotifyProcPtr             ClipNotify;

/* Pixmap procedures */

CreatePixmapProcPtr          CreatePixmap;
DestroyPixmapProcPtr         DestroyPixmap;

/* Backing store procedures */

SaveDoomedAreasProcPtr       SaveDoomedAreas;
RestoreAreasProcPtr          RestoreAreas;
ExposeCopyProcPtr            ExposeCopy;
TranslateBackingStoreProcPtr TranslateBackingStore;
ClearBackingStoreProcPtr     ClearBackingStore;
DrawGuaranteeProcPtr         DrawGuarantee;

/* Font procedures */

RealizeFontProcPtr          RealizeFont;
UnrealizeFontProcPtr        UnrealizeFont;

/* Cursor Procedures */

ConstrainCursorProcPtr      ConstrainCursor;
CursorLimitsProcPtr         CursorLimits;
DisplayCursorProcPtr        DisplayCursor;
RealizeCursorProcPtr        RealizeCursor;
```

```
UnrealizeCursorProcPtr      UnrealizeCursor;
RecolorCursorProcPtr        RecolorCursor;
SetCursorPositionProcPtr    SetCursorPosition;

/* GC procedures */

CreateGCProcPtr              CreateGC;

/* Colormap procedures */

CreateColormapProcPtr         CreateColormap;
DestroyColormapProcPtr        DestroyColormap;
InstallColormapProcPtr        InstallColormap;
UninstallColormapProcPtr      UninstallColormap;
ListInstalledColormapsProcPtr ListInstalledColormaps;
StoreColorsProcPtr            StoreColors;
ResolveColorProcPtr           ResolveColor;

/* Region procedures */

RegionCreateProcPtr         RegionCreate;
RegionInitProcPtr           RegionInit;
RegionCopyProcPtr           RegionCopy;
RegionDestroyProcPtr        RegionDestroy;
RegionUninitProcPtr         RegionUninit;
IntersectProcPtr            Intersect;
UnionProcPtr                Union;
SubtractProcPtr             Subtract;
InverseProcPtr              Inverse;
RegionResetProcPtr          RegionReset;
TranslateRegionProcPtr      TranslateRegion;
RectInProcPtr               RectIn;
PointInRegionProcPtr        PointInRegion;
RegionNotEmptyProcPtr       RegionNotEmpty;
RegionEmptyProcPtr          RegionEmpty;
RegionExtentsProcPtr        RegionExtents;
RegionAppendProcPtr         RegionAppend;
RegionValidateProcPtr       RegionValidate;
BitmapToRegionProcPtr       BitmapToRegion;
RectsToRegionProcPtr        RectsToRegion;
SendGraphicsExposeProcPtr   SendGraphicsExpose;

/* Operating System Procedures */

ScreenBlockHandlerProcPtr   BlockHandler;
```

```
    ScreenWakeupHandlerProcPtr  WakeupHandler;

    pointer                     blockData;
    pointer                     wakeupData;
    DevUnion                    *devPrivates;
    CreateScreenResourcesProcPtr CreateScreenResources;
    ModifyPixmapHeaderProcPtr   ModifyPixmapHeader;
    PixmapPtr                   pScratchPixmap;

    MarkWindowProcPtr           MarkWindow;
    MarkOverlappedWindowsProcPtr MarkOverlappedWindows;
    ChangeSaveUnderProcPtr      ChangeSaveUnder;
    PostChangeSaveUnderProcPtr  PostChangeSaveUnder;
    MoveWindowProcPtr           MoveWindow;
    ResizeWindowProcPtr         ResizeWindow;
    GetLayerWindowProcPtr       GetLayerWindow;
    HandleExposuresProcPtr      HandleExposures;
    ReparentWindowProcPtr       ReparentWindow;
    ChangeBorderWidthProcPtr    ChangeBorderWidth;
    MarkUnrealizedWindowProcPtr MarkUnrealizedWindow;
} ScreenRec;
```

MEMBERS

`myNum`
: Specifies an index into an array of screens to this `ScreenRec` structure.

`id`
: Specifies the screen number — a unique number assigned to the screen by the server at startup.

`width, height`
: Specify the width and height of the screen, in pixels.

`mmWidth, mmHeight`
: Specify the width and height of the screen, in millimeters.

`numDepths`
: Specifies the number of depths that the screen supports.

`rootDepth`
: Specifies the depth of the root window.

ScreenRec

`allowedDepths`
Specifies the number of depths that the screen supports.

`rootVisual`
Specifies the visual class of the root window.

`defColormap`
Specifies the default colormap.

`minInstalledCmaps`, `maxInstalledCmaps`
Specify the minimum and maximum number of colormaps that the user can install on the screen.

`backingStoreSupport`, `saveUnderSupport`
Specify whether backing store and save under are supported.

`whitePixel`, `blackPixel`
Specify the pixel value of white and black.

`rgf`
Specifies an array of pointers.

`GCperDepth`
Specifies the number of graphics contexts that a depth supports.

`PixmapPerDepth`
Specifies the number of pixmaps that a depth supports.

`devPrivate`
Specifies a pointer to the device private areas.

`numVisuals`
Specifies the number of visuals that the screen supports.

`visuals`
Specifies a pointer to the visuals.

`WindowPrivateLen`
Specifies the length of a window private area.

ScreenRec

`WindowPrivateSizes`
Specifies a pointer to the window private areas.

`totalWindowSize`
Specifies the total size of all window private areas.

`GCPrivateLen`
Specifies the size of a graphics context private

`GCPrivateSizes`
Specifies a pointer to the graphics context private areas.

`totalGCSize`
Specifies the total size of all graphics context private areas.

Screen Procedures
Define the routines that handle screen operations. You need to define the following screen procedures:

Procedure	Description
`CloseScreen`	Closes the screen and frees resources that the screen uses.
	VGA default: None (The DDX must provide this routine)
`QueryBestSize`	Returns the largest possible size for a cursor, tile, or stipple.
	VGA default: `mfbQueryBestSize`
`SaveScreen`	Turns the screen saver on or off, or forces the screen saver to change modes.
	VGA default: `colorSaveScreen`
`GetImage`	Extracts a portion of a drawable image and stores it in a `XImage` structure.
	VGA default: `vgaGetImage`
`GetSpans`	Extracts a number of lines from the drawable image and stores them in a buffer.
	VGA default: `vgaGetSpans`

Procedure	Description
PointerNonInterestBox	Called by the DIX, indicates whether pointer events can be ignored. VGA default: colorPointerNonInterestBox
SourceValidate	Ensures that a drawable image is ready to be copied. VGA default: NULL pointer

Window Procedures

Define the routines that handle window operations. You need to define the following window procedures:

Procedure	Description
CreateWindow	Creates a new window. VGA default: vgaCreateWindow
DestroyWindow	Frees resources associated with a window that is about to be destroyed. VGA default: vgaDestroyWindow
PositionWindow	Updates data structures to a new window position prior to moving the window to that position. VGA default: vgaPositionWindow
ChangeWindowAttributes	Updates window attributes and any resources that depend on those attributes. VGA default: vgaChangeWindowAttributes
RealizeWindow	Allocates and initializes resources a window needs if it is going to be mapped to the screen. VGA default: vgaMapWindow
UnrealizeWindow	Deallocates the resources that RealizeWindow allocates when a window is unmapped. VGA default: vgaUnmapWindow

ScreenRec

Procedure	Description
ValidateTree	Computes the border clip region and window clip region for every marked window in a window tree when a portion of the tree changes.
	VGA default: miValidateTree
PostValidateTree	Performs device-dependent operations on a window tree that has been validated.
	VGA default: NULL pointer
WindowExposures	Paints the exposed areas of any window that has been validated.
	VGA default: miWindowExposures
PaintWindowBackground	Paints the exposed areas of a window background.
	VGA default: vgaPaintWindow
PaintWindowBorder	Paints the exposed areas of a window border.
	VGA default: vgaPaintWindow
CopyWindow	Copies an exposed area from a window to another place on the screen.
	VGA default: vgaCopyWindow
ClearToBackground	Draws the window background or generates exposure events for a portion of a window.
	VGA default: miClearToBackground
ClipNotify	Adjusts the window's hardware clipping resources when the clip list changes.
	VGA default: None

Pixmap Procedures

Define the routines that handle pixmap operations. You need to define the following pixmap procedures:

Procedure	Description
CreatePixmap	Allocates a PixmapRec structure for a new pixmap.
	VGA default: vgaCreatePixmap

Procedure	Description
DestroyPixmap	Deallocates the resources for a pixmap. VGA default: vgaDestroyPixmap

Backing Store Procedures

Define the routines that handle backing store operations. You need to define the following backing store procedures:

Procedure	Description
SaveDoomedAreas	Saves a portion of a window that is about to be obscured or moved and places it in a backing store pixmap. VGA default: Null pointer
RestoreAreas	Recovers an area of a window that was previously saved to a backing store pixmap. VGA default: Null pointer
ExposeCopy	Copies an area from a backing store pixmap, moves it to another place on the screen, and exposes any portion of the area that is no longer obscured. VGA default: Null pointer
TranslateBackingStore	Allocates a new backing store pixmap, if necessary, when a window is resized or moved. VGA default: Null pointer
ClearBackingStore	Clears an area of a backing store pixmap. VGA default: Null pointer
DrawGuarantee	Specifies whether any upcoming graphics events will be clipped to the window's border clip region. VGA default: Null pointer

Font Procedures

Define the routines that handle font operations. You need to define the following font procedures:

ScreenRec

Procedure	Description
RealizeFont	Initializes any font-specific data structures for a screen when a font is opened.
	VGA default: vgaRealizeFont
UnrealizeFont	Frees any resources that RealizeFont allocates.
	VGA default: vgaUnrealizeFont

Cursor Procedures

Define the routines that handle cursor operations. You need to define the following cursor procedures:

Procedure	Description
ConstrainCursor	Forces a cursor to stay within a given area.
	VGA default: None
CursorLimits	Returns the physical constraints a cursor would have if ConstrainCursor was called with a specified area.
	VGA default: None
DisplayCursor	Displays the cursor on the screen.
	VGA default: None
RealizeCursor	Allocates and initializes device-specific resources for a cursor before displaying the cursor on the screen.
	VGA default: None
UnrealizeCursor	Deallocates the resources for a cursor.
	VGA default: None
RecolorCursor	Changes the color of a cursor.
	VGA default: None
SetCursorPosition	Moves the cursor to the specified position on the screen.
	VGA default: None

Graphics Context Procedure

Defines the routine that handles graphics context operations. You need to define the following graphics context procedure:

Procedure	Description
`CreateGC`	Allocates resources needed for a new graphics context.
	VGA default: `vgaCreateGC`

Colormap Procedures

Define the routines that handle colormap operations. You need to define the following colormap procedures:

Procedure	Description
`CreateColormap`	Allocates resources for a new colormap.
	VGA default: `vgaCreateColormap`
`DestroyColormap`	Deallocates resources for a colormap that is to be destroyed.
	VGA default: `vgaDestroyColormap`
`InstallColormap`	Changes the mapping of pixel values to colors that match a new colormap.
	VGA default: None
`UninstallColormap`	Removes a colormap and replaces it with the default colormap for the screen.
	VGA default: None
`ListInstalledColormaps`	Returns the numbers and resource IDs of all colormaps installed on the screen.
	VGA default: None
`StoreColors`	Stores one or more colors in a colormap.
	VGA default: None
`ResolveColor`	Adjusts the server's universal color values to values that are appropriate for the screen.
	VGA default: `vgaResolveColor`

ScreenRec

Region Procedures

Define the routines that handle region operations. You need to define the following region procedures:

Procedure	Description
`RegionCreate`	Allocates and initializes a `RegionRec` structure for a new region.
	VGA default: `miRegionCreate`
`RegionInit`	Initializes the `RegionRec` structure for an existing region.
	VGA default: `miRegionInit`
`RegionCopy`	Copies the contents of one region to another.
	VGA default: `miRegionCopy`
`RegionDestroy`	Deallocates the resources for a region.
	VGA default: `miRegionDestroy`
`RegionUninit`	Deallocates resources for a statically allocated region but does not deallocate its `RegionRec` structure.
	VGA default: `miRegionUninit`
`Intersect`	Returns the intersection of two regions.
	VGA default: `miIntersect`
`Union`	Returns the union of two regions.
	VGA default: `miUnion`
`Subtract`	Returns the difference between two regions.
	VGA default: `miSubtract`
`Inverse`	Returns the inverse of a region.
	VGA default: `miInverse`
`RegionReset`	Changes the extents of a region.
	VGA default: `miRegionReset`
`TranslateRegion`	Moves a region to a new location on the screen.
	VGA default: `miTranslateRegion`

Procedure	Description
RectIn	Determines whether a rectangle falls within a region.
	VGA default: miRectIn
PointInRegion	Determines whether a point falls within a region.
	VGA default: miPointInRegion
RegionNotEmpty	Returns true if a specified region is not empty.
	VGA default: miRegionNotEmpty
RegionEmpty	Returns true if a specified region is empty.
	VGA default: miRegionEmpty
RegionExtents	Returns the smallest rectangle that encloses a specified region.
	VGA default: miRegionExtents
RegionAppend	Concatenates two regions.
	VGA default: miRegionAppend
RegionValidate	Creates a single region from several screen areas concatenated by multiple calls to RegionAppend.
	VGA default: miRegionValidate
BitmapToRegion	Converts a bitmap to a region.
	VGA default: mfbBitmapToRegion
RectsToRegion	Creates a region from several rectangles.
	VGA default: miRectsToRegion
SendGraphicsExpose	Sends a GraphicsExpose or NoExpose event to the client for a specified region.
	VGA default: miSendGraphicsExpose

Operating System Procedures

Define the routines that handle operating system operations. The MI library defines the following operating system procedures. A DDX does not usually need to define routines to perform these operations.

ScreenRec

Procedure	Description
BlockHandler	VGA default: NoopDDA
WakeupHandler	VGA default: NoopDDA
ModifyPixmapHeader	VGA default: miModifyPixmapHeader
CreateScreenResources	VGA default: miCreateScreenResources
MarkWindow	VGA default: miMarkWindow
MarkOverlappedWindows	VGA default: miMarkOverlappedWindows
ChangeSaveUnder	VGA default: miChangeSaveUnder
PostChangeSaveUnder	VGA default: miPostChangeSaveUnder
MoveWindow	VGA default: miMoveWindow
ResizeWindow	VGA default: miResizeWindoW
GetLayerWindow	VGA default: miGetLayerWindow
HandleExposures	VGA default: miHandleExposures
ReparentWindow	VGA default: NULL pointer
ChangeBorderWidth	VGA default: miChangeBorderWidth
SetShape	VGA default: miSetShape
MarkUnrealizedWindow	VGA default: miMarkUnrealizedWindow

blockData
: Specifies a pointer to an area that the block handler can use.

wakeupData
: Specifies a pointer to an area that the wakeup handler can use.

devPrivates
: Specifies a pointer to a DDX-specific data structure.

pScratchPixmap
: Specifes a pointer to an area that the DDX can use for pixmaps.

DESCRIPTION

The `ScreenRec` structure defines the characteristics of a screen. The DDX creates one `ScreenRec` structure for each screen detected when the server starts up.

RELATED INFORMATION

Appendix A, Data Structures: `DepthRec`, `VisualRec`

VisualRec

NAME

VisualRec – Used by the DDX to describe the characteristics of a visual class

INCLUDE FILE

/usr/X11R6/xc/programs/Xserver/include/scrnintstr.h

SYNOPSIS

```
typedef struct _Visual {
    VisualID            vid;
    short               class;
    short               bitsPerRGBValue;
    short               ColormapEntries;
    short               nplanes;
    unsigned long       redMask, greenMask, blueMask;
    int                 offsetRed, offsetGreen, offsetBlue;
} VisualRec;
```

MEMBERS

vid
: Specifies the client ID of the visual.

class
: Specifies the visual class, either GrayScale, StaticColor, PseudoColor, or TrueColor.

bitsPerRGBValue
: Specifies the number of significant bits in a color cell needed to store the red, green, and blue primary colors.

ColormapEntries
: Specifies the number of entries in a colormap for this visual class. The GrayScale, StaticColor, and PseudoColor visual classes use colormap entries.

nplanes
: Specifies the number of bits per pixel.

`redMask`, `greenMask`, `blueMask`
: Specifies the valid bits that can appear in the red, green, and blue fields of a pixel value. The `PseudoColor` visual class uses this structure member.

`offsetRed`, `offsetGreen`, `offsetBlue`
: Specifies the range within the 24-bit address that contains the color and intensity for a pixel. The `TrueColor` visual class uses this structure member.

DESCRIPTION

The `VisualRec` structure defines the characteristics of a single visual class. You define an array of these structures, one for each visual class that the display supports.

RELATED INFORMATION

Appendix A, Data Structures: `Depthrec`, `ScreenRec`

NAME

`ws_color_cell` – Used by the display driver to describe a single color entry in a colormap

INCLUDE FILE

`/usr/sys/include/sys/workstation.h`

SYNOPSIS

```
typedef struct {
    unsigned int index;
    unsigned short  red;
    unsigned short  green;
    unsigned short  blue;
    unsigned short  pad;
} ws_color_cell;
```

MEMBERS

`index`
Specifies the offset of this entry from the beginning of the colormap.

`red`
Specifies the intensity of red in this color entry.

`green`
Specifies the intensity of green in this color entry.

`blue`
Specifies the intensity of blue in this color entry.

`pad`
Forces structure to align on a word boundary.

DESCRIPTION

The colormap is made up of a number of `ws_color_cell` structures. Each entry in the colormap determines the amount of red, green, and blue in the color.

RELATED INFORMATION

Appendix C, Driver Routines: `load_color_map_entry`, `load_cursor`, `recolor_cursor`

NAME

`ws_color_map_functions` – Used by the display driver to specify the routines that perform colormap functions for a particular graphics display

INCLUDE FILE

`/usr/sys/include/sys/wsdevice.h`

SYNOPSIS

```
typedef struct {
    caddr_t (*init_colormap_handle)();
    int (*init_color_map)();
    int (*load_color_map_entry)();
    void(*clean_color_map)();
    int (*video_on)();
    int (*video_off)();
    caddr_t colormap_handle;
    int (*cmap_private)();
} ws_color_map_functions;
```

MEMBERS

`init_colormap_handle`
: Specifies a pointer to the driver routine that initializes the colormap handle.

`init_color_map`
: Specifies a pointer to the driver routine that sets up a colormap on the graphics board.

`load_color_map_entry`
: Specifies a pointer to the driver routine that initializes a single entry in a colormap.

`clean_color_map`
: Specifies a pointer to the driver routine that reinitializes a colormap that has become dirty.

`video_on`
: Specifies a pointer to the driver routine that turns the screen on.

`video_off`
Specifies a pointer to the driver routine that turns the screen off.

`colormap_handle`
Specifies the address of driver-specific colormap information.

`cmap_private`
Reserved for future use by DIGITAL.

DESCRIPTION

The `ws_colormap_functions` structure contains pointers to the routines that perform colormap operations for the display driver.

RELATED INFORMATION

Appendix A, Data Structures: `ws_screens`

Appendix C, Driver Routines: `clean_colormap`, `load_color_map_entry`, `recolor_cursor`, `set_cursor_position`, `video_on`, `ws_register_screen`

NAME

ws_cursor_data – Used by the display driver to describe the cursor to be displayed in a particular screen

INCLUDE FILE

/usr/sys/include/sys/workstation.h

SYNOPSIS

```
typedef struct {
    short screen;
    short width, height;
    short x_hot;
    short y_hot;
    unsigned int *cursor;
    unsigned int *mask;
} ws_cursor_data;
```

MEMBERS

screen
: Specifies the screen in which to display the cursor.

width, height
: Specifies the width and height of the cursor in pixels.

x_hot, y_hot
: Specify the coordinates of the cursor. The display device driver usually maintains this information.

cursor
: Specifies a pointer to the cursor area.

mask
: Specifies a pointer to the cursor mask.

DESCRIPTION

The ws_cursor_data structure contains the information the display driver needs to draw and move the graphics cursor.

RELATED INFORMATION

Appendix C, Driver Routines: `load_color_map_entry`, `load_cursor`

NAME

`ws_cursor_functions` – Used by the display driver to specify the routines that perform cursor functions for a particular graphics display

INCLUDE FILE

`/usr/sys/include/sys/wsdevice.h`

SYNOPSIS

```
typedef struct {
    caddr_t (*init_cursor_handle)();
    int (*load_cursor)();
    int (*recolor_cursor)();
    int (*set_cursor_position)();
    int (*cursor_on_off)();
    caddr_t cursor_handle;
    int (*cursor_private)();
} ws_cursor_functions;
```

MEMBERS

`init_cursor_handle`
: Specifies a pointer to the routine to initialize the cursor handle.

`load_cursor`
: Specifies a pointer to the routine that loads the cursor map.

`recolor_cursor`
: Specifies a pointer to the driver routine that changes the background and foreground colors of the cursor.

`set_cursor_position`
: Specifies a pointer to the driver routine that positions the cursor on the screen.

`cursor_on_off`
: Specifies a pointer to the driver routine that turns the cursor on and off.

cursor_handle
: Specifies the address of a driver-specific data structure that contains cursor information.

cursor_private
: Reserved for future use by DIGITAL.

DESCRIPTION

The ws_cursor_functions structure contains pointers to the routines that perform cursor operations for the display driver.

RELATED INFORMATION

Appendix A, Data Structures: ws_cursor_data, ws_screens

Appendix C, Driver Routines: cursor_on_off, load_cursor, ws_register_screen

NAME

`ws_depth_descriptor` – Used by the display driver to describe the characteristics of the depths that the graphics board supports

INCLUDE FILE

`/usr/sys/include/sys/workstation.h`

SYNOPSIS

```
typedef struct {
    short screen;
    short which_depth;
    short fb_width;
    short fb_height;
    short depth;
    short bits_per_pixel;
    short scanline_pad;
    caddr_t physaddr;
    caddr_t pixmap;
    caddr_t plane_mask_phys;
    caddr_t plane_mask;
} ws_depth_descriptor;
```

MEMBERS

`screen`
Specifies the ID of the screen associated with this depth descriptor.

`which_depth`
Specifies the ID of this depth descriptor.

`fb_width`
Specifies the width of the frame buffer in pixels.

`fb_height`
Specifies the height of the frame buffer in pixels.

`depth`
Specifies the current depth.

`bits_per_pixel`
Specifies the number of bits per pixels for this depth.

`scanline_pad`
Not used.

`physaddr`
Specifies the physical address of the depth. The display driver uses this address.

`pixmap`
Specifies the user-space address of the frame buffer. The DDX uses this address.

`plane_mask_phys`
Specifies the physical address of the plane mask, if any. The display driver uses this address.

`plane_mask`
Specifies the user-space address of the plane mask. The DDX typically uses this user-mapped address to access the graphics board registers.

DESCRIPTION

The `ws_depth_descriptor` structure defines the number of bits per pixel of a window or pixmap. This determines the number of colors that the screen can display. The display driver also stores the user-mapped address for the frame buffer and registers in this data structure.

RELATED INFORMATION

Appendix C, Driver Routines: `ws_map_region`, `ws_register_screen`, `ws_unmap_screen`

NAME

ws_map_control – Used by the display driver to control mapping at a particular depth

INCLUDE FILE

/usr/sys/include/sys/workstation.h

SYNOPSIS

```
typedef struct {
    short screen;
    short which_depth;
    short map_unmap;
} ws_map_control;
```

MEMBERS

screen
: Specifies the screen to be mapped.

which_depth
: Specifies an index into an array of ws_depth_descriptor structures, which point to the depth to be supported.

map_unmap
: Contains either MAP_SCREEN or UNMAP_SCREEN to indicate the current state of the screen.

DESCRIPTION

The ws_map_control structure stores information used during a MAP_UNMAP_SCREEN ioctl system call.

RELATED INFORMATION

Appendix C, Driver Routines: map_unmap_screen, ws_map_region

NAME

ws_screen_descriptor – Used by the display driver to describe the characteristics of the graphics display screen

INCLUDE FILE

/usr/sys/include/sys/workstation.h

SYNOPSIS

```
typedef struct {
   short screen;
   ws_monitor monitor_type;
   char moduleID[MODULE_ID_LEN];
   short width;
   short height;
   short root_depth;
   short allowed_depths;
   short nvisuals;
   short x, y;
   short row, col;
   short max_row, max_col;
   short f_width, f_height;
   short cursor_width;
   short cursor_height;
   short min_installed_maps;
   short max_installed_maps;
} ws_screen_descriptor;
```

MEMBERS

screen
: Specifies the ID of the screen.

monitor_type
: Specifies the monitor type, either MONOCHROME or COLOR.

moduleID
: Specifies the string that identifies the module. This is the same string that the sizer –gt command displays.

ws_screen_descriptor

`width`
Specifies the width of the screen in pixels.

`height`
Specifies the height of the screen in pixels.

`root_depth`
Specifies the depth to be used as the root.

`allowed_depths`
Specifies the size of the array of `ws_depth_descriptor` structures describing the depths that are present.

`nvisuals`
Specifies the size of the array of `ws_visual_descriptor` structures describing the visual types that are supported.

`x`, `y`
Specifies the current coordinates of the pointer.

`row`, `col`
Specifies the current text position. The Workstation Subsystem uses this information for console support.

`max_row`, `max_col`
Specifies the maximum row and column position allowed for text. The Workstation Subsystem uses this information for console support.

`f_width`, `f_height`
Specifies the console font width and height. The Workstation Subsystem uses this information for console support.

`cursor_width`, `cursor_height`
Specifies the maximum size of the cursor for this screen.

`min_installed_maps`, `max_installed_maps`
Specifies the minimum and maximum number of colormaps that the screen can support.

DESCRIPTION

The `ws_screen_descriptor` structure describes the characteristics of the graphics display screen.

RELATED INFORMATION

Appendix A, Data Structures: `ws_screens`

Appendix C, Driver Routines: `init_screen`, `ioctl`, `load_cursor`, `map_unmap_screen`, `recolor_cursor`, `set_cursor_position`, `ws_is_mouse_on`, `ws_map_region`, `ws_register_screen`

ws_screen_functions

NAME

`ws_screen_functions` – Used by the display driver to specify the routines that perform screen functions for a particular graphics display

INCLUDE FILE

`/usr/sys/include/sys/wsdevice.h`

SYNOPSIS

```
typedef struct {
    caddr_t (*init_screen_handle)();
    int (*init_screen)();
    int (*clear_screen)();
    int (*scroll_screen)();
    int (*blitc)();
    int (*map_unmap_screen)();
    int (*ioctl)();
    void (*close)();
    caddr_t screen_handle;
    int (*set_get_power_level)();
    int (*screen_private)();
    int (*screen_private2)();
} ws_screen_functions;
```

MEMBERS

`init_screen_handle`
Specifies a pointer to the driver routine that initializes the screen handle.

`init_screen`
Specifies a pointer to the driver routine that initializes the driver's screen handle.

`clear_screen`
Specifies a pointer to the driver routine that clears the graphics display. This is a console driver function; you should implement it as a stub in the display device driver.

`scroll_screen`
Specifies a pointer to the driver routine that scrolls the graphics display when it is used as the console terminal. This is a console driver function; you should implement it as a stub in the display device driver.

`blitc`
Specifies a pointer to the driver routine that displays a character on the graphics display when it is used as the console terminal. This is a console driver function; you should implement it as a stub in the display device driver.

`map_unmap_screen`
Specifies a pointer to the driver routine that performs memory mapping.

`ioctl`
Specifies a pointer to an optional driver routine that handles a board-specific `ioctl` command.

`close`
Specifies a pointer to the driver routine that returns the graphics display to console mode.

`screen_handle`
Specifies the address of driver-specific information.

`set_get_power_level`
Specifies the address of an optional driver routine that performs DPMS power management.

`screen_private`, `screen_private2`
Reserved for future use by DIGITAL.

DESCRIPTION

The `ws_screen_functions` structure contains pointers to the routines that perform screen operations for the display driver.

ws_screen_functions

RELATED INFORMATION

Appendix A, Data Structures: `ws_screens`

Appendix C, Driver Routines: `close`, `init_screen`, `ioctl`, `map_unmap_screen`, `ws_register_screen`

NAME

ws_screens – Used by the display driver to describe the characteristics of a display screen

INCLUDE FILE

/usr/sys/include/sys/wsdevice.h

SYNOPSIS

```
typedef struct {
    ws_screen_descriptor *sp;
    ws_visual_descriptor *vp;
    ws_depth_descriptor *dp;
    ws_screen_functions *f;
    ws_color_map_functions *cmf;
    ws_cursor_functions *cf;
    ws_screen_box adj_screens;
    struct controller *ctlr;
} ws_screens;
```

MEMBERS

sp
: Specifies the address of a ws_screen_descriptor structure, which describes characteristics of the screen, such as its height and width.

vp
: Specifies a pointer to an array of ws_visual_descriptor structures, which describe the visual classes that the screen supports.

dp
: Specifies a pointer to an arrary of ws_depth_descriptor structures, which describe the depths that the screen supports.

f
: Specifies the address of a ws_screen_functions structure, which defines the routines that perform screen functions for this graphics display.

`cmf`
Specifies the address of a `ws_color_map_functions` structure, which defines the routines that perform colormap functions for this graphics display.

`cf`
Specifies the address of a `ws_cursor_functions` structure, which defines the routines that perform cursor functions for this graphics display.

`ctlr`
Specifies the address of the `controller` structure associated with this graphics display device driver.

DESCRIPTION

The `ws_screens` structure includes several data structures to completely describe the characteristics of a screen, including hardware characteristics and the routines that the device driver calls to perform graphics operations.

RELATED INFORMATION

Appendix A, Data Structures: `ws_color_map_functions`, `ws_cursor_functions`, `ws_depth_descriptor`, `ws_screen_descriptor`, `ws_screen_functions`, `ws_visual_descriptor`

Appendix C, Driver Routines: `ws_get_screen`

NAME

ws_visual_descriptor – Used by the display driver to describe the visual classes that the graphics display supports

INCLUDE FILE

/usr/sys/include/sys/workstation.h

SYNOPSIS

```
typedef struct {
    short screen;
    short which_visual;
    short screen_class;
    short depth;
    unsigned long red_mask, green_mask, blue_mask;
    short bits_per_rgb;
    int color_map_entries;
} ws_visual_descriptor;
```

MEMBERS

screen
: Specifies the screen associated with this visual descriptor.

which_visual
: Specifies the visual number.

screen_class
: Specifies the visual class, either StaticGray, GrayScale, StaticColor, PseudoColor, TrueColor, or DirectColor.

depth
: Specifies the number of bits per pixel for this visual.

red_mask, green_mask, blue_mask
: Specifies the red, green, and blue masks for a given visual class.

bits_per_rgb
: Specifies the number of significant bits in the color cell that are used to store red, green, and blue colors.

`color_map_entries`
Specifies the number of colors in the colormap.

DESCRIPTION

The `ws_visual_descriptor` structure defines the characteristics of one server visual class. You usually create an array of these structures to define each of the visual classes that the graphics display supports.

RELATED INFORMATION

Appendix A, Data Structures: `ws_screens`

Appendix C, Driver Routines: `ws_register_screen`

B

Graphics Device ioctl Commands

The reference pages in this appendix describe the `ioctl` commands that a DDX can issue with an `ioctl` system call to the `/dev/ws0` device. The Workstation Subsystem receives the system call and dispatches it to the appropriate device driver routine.

The descriptions of `ioctl` commands can include the following sections.

Name

This section lists the name of the `ioctl` command, along with a short description of its purpose.

Synopsis

This section lists the associated header files for the command and the syntax of the system call that invokes the `ioctl` command.

Arguments

This section provides descriptions of the arguments associated with a given `ioctl` command.

Description

This section provides a more complete description of the purpose of the command, including information on any associated structures and members.

Notes

This section discusses pertinent information regarding the behavior of the `ioctl` command within the server environment.

Restrictions

This section lists any known limitations on the `ioctl` command's behavior or use.

Example

This section povides an example that shows how to use the `ioctl` command within a DDX.

Return Values

This section lists the values that the `ioctl` command returns to the caller.

Related Information

This section lists data structures or routines that are related to the `ioctl` command.

NAME

CURSOR_ON_OFF – Turns the cursor associated with a specific screen on or off

SYNOPSIS

```
#include sys/ioctl.h
#include sys/workstation.h
ioctl(
        int gfx_fd,
        unsigned long CURSOR_ON_OFF,
        void *ws_driver_structure_pointer);
```

ARGUMENTS

gfx_fd
Specifies the file descriptor of the graphics device-special file (by convention /dev/ws0) that the open system call returns when the server is initialized. The server uses this file descriptor in subsequent ioctl calls to the Workstation Subsystem.

CURSOR_ON_OFF
Specifies the ioctl command used to access the ws_cursor_control structure in /usr/sys/include/sys/workstation.h.

ws_driver_structure_pointer
Specifies a pointer to the ws_cursor_control structure defined in /usr/sys/include/sys/workstation.h.

DESCRIPTION

The CURSOR_ON_OFF ioctl command turns the cursor on or off for the screen that is described in the ws_cursor_control structure. The ws_cursor_control structure is defined in /usr/sys/include/sys/workstation.h and pointed to by the **ws_driver_structure_pointer* argument.

RESTRICTIONS

Because DIGITAL does not support multiple seats, the graphics subsystem supports only one workstation per machine.

EXAMPLE

The following example shows how the generic VGA DDX issues the CURSOR_ON_OFF ioctl command from the genInitVGA routine:

```
void genInitVGA(int index)
{
    ws_cursor_control cc;
⋮
   /*
    *  Turn off the cursor
    */

    cc.screen = index;
    cc.control = CURSOR_OFF;

    if (ioctl(wsFd, CURSOR_ON_OFF, &cc) == -1) {
        ErrorF( "error enabling/disabling cursor\n");
        exit(1);
    }
```

RETURN VALUES

If successful, the CURSOR_ON_OFF ioctl command returns 0 (zero). If unsuccessful, it returns –1.

RELATED INFORMATION

Appendix B, ioctl Commands: GET_DEPTH_INFO, MAP_SCREEN_AT_DEPTH, VIDEO_ON_OFF

Appendix C, Driver Routines: close, ioctl

Files: /usr/sys/include/sys/ioctl.h, /usr/sys/include/sys/workstation.h

NAME

GET_DEPTH_INFO – Gets information about the supported depth of a graphics device

SYNOPSIS

```
#include sys/ioctl.h
#include sys/workstation.h
int ioctl(
          int gfx_fd,
          unsigned long GET_DEPTH_INFO,
          void *ws_driver_structure_pointer);
```

ARGUMENTS

gfx_fd
Specifies the file descriptor of the graphics device-special file (by convention /dev/ws0) that the open system call returns when the server is initialized. The server uses this file descriptor in subsequent ioctl calls to the Workstation Subsystem.

GET_DEPTH_INFO
Specifies the ioctl command used to access the ws_depth_descriptor structure in /usr/sys/include/sys/workstation.h.

ws_driver_structure_pointer
Specifies a pointer to the ws_depth_descriptor structure in /usr/sys/include/sys/workstation.h.

DESCRIPTION

The GET_DEPTH_INFO ioctl command gets information about the depth of a graphics device that is described in the ws_depth_descriptor structure. The ws_depth_descriptor structure is defined in /usr/sys/include/sys/workstation.h and pointed to by the **ws_driver_structure_pointer* argument.

The DDX should issue the GET_DEPTH_INFO command after the MAP_SCREEN_AT_DEPTH command, when it returns both the virtual addresses in the pixmap member for the graphics device's memory and the

addresses in the plane_mask member where the device's registers have been mapped. In this way, the server can access the graphics device directly without having to go through the graphics driver.

RESTRICTIONS

Because DIGITAL does not support multiple seats, the graphics subsystem supports only one workstation per machine.

Although most VGA devices support more than one depth of display with 16-bit pixels and 24-bit pixels (also known as true color), the DIGITAL VGA controllers currently support only a single depth (8-bit pixels for all but the generic VGA driver, which supports 4-bit pixels).

EXAMPLE

The following example shows how the generic VGA DDX issues a GET_DEPTH_INFO ioctl command from its genInitVGA routine:

```
ws_depth_descriptor *dpth;

void genInitVGA(int index)
{
⋮
   if (dpth == NULL)
       dpth = (ws_depth_descriptor *)
               Xcalloc(sizeof(ws_depth_descriptor));

       dpth->screen = index;
       if ((err=ioctl(wsFd,GET_DEPTH_INFO, dpth))) {
           perror("Failed to get depth information");
           exit (1);
       }
```

RETURN VALUES

If successful, the GET_DEPTH_INFO ioctl command returns 0 (zero). If unsuccessful, it returns –1.

RELATED INFORMATION

Appendix B, ioctl Commands: CURSOR_ON_OFF, GET_DEPTH_INFO, MAP_SCREEN_AT_DEPTH, VIDEO_ON_OFF

GET_DEPTH_INFO

Appendix C, Driver Routines: `close`, `ioctl`

Files: `/usr/sys/include/sys/ioctl.h`,
`/usr/sys/include/sys/workstation.h`

GET_SCREEN_INFO, SET_SCREEN_INFO

NAME

GET_SCREEN_INFO, SET_SCREEN_INFO – Gets or sets information about a single graphics device

SYNOPSIS

```
#include sys/ioctl.h
#include sys/workstation.h
int ioctl(
          int gfx_fd,
          unsigned long GET_SCREEN_INFO,
          void *ws_driver_structure_pointer,
int ioctl(
          int gfx_fd,
          unsigned long SET_SCREEN_INFO,
          void *ws_driver_structure_pointer);
```

ARGUMENTS

gfx_fd
Specifies the file descriptor of the graphics device-special file (by convention /dev/ws0) that the open system call returns when the server is initialized. The server uses this file descriptor in subsequent ioctl calls to the Workstation Subsystem.

GET_SCREEN_INFO
Specifies the ioctl command used to access the ws_screen_descriptor structure.

SET_SCREEN_INFO
Specifies the ioctl command used to change the ws_screen_descriptor structure.

ws_driver_structure_pointer
Specifies a pointer to the ws_screen_descriptor structure.

DESCRIPTION

The InitOutput function in init.c issues the GET_SCREEN_INFO ioctl command to get information about a single graphics device that is described by the ws_monitor and ws_screen_descriptor structures. These

structures are defined in /usr/sys/include/sys/workstation.h and pointed to by **ws_driver_structure_pointer*. The SET_SCREEN_INFO ioctl command initializes the ws_screen_descriptor structure. The ws_screen_descriptor structure contains data that allows the server to determine which display device is present and to tailor its subsequent handling of the screen in a hardware-dependent manner.

The Workstation Subsystem uses the screen ioctl commands to inquire about and control the configured graphics display devices. Each screen ioctl command requires a screen number as the first field of the data structure passed; the screen number tells the Workstation Subsystem which of the graphics drivers should be called if any hardware-dependent action is necessary.

RESTRICTIONS

Because DIGITAL does not support multiple seats, the graphics subsystem supports only one workstation per machine.

EXAMPLE

The following example shows how the generic VGA DDX issues a GET_SCREEN_INFO ioctl command from its genSetInfo routine:

```
void genSetInfo(ScreenPtr pScreen, int index)
{
    ws_screen_descriptor wsd;

    wsd.screen = index;
    wsd.width = pScreen->width;
    wsd.height = pScreen->height;
    strcpy(wsd.moduleID, "Set_Info");
    ioctl(wsFd, GET_SCREEN_INFO, &wsd);
    screenDesc[index].width = pScreen->width;
    screenDesc[index].height = pScreen->height;
}
```

RETURN VALUES

If successful, the GET_SCREEN_INFO and SET_SCREEN_INFO ioctl commands return 0 (zero). If unsuccessful, they return –1.

RELATED INFORMATION

Appendix B, `ioctl` Commands: `CURSOR_ON_OFF`, `GET_DEPTH_INFO`, `MAP_SCREEN_AT_DEPTH`, `VIDEO_ON_OFF`

Appendix C, Driver Routines: `close`, `ioctl`

Files: `/usr/sys/include/sys/ioctl.h`, `/usr/sys/include/sys/workstation.h`

NAME

MAP_SCREEN_AT_DEPTH – Maps a graphics device's memory and its registers to the server's virtual address space

SYNOPSIS

```
#include sys/ioctl.h
#include sys/workstation.h

int ioctl(
        int gfx_fd,
        unsigned long MAP_SCREEN_AT_DEPTH,
        void *ws_driver_structure_pointer);
```

ARGUMENTS

gfx_fd
Specifies the file descriptor of the graphics device-special file (by convention /dev/ws0) that the open system call returns when the server is initialized. The server uses this file descriptor in subsequent ioctl calls to the Workstation Subsystem.

MAP_SCREEN_AT_DEPTH
Specifies the ioctl command used to access the ws_map_control structure.

ws_driver_structure_pointer
Specifies a pointer to the ws_map_control structure.

DESCRIPTION

The MAP_SCREEN_AT_DEPTH ioctl command tells the graphics driver to map the graphics device's memory and its registers into the virtual address space of the server. This allows the server to access the graphics hardware directly. The virtual addresses that are allocated when MAP_SCREEN_AT_DEPTH is invoked are returned in a subsequent call to GET_DEPTH_INFO. MAP_SCREEN_AT_DEPTH maps frame buffer memory; GET_DEPTH_INFO returns the virtual address of the mapping.

This ioctl command accesses the ws_map_control data structure defined in /usr/sys/include/sys/workstation.h.

RESTRICTIONS

Because DIGITAL does not support multiple seats, the graphics subsystem supports only one workstation per machine.

Although most VGA devices support more than one depth of display with 16-bit pixels and 24-bit pixels (also known as `TrueColor`), the DIGITAL VGA controllers currently support only a single depth (8-bit pixels for all but the generic VGA driver, which supports 4-bit pixels).

EXAMPLES

The following example shows how the generic VGA DDX issues a `MAP_SCREEN_AT_DEPTH` `ioctl` command from the `genInitVGA` routine:

```
static ws_map_control * map;

void genInitVGA(int index)
{
⋮
    if (map == NULL)
        map = (ws_map_control *) Xcalloc(sizeof(map));

    if (!init_flags[index]) {
        map->map_unmap = MAP_SCREEN;
        map->screen = index;
        if ((err=ioctl(wsFd, MAP_SCREEN_AT_DEPTH, map))) {
            perror("Failed to map screen at depth");
            exit (1);
        }
        init_flags[index] = 1;
    }
⋮
```

RETURN VALUES

If successful, the `MAP_SCREEN_AT_DEPTH` `ioctl` command returns 0 (zero). If unsuccessful, it returns –1.

RELATED INFORMATION

Appendix B, `ioctl` Commands: `CURSOR_ON_OFF`, `GET_DEPTH_INFO`, `MAP_SCREEN_AT_DEPTH`, `VIDEO_ON_OFF`

MAP_SCREEN_AT_DEPTH

Appendix C, Driver Routines: `close`, `ioctl`

Files: `/usr/sys/include/sys/ioctl.h`,
`/usr/sys/include/sys/workstation.h`

SET_CURSOR_POSITION

NAME

SET_CURSOR_POSITION – Moves the cursor to a specified position

SYNOPSIS

```
#include sys/ioctl.h
#include sys/workstation.h
ioctl(
        int gfx_fd,
        unsigned long SET_CURSOR_POSITION,
        void *ws_driver_structure_pointer);
```

ARGUMENTS

gfx_fd
Specifies the file descriptor of the graphics device-special file (by convention /dev/ws0) that the open system call returns when the server is initialized. The server uses this file descriptor in subsequent ioctl calls into the Workstation Subsystem.

SET_CURSOR_POSITION
Specifies the ioctl command used to access the ws_cursor_position structure in /usr/sys/include/sys/workstation.h.

ws_driver_structure_pointer
Specifies a pointer to the ws_cursor_position structure in /usr/sys/include/sys/workstation.h.

DESCRIPTION

The SET_CURSOR_POSITION ioctl command causes the cursor to move to the specified coordinates. The screen is specified by the screen member of the ws_cursor_position structure; the coordinates are specified by the x and y members of the ws_cursor_position structure. The structure is defined in /usr/sys/include/sys/workstation.h.

RESTRICTIONS

Because DIGITAL does not support multiple seats, the graphics subsystem supports only one workstation per machine.

RETURN VALUES

If successful, the SET_CURSOR_POSITION ioctl command returns 0 (zero). If unsuccessful, it returns –1.

RELATED INFORMATION

Appendix B, ioctl Commands: CURSOR_ON_OFF, GET_DEPTH_INFO, MAP_SCREEN_AT_DEPTH, VIDEO_ON_OFF

Appendix C, Driver Routines: close, ioctl

Files: /usr/sys/include/sys/ioctl.h, /usr/sys/include/sys/workstation.h

VIDEO_ON_OFF

NAME

VIDEO_ON_OFF – Turns the video on or off on a specific screen

SYNOPSIS

```
#include sys/ioctl.h
#include sys/workstation.h

ioctl(
        int gfx_fd,
        unsigned long VIDEO_ON_OFF,
        void *ws_driver_structure_pointer);
```

ARGUMENTS

gfx_fd
Specifies the file descriptor of the graphics device-special file (by convention /dev/ws0) that the open system call returns when the server is initialized. The server uses this file descriptor in subsequent ioctl calls to the Workstation Subsystem.

VIDEO_ON_OFF
Specifies the ioctl command used to access the ws_video_control structure.

ws_driver_structure_pointer
Specifies a pointer to the ws_video_control structure.

DESCRIPTION

The VIDEO_ON_OFF ioctl command turns the video on or off. The screen is described by the ws_video_control structure defined in /usr/sys/include/sys/workstation.h and pointed to by *ws_driver_structure_pointer*. Primarily, a screen saver uses this ioctl command, which is called from the WS layer of the DIGITAL DDX implementation. However, if you want to use some other method of blanking the screen (for example, to implement Green Mode or Deep Green Mode), then you would call this ioctl command from your board-specific DDX.

RESTRICTIONS

Because DIGITAL does not support multiple seats, the graphics subsystem supports only one workstation per machine.

EXAMPLE

The following example shows how the generic WS layer of the DDX uses the VIDEO_ON_OFF ioctl command:

```
Bool
wsSaveScreen(pScreen, on)
    ScreenPtr pScreen;
    int on;
{
    ws_video_control vc;
    vc.screen = WS_SCREEN(pScreen);
    if (on == SCREEN_SAVER_FORCER) {
 lastEventTime = queue->time;
    } else if (on == SCREEN_SAVER_ON) {
 vc.control = 0;
 if (ioctl(wsFd, VIDEO_ON_OFF, &vc) < 0)
     ErrorF("VIDEO_ON_OFF: failed to turn screen off.\n");
    } else {
 vc.control = 1;
 if (ioctl(wsFd, VIDEO_ON_OFF, &vc) < 0)
     ErrorF("VIDEO_ON_OFF: failed to turn screen on.\n");
    }
    return TRUE;
}
```

RETURN VALUES

If successful, the VIDEO_ON_OFF ioctl command returns 0 (zero). If unsuccessful, it returns –1.

RELATED INFORMATION

Appendix B, ioctl Commands: CURSOR_ON_OFF, GET_DEPTH_INFO, MAP_SCREEN_AT_DEPTH, SET_CURSOR_POSITION

Appendix C, Driver Routines: close, ioctl

Files: /usr/sys/include/sys/ioctl.h, /usr/sys/include/sys/workstation.h

C

Graphics Device Driver Routines

The reference pages in this appendix describe routines that a board-specific graphics driver uses to interface with the Workstation Subsystem, including cursor, colormap, and screen functions. Each group of functions (cursor, colormap, and screen) also shares a corresponding `init_handle` function, which initializes the private data structure specific to the group.

Routine descriptions contain the following sections:

Name

This section lists the name of the routine along with a summary of its purpose.

Synopsis

This section contains the function prototype, which gives you the following information:

- Return type

  The data type of the return value, in bold font, or `void` if the routine does not return a value.

- Routine name

  The name of the routine, in bold font. Note that routine names are case sensitive.

- Argument data type

  The argument's C type definition. Data type keywords are in bold font.

- Argument name

  The argument name, in italic font.

Arguments

This section contains a description of each argument.

Data Structures

If the routine accesses any global data structures, they are listed in this section along with the name of the header file that defines them.

Description

This section explains the tasks that the routine performs.

Return value

This section shows the return values that the routine can return, or “None” if no value is returned. If the routine returns an error value, this value is also described in the Return Value section.

Notes

This section discusses information that falls into the following categories:

- Hardware-specific information

  Some interfaces behave differently depending on the architecture of the hardware.

- Operating system-specific information

  Some interfaces behave differently depending on the implementation of the operating system.

- Information pertinent to device drivers

  Some interfaces require specific information important to the device driver writer.

Example

This section provides an example that shows how you typically use a routine or data structure.

Related Information

This section lists related structures and routines. You can refer to the reference pages for these structure and routines for additional information.

NAME

clean_color_map – Cleans dirty colormap entries

SYNOPSIS

```
void (*ws_color_map_functions->clean_color_map)(
        caddr_t colormap_handle);
```

ARGUMENTS

colormap_handle
Specifies the virtual address (handle) of device-specific information. Typically this is a pointer to a private data structure that may contain information such as the address of the hardware, state information, and other information that may be shared between drivers. The Workstation Subsystem allows separate handles for the cursor, colormap, and the screen functions.

DESCRIPTION

The clean_color_map routine cleans any colormap entries found to be dirty in the driver's colormap data structures, which are typically accessed through the *colormap_handle* argument.

RETURN VALUES

None

EXAMPLES

The following code is from the DIGITAL implementation of clean_color_map in the myvga example driver:

```
void
myvga_clean_color_map(caddr_t colormap_handle)
{
   register struct myvga_type *scp =
      (struct myvga_type *)colormap_handle;
   register struct myvga_color_cell *entry;
   register int i, s, lasti;
```

clean_color_map

```
        if (myvga_developer_debug)
           printf("myvga_clean_color_map: entry\n");

        if (!IS_MYVGA_DIRTY_CMAP(scp))
           return;

        /*
         * No interrupts when using autoincrement mode of VDAC
         */
        s = splbio();

        /*
         * Change the "dirty" colormap entries.
         */
        entry = &scp->cells[scp->min_dirty];
        lasti = -2;
        for (i = scp->min_dirty; i <= scp->max_dirty ; i++, entry++)
        {
           if (entry->dirty_cell)
           {
      if (i != (lasti + 1))
         OUTB(MYVGA_PEL_ADDR_WMODE, i);

      OUTB(MYVGA_PEL_DATA, entry->red);
      OUTB(MYVGA_PEL_DATA, entry->green);
      OUTB(MYVGA_PEL_DATA, entry->blue);

      entry->dirty_cell = 0;
      lasti = i;
           }
        }

        /*
         * Reset to "clean" status
         */
        scp->min_dirty = 256;
        scp->max_dirty = 0;
        CLR_MYVGA_DIRTY_CMAP(scp);

        splx(s);

    }
```

RELATED INFORMATION

Appendix A, Data Structures: `ws_color_map_functions`

Appendix C, Driver Routines: `init_color_map`, `init_colormap_handle`, `load_color_map_entry`, `video_off`, `video_on`

Files: `/usr/sys/include/sys/workstation.h`, `/usr/sys/include/sys/wsdevice.h`.

close

NAME

`close` – Puts the display into console state when the Workstation Subsystem is closed

SYNOPSIS

```
void (*ws_screen_functions->close)(
        caddr_t screen_handle);
```

ARGUMENTS

screen_handle
: Specifies the virtual address (handle) of device-specific information. Typically this is a pointer to a private data structure that may contain information such as the address of the hardware, state information, and other information that may be shared between drivers. The Workstation Subsystem allows separate handles for the cursor, colormap, and the screen functions.

DESCRIPTION

The Workstation Subsystem calls the `close` function whenever the Workstation Subsystem is closed, unless the function is defined as NULL. The device driver should call this function when changing to console mode.

You group the `close` function with the screen functions in the `ws_screen_functions` structure in `/usr/sys/include/sys/wsdevice.h`.

RETURN VALUES

None

EXAMPLES

The following code is from the DIGITAL implementation of `close` in the myvga example driver:

```
void
myvga_close(caddr_t screen_handle)
```

```
{
    register struct myvga_type *scp =
        (struct myvga_type *)screen_handle;

    if (myvga_developer_debug)
        printf("myvga_close: entry\n");
}
```

RELATED INFORMATION

Appendix A, Data Structures: `ws_screen_functions`

Appendix C, Driver Routines: `init_screen`, `init_screen_handle`, `ioctl`, `map_unmap_screen`

Files: `/usr/sys/include/sys/workstation.h`, `/usr/sys/include/sys/wsdevice.h`

console_attach

NAME

console_attach – Attaches the named controller as the kernel console driver

SYNOPSIS

```
int console_attach(
        struct controller *ctlr);
```

ARGUMENTS

ctlr
Specifies the controller to be attached as the kernel console driver.

DATA STRUCTURES

The driver's console_attach interface accesses the controller structure defined in /usr/sys/include/io/common/devdriver.h.

DESCRIPTION

The console_attach function is an entry point to the graphics driver that attaches the controller (pointed to by *ctlr*) as the kernel console driver.

A console_attach function must be implemented for each graphics driver. Before calling console_attach, you initialize the console_attach member of the controller structure with the name of this function. The console_attach member is a private member defined in /usr/sys/include/arch/alpha/hal/console.h.

Note that for VGA-class drivers, you can call install_vga_console to do the actual work of attaching the graphics console to the system.

RETURN VALUES

If successful, the interface returns 0 (zero). If unsuccessful, it returns –1.

EXAMPLES

The following example shows how the myvga device driver implements the console_attach interface:

```
int
myvga_console_attach(struct controller *ctlr)
{
   int status = -1;  /* Failure */

   if (myvga_developer_debug)
      printf("MYVGA_console_attach entrypoint\n");

   /*
    * Use standard VGA console support.
    * Set global to indicate use of
    * current board VGA register state
    * to be used for setting text mode 3.
    * The default is to reinitialize the
    * board/VGA to VGA mode 3 settings.
    *
    */
   vga_use_orig_state = 1;
   status = install_vga_console(ctlr);

   if (status == -1) {
      printf("myvga: could not support vga console\n");
   }

   return(status);
}
```

RELATED INFORMATION

Appendix C, Driver Routines: `install_vga_console`

Files: `/usr/sys/include/io/common/devdriver.h`

NAME

`cursor_on_off` – Turns the cursor on and off

SYNOPSIS

```
int (*ws_cursor_functions->cursor_on_off)(
        caddr_t cursor_handle,
        int on_off);
```

ARGUMENTS

cursor_handle
: Specifies the virtual address (handle) of device-specific information. Typically this is a pointer to a private data structure that may contain information such as the address of the hardware, state information, and other information that may be shared between drivers. The Workstation Subsystem allows separate handles for the cursor, colormap, and the screen functions.

on_off
: Specifies whether the cursor is on or off. A positive number indicates that the cursor is on.

DESCRIPTION

The `cursor_on_off` function turns the cursor on and off.

RETURN VALUES

If successful, the `cursor_on_off` function returns 0 (zero). If unsuccessful, it returns –1.

EXAMPLES

The following example shows how an ATI Mach64 graphics adapter might implement the `cursor_on_off` function:

```
ati64_cursor_on_off(cursor_handle, on_off)
    caddr_t cursor_handle;
    int on_off;
```

```
{
    register struct vga_info *vp =
            (struct vga_info *)cursor_handle;
    register struct ati64_type *ap =
            &ati64_info[vp->unit];
    unsigned int data = 0;

    /* Set local driver flags. */
    if (on_off)
        SET_VGA_CURSOR_ON(vp);
    else
        CLR_VGA_CURSOR_ON(vp);

    /* Do not bother if screen is OFF. */
    if (!IS_VGA_SCREEN_ON(vp))
        return(0);

    /*
     * The following code assumes that the cursor position is
     * up-to-date (by ati64_cursor_set_position), so that
     * simply turning the cursor back on will find it at
     * the desired spot.
     */
    data = REGR(vp, ap, GEN_TEST_CNTL);
    if (on_off) {
        /*
         * Turn ON the cursor by setting
         * bit 7 of the GEN_TEST_CNTL register.
         */
        data |= 0x80;
    }
    else {
        /*
         * Turn OFF the cursor by resetting
         * bit 8 of the GEN_TEST_CNTL register.
         */
        data &= ~0x80;
    }
    REGW(vp, ap, GEN_TEST_CNTL, data);

   return(0);
}
```

RELATED INFORMATION

Appendix A, Data Structures: `ws_cursor_functions`

Appendix C, Driver Routines: `init_cursor_handle`, `load_cursor`, `recolor_cursor`, `set_cursor_position`

Files: `/usr/sys/include/sys/workstation.h`, `/usr/sys/include/sys/wsdevice.h`

NAME

drvr_register_saveterm – Registers or deregisters a saveterm interface with the kernel

SYNTAX

```
void drvr_register_saveterm(
          void (*callback)(),
          caddr_t param,
          int flags);
```

ARGUMENTS

callback
Specifies the name of the callback routine.

param
Specifies the parameter to be passed to the callback routine.

flags
Specifies whether the drvr_register_saveterm interface should register or deregister the callback routine. Valid values for this argument are DRVR_REGISTER and DRVR_UNREGISTER.

DESCRIPTION

The drvr_register_saveterm interface either registers or deregisters a saveterm callback routine. The kernel calls this routine when the user presses the halt button. Graphics display device drivers provide a saveterm routine to reset the hardware for use by the console firmware if any special action is needed to return the controller to console mode.

RETURN VALUE

None

NAME

`init_color_map` – Resets the colormap to a state where console output can be read on the screen

SYNOPSIS

```
int (*ws_color_map_functions->init_color_map)(
        caddr_t colormap_handle);
```

ARGUMENTS

colormap_handle
: Specifies the virtual address (handle) of device-specific information. Typically this is a pointer to a private data structure that may contain information such as the address of the hardware, state information, and other information that may be shared between drivers. The Workstation Subsystem allows separate handles for the cursor, colormap, and the screen functions.

DESCRIPTION

The Workstation Subsystem calls the `init_color_map` function whenever the Workstation Subsystem is opened, closed, or whenever the console screen is initialized. The function resets the colormap so that output to the console can be read on the console screen.

RETURN VALUES

If successful, the `init_color_map` function returns 0 (zero). If unsuccessful, it returns –1.

EXAMPLES

The following code is from the DIGITAL implementation of `init_color_map` in the myvga example driver:

```
int
myvga_init_color_map(caddr_t colormap_handle)
{
   register struct myvga_type *scp =
```

```
      (struct myvga_type *)colormap_handle;
   register unsigned char *cp =
           &((vgaHWPtr)scp->new_state)->DAC[0];
   register int i;

   if (myvga_developer_debug)
      printf("myvga_init_color_map: entry\n");

   /* NOTE: using 8-bit values */
   OUTB(MYVGA_PEL_MASK, 0xff);
   OUTB(MYVGA_PEL_ADDR_WMODE, 0);
   for (i = 0; i < 768; i++)
      OUTB(MYVGA_PEL_DATA, *cp++);

   return(0);
}
```

RELATED INFORMATION

Appendix A, Data Structures: `ws_color_map_functions`

Appendix C, Driver Routines: `clean_color_map`, `init_colormap_handle`, `load_color_map_entry`, `video_off`, `video_on`

Files: `/usr/sys/include/sys/workstation.h`, `/usr/sys/include/sys/wsdevice.h`

init_colormap_handle

NAME

`init_colormap_handle` – Initializes the handle to the colormap functions

SYNOPSIS

```
caddr_t init_colormap_handle(
          caddr_t colormap_handle,
          caddr_t address,
          int unit,
          int type);
```

ARGUMENTS

colormap_handle
: Specifies the virtual address (handle) of device-specific information. Typically this is a pointer to a private data structure that may contain information such as the address of the hardware, state information, and other information that may be shared between drivers. The Workstation Subsystem allows separate handles for the cursor, colormap, and the screen functions.

address
: Specifies the virtual address of the *`colormap_handle`* argument.

unit
: Specifies the unit number of the controller that the graphics card is attached to.

type
: Specifies the type of graphics card.

DESCRIPTION

The `init_colormap_handle` function initializes the *`colormap_handle`* argument for each graphics device driver.

RETURN VALUES

On success, the `init_colormap_handle` function returns a pointer to a colormap handle. This pointer is NULL if an error occurs.

EXAMPLES

The following code is from the DIGITAL implementation of `init_colormap_handle` in the myvga example driver:

```
caddr_t
myvga_init_color_map_handle (caddr_t colormap_handle,
        caddr_t address,
                            int unit,
                            int type)
{
    struct myvga_type *scp =
              ((struct myvga_type **)colormap_handle)[unit];

    if (myvga_developer_debug)
        printf("myvga_color_map_init_colormap_handle: entry\n");

    return (caddr_t)scp ;
}
```

RELATED INFORMATION

Appendix A, Data Structures: `ws_color_map_functions`

Appendix C, Driver Routines: `clean_color_map`, `init_color_map`, `load_color_map_entry`, `video_off`, `video_on`

Files: `/usr/sys/include/sys/workstation.h`, `/usr/sys/include/sys/wsdevice.h`

NAME

`init_cursor_handle` – Initializes the cursor handle for the specified graphics card

SYNOPSIS

```
caddr_t init_cursor_handle(
          caddr_t cursor_handle,
          caddr_t address,
          int unit,
          int type);
```

ARGUMENTS

cursor_handle
: Specifies the virtual address (handle) of device-specific information. Typically this is a pointer to a private data structure that may contain information such as the address of the hardware, state information, and other information that may be shared between drivers. The Workstation Subsystem allows separate handles for the cursor, colormap, and the screen functions.

address
: Specifies the virtual address of the *cursor_handle* argument.

unit
: Specifies the unit number of the controller that the graphics card is attached to.

type
: Specifies the type of graphics card.

DESCRIPTION

The `init_cursor_handle` function initializes the *cursor_handle* argument for the cursor for each graphics device driver.

RETURN VALUES

On success, the `init_cursor_handle` function returns a pointer to a cursor handle. This pointer is NULL if an error occurs.

EXAMPLES

The following code is from the DIGITAL implementation of `init_cursor_handle` in the myvga example driver:

```
caddr_t
myvga_init_cursor_handle(caddr_t cursor_handle,
    caddr_t address,
                         int unit,
                         int type)
{
    struct myvga_type *scp =
              ((struct myvga_type **)cursor_handle)[unit];

    if (myvga_developer_debug)
        printf("myvga_cursor_init_cursor_handle: entry\n");

    return (caddr_t)scp ;
}
```

RELATED INFORMATION

Appendix A, Data Structures: `ws_cursor_functions`

Appendix C, Driver Routines: `cursor_on_off`, `load_cursor`, `recolor_cursor`, `set_cursor_position`

Files: `/usr/sys/include/sys/workstation.h`, `/usr/sys/include/sys/wsdevice.h`

NAME

`init_screen` – Initializes the screen

SYNOPSIS

```
int (*ws_screen_functions->init_screen)(
        caddr_t screen_handle,
        ws_screen_descriptor *screen);
```

ARGUMENTS

screen_handle
: Specifies the virtual address (handle) of device-specific information. Typically this is a pointer to a private data structure that may contain information such as the address of the hardware, state information, and other information that may be shared between drivers. The Workstation Subsystem allows separate handles for the cursor, colormap, and the screen functions.

screen
: Specifies a pointer to the `ws_screen_descriptor` structure, which describes the attributes of the screen.

DATA STRUCTURES

The `init_screen` function accesses the `ws_screen_descriptor` structure defined in `/usr/sys/include/sys/workstation.h`.

DESCRIPTION

The Workstation Subsystem calls the `init_screen` function when the graphics screen is first initialized.

RETURN VALUES

If successful, the `init_screen` function returns 0 (zero). If unsuccessful, it returns –1.

EXAMPLES

The following code is from the DIGITAL implementation of `init_screen` in the myvga example driver:

```
myvga_init_screen(caddr_t screen_handle,
    ws_screen_descriptor *screen)
{
   register struct myvga_type *scp =
           (struct myvga_type *)screen_handle;
   register ws_screen_descriptor *sp = &scp->screen;

   if (myvga_developer_debug)
       printf("myvga_init_screen: entry\n");

   /*
    *   Save and restore the standard VGA registers
    *   to ensure they get set up, particularly for
    *   the case when in serial console mode and the
    *   server is started.
    */

   /*
    * Save the MYVGA register contents left behind by
    * the console.
    */
   VGA_save_registers(scp->orig_state);

   /*
    *  Reset the registers to text mode 3 values.
    */
   VGA_restore_registers(scp->new_state);

   return 0;
}
```

RELATED INFORMATION

Appendix A, Data Structures: `ws_screen_functions`

Appendix C, Driver Routines: `close`, `init_screen_handle`, `ioctl`, `map_unmap_screen`

Files: `/usr/sys/include/sys/workstation.h`, `/usr/sys/include/sys/wsdevice.h`

NAME

`init_screen_handle` – Initializes the screen handle for the specified graphics card

SYNOPSIS

```
caddr_t init_screen_handle(
          caddr_t screen_handle,
          caddr_t address,
          int unit,
          int type);
```

ARGUMENTS

screen_handle
: Specifies the virtual address (handle) of device-specific information. Typically this is a pointer to a private data structure that may contain information such as the address of the hardware, state information, and other information that may be shared between drivers. The Workstation Subsystem allows separate handles for the cursor, colormap, and the screen functions.

address
: Specifies the virtual address of the *screen_handle* argument.

unit
: Specifies the unit number of the controller that the graphics card is attached to.

type
: Specifies the type of graphics card.

DESCRIPTION

The `init_screen_handle` function initializes the *screen_handle* argument for the screen for each graphics device driver.

RETURN VALUES

On success, the `init_screen_handle` function returns a pointer to a screen handle. This pointer is NULL if an error occurs.

EXAMPLES

The following code is from the DIGITAL implementation of `init_screen_handle` in the myvga example driver:

```
caddr_t
myvga_init_screen_handle (caddr_t screen_handle,
     caddr_t address,
                          int unit,
                          int type)
{
    register struct myvga_type *scp =
            ((struct myvga_type **)screen_handle)[unit];
    register int i;

    scp->unit = unit;

    if (myvga_developer_debug)
        printf("myvga_init_screen_handle: screen=%d entry\n",
               scp->screen.screen);

   return (caddr_t) scp;
}
```

RELATED INFORMATION

Appendix A, Data Structures: `ws_screen_functions`

Appendix C, Driver Routines: `close`, `init_screen`, `ioctl`, `map_unmap_screen`

Files: `/usr/sys/include/sys/workstation.h`, `/usr/sys/include/sys/wsdevice.h`

NAME

`install_vga_console` – Installs a VGA console as the system console

SYNOPSIS

```
int install_vga_console(
        struct controller *ctlr);
```

ARGUMENTS

ctlr
Specifies a pointer to the driver's `controller` structure.

DATA STRUCTURES

The `install_vga_console` routine accesses the driver's `controller` structure defined in `/usr/sys/include/io/common/devdriver.h`.

DESCRIPTION

The `install_vga_console` routine lets you register the VGA console implemented by DIGITAL with the Workstation Subsystem. If you are implementing a VGA-class card or a compound card that uses a VGA chip for the console, you can call this routine instead of writing your own VGA graphics console driver. You call the `install_vga_console` routine from the device driver's `console_attach` routine.

RETURN VALUES

If successful, the `install_vga_console` routine returns 0 (zero) or a positive integer. If unsuccessful, it returns –1.

EXAMPLES

The `myvga_console_attach` routine calls `install_vga_console` as follows to install the VGA console as the system console:

```
int
myvga_console_attach(struct controller *ctlr)
{
```

```
.
.
.
  status = install_vga_console(ctlr);
.
.
.
}
```

RELATED INFORMATION

Appendix C, Driver Routines: `console_attach`

Files: `/usr/sys/include/io/common/devdriver.h`
`/usr/sys/include/io/dec/ws/vga_support.h`

NAME

`ioctl` – Allows the screen to implement arbitrary I/O control

SYNOPSIS

```
int (*ws_screen_functions->ioctl)(
        ws_screen_descriptor *screen,
        int cmd,
        caddr_t data,
        int flag);
```

ARGUMENTS

screen
Specifies a pointer to the typedef structure `ws_screen_descriptor`, which describes the attributes of the screen.

cmd
Specifies the command to be executed.

data
Specifies the virtual address where the data is to reside.

flag
Specifies a flag that may be passed by the function.

DATA STRUCTURES

The `ioctl` function accesses the `ws_screen_descriptor` structure defined in `/usr/sys/include/sys/workstation.h`.

DESCRIPTION

The `ioctl` function allows the specified screen to perform driver-specific `ioctl` commands. As a result, this function allows you to implement functions that the Workstation Subsystem does not provide. For example, you might use this function to implement an `ioctl` command that blocks the server until the graphics device is again free.

The definition of the `ioctl` command must be in a header file to be included by the DDX component for the adapter. This is an optional function, and if you provide a stub, it will never be called.

The ioctl function is grouped with the screen functions in the ws_screen_functions structure in /sys/wsdevice.h.

RETURN VALUES

The individual ioctl command determines the value that it returns.

EXAMPLES

The following example shows how the myvga example driver implements the ioctl function:

```
int
myvga_ioctl(caddr_t screen_handle,
            int request,
            caddr_t data)
{
   int i,j;
   int status = 0;
   struct myvga_ioc_type *ioctp =
           (struct myvga_ioc_type *)data;

   if (myvga_developer_debug)
      printf("myvga_ioctl:\n");

   switch(request) {
      case MYVGA_IOC_FLASH:
        if (myvga_developer_debug)

           printf("\tMYVGA_IOC_FLASH\n");
        myvga_video_off(screen_handle);
        DELAY(10000);
        myvga_video_on(screen_handle);
        break;

    default:
        if (myvga_developer_debug)
           printf("\tUnsupported ioctl command\n");
        break;
   }

   return status;
}
```

RELATED INFORMATION

Appendix A, Data Structures: `ws_screen_functions`

Appendix C, Driver Routines: `close`, `init_screen`, `init_screen_handle`, `map_unmap_screen`

Files: `/usr/sys/include/sys/workstation.h`, `/usr/sys/include/sys/wsdevice.h`

NAME

load_color_map_entry – Loads the colormap entry into a colormap

SYNOPSIS

```
int (*ws_color_map_functions->load_color_map_entry)(
        caddr_t colormap_handle,
        int map,
        ws_color_cell *entry);
```

ARGUMENTS

colormap_handle
Specifies the virtual address (handle) of device-specific information. Typically this is a pointer to a private data structure that may contain information such as the address of the hardware, state information, and other information that may be shared between drivers. The Workstation Subsystem allows separate handles for the cursor, colormap, and the screen functions.

map
Specifies the colormap map.

entry
Specifies a pointer to the typedef structure ws_color_cell, which contains the colormap entry index and the values of red, green, and blue that are to be loaded into the colormap map.

DATA STRUCTURES

The load_color_map_entry function accesses the ws_color_cell, and ws_cursor_data structures defined in /usr/sys/include/sys/workstation.h.

DESCRIPTION

The load_color_map_entry function loads the specified colormap entry with the values of red, green, and blue. This function can wait to load the entry until vertical refresh interrupt time, but it cannot wait for the interrupt to load the colormap. Also, the function must return immediately in order not to affect the performance of X clients.

RETURN VALUES

If successful, the load_color_map_entry function returns 0 (zero). If unsuccessful, it returns –1.

EXAMPLES

The following code is from the DIGITAL implementation of load_color_map_entry in the myvga example driver:

```
int
myvga_load_color_map_entry_6bit(caddr_t colormap_handle,
                                int map,
                                register ws_color_cell *entry)
{
   struct myvga_type *scp =
           (struct myvga_type *)colormap_handle;
   int shift = 10;
   register int index = entry->index;
   register unsigned int mask = 0x0000ffff >> shift;
   int s;

   if (myvga_developer_debug)
      printf("myvga_load_color_map_entry_6bit: entry 0x%x = ",
             entry->index);
      printf("(0x%x, 0x%x, 0x%x)\n",
             entry->red, entry->green, entry->blue);

   SET_MYVGA_6BIT_DAC(scp);

   /* Update CLUT database with 6-bit DAC values */

   if (index >= 256 || index < 0)
      return(-1);

   s = splbio();

   scp->cells[index].red   = (entry->red   >> shift) & mask;
   scp->cells[index].green = (entry->green >> shift) & mask;
   scp->cells[index].blue  = (entry->blue  >> shift) & mask;
   scp->cells[index].dirty_cell = 1;

   if (index < scp->min_dirty)
      scp->min_dirty = index;
   if (index > scp->max_dirty)
```

```
      scp->max_dirty = index;

   splx(s);

   /*
    * Enable a VBLANK interrupt to load dirty color cells.
    * If VBLANK interrupts are not being used, do it now.
    */
   if (!IS_MYVGA_DIRTY_CMAP(scp)) {
      SET_MYVGA_DIRTY_CMAP(scp);
      if (!myvga_enable_interrupt(scp))
         myvga_clean_color_map(colormap_handle);
   }

   return (0);
}
```

RELATED INFORMATION

Appendix A, Data Structures: `ws_color_map_functions`

Appendix C, Driver Routines: `init_color_map`, `init_colormap_handle`, `video_off`, `video_on`

Files: `/usr/sys/include/sys/workstation.h`, `/usr/sys/include/sys/wsdevice.h`

load_cursor

NAME

`load_cursor` – Loads the cursor

SYNOPSIS

```
int (*ws_cursor_functions->load_cursor)(
        caddr_t cursor_handle,
        ws_screen_descriptor *screen,
        ws_cursor_data *cursor);
```

ARGUMENTS

cursor_handle
: Specifies the virtual address (handle) of device-specific information. Typically this is a pointer to a private data structure that may contain information such as the address of the hardware, state information, and other information that may be shared between drivers. The Workstation Subsystem allows separate handles for the cursor, colormap, and the screen functions.

screen
: Specifies a pointer to the `ws_screen_descriptor` structure, which describes the attributes of the screen.

cursor
: Specifies a pointer to the `ws_cursor_data` structure, which describes the cursor.

DATA STRUCTURES

The `load_cursor` function accesses the `ws_color_cell` and `ws_cursor_data` structures defined in `/usr/sys/include/sys/workstation.h`.

DESCRIPTION

The `load_cursor` function loads the cursor with the cursor specified by **cursor*. Since this function may need information about the screen the cursor is associated with, the screen structure **screen* is also passed.

RETURN VALUES

If successful, the load_cursor function returns 0 (zero). If unsuccessful, it returns –1.

EXAMPLES

The following example shows how the myvga example driver implements the load_cursor function:

```
int
myvga_load_cursor(caddr_t cursor_handle,
    ws_screen_descriptor *screen,
                ws_cursor_data *cursor)
{
   register struct myvga_type *scp =
           (struct myvga_type *)cursor_handle;

   scp->x_hot = cursor->x_hot;
   scp->y_hot = cursor->y_hot;

   if (myvga_developer_debug)
      printf("myvga_load_cursor: entry: screen=%d hot (%d,%d)\n",
      scp->screen.screen, scp->x_hot, scp->y_hot);

   myvga_set_cursor_position(cursor_handle, screen,
        screen->x, screen->y);

   return(0);
}
```

RELATED INFORMATION

Appendix A, Data Structures: ws_color_cell, ws_cursor_data, ws_cursor_functions

Appendix C, Driver Routines: cursor_on_off, init_cursor_handle, load_cursor, recolor_cursor, set_cursor_position

Files: /usr/sys/include/sys/workstation.h, /usr/sys/include/sys/wsdevice.h

map_unmap_screen

NAME

`map_unmap_screen` – Creates the memory map for the specified device

SYNOPSIS

```
int (*ws_screen_functions->map_unmap_screen)(
        caddr_t screen_handle,,
        ws_depth_descriptor *depth,,
        ws_screen_descriptor *screen,,
        ws_map_control *mc);
```

ARGUMENTS

screen_handle
: Specifies the virtual address (handle) of device-specific information. Typically this is a pointer to a private data structure that may contain information such as the address of the hardware, state information, and other information that may be shared between drivers. The Workstation Subsystem allows separate handles for the cursor, colormap, and the screen functions.

depth
: Specifies a pointer to the `ws_depth_descriptor` structure, which describes the attributes of the screens depth.

screen
: Specifies a pointer to the `ws_screen_descriptor` structure, which describes the attributes of the screen.

mc
: Specifies a pointer to the `/usr/sys/include/sys/workstation.h` structure, which controls the depth mapping.

DATA STRUCTURES

The `map_unmap_screen` function accesses the `ws_depth_descriptor`, `ws_screen_descriptor`, and `ws_map_control` structures defined in `/usr/sys/include/sys/workstation.h`.

DESCRIPTION

The `map_unmap_screen` function creates the memory map for the device specified by **screen*. That is, it gets the user address for the frame buffer and hardware registers. When the function exits, the `pixmap` member of the `ws_depth_descriptor` structure contains the user-space address of the frame buffer, and the `plane_mask` member of the `ws_depth_descriptor` structure contains the user-space address of the I/O registers.

The `map_unmap_screen` function must convert the I/O handle to a physical address, then convert the physical address to a kernel unmapped virtual address (kseg). The function passes the kseg to the `ws_map_region` routine, which performs the memory mapping.

RETURN VALUES

On success, the `map_unmap_screen` function returns 0 (zero). If it cannot map the frame buffers or registers, the function returns `ENOMEM`.

EXAMPLES

The following example shows how the myvga device driver implements the `map_unmap_screen` function:

```
int
myvga_map_unmap_screen(caddr_t screen_handle,
                       ws_depth_descriptor *depths,
                       ws_screen_descriptor *screen,
                       ws_map_control *mp)
{
    register struct myvga_type *scp =
            (struct myvga_type *)screen_handle;
    register ws_depth_descriptor *dp;
    io_handle_t handle;
    caddr_t temp;
    int nbytes;
    register struct controller *ctlr =
            myvgainfo[scp->unit];

    if (myvga_developer_debug)
       printf("myvga_map_unmap_screen: entry\n");

    /* Unmapping is not supported.) */
```

map_unmap_screen

```
if (mp->map_unmap == UNMAP_SCREEN)
    return (EINVAL);

/*
 * Do the mapping only once per screen.
 * Otherwise, assume the information is
 * available, and return.
 */

if ((IS_MYVGA_MAPPED(scp)) && (scp->mapped_pid ==
        u.u_procp->p_pid))
{
    return(0);
}

dp = depths + mp->which_depth;

/*
 * Convert the I/O handle to a physical address,
 * then convert the physical address to a kseg.
 */

handle = scp->mem_handle;
temp = (caddr_t) iohandle_to_phys(handle, IOH_SPARSE_BYTE);
temp = (caddr_t) PHYS_TO_KSEG(temp);

/*
 * Calculate the size of the frame buffer by
 * subtracting the base address from the high
 * address of the frame buffer.
 */

nbytes = iohandle_to_phys(handle + HIGHMAP_SIZE,
            IOH_SPARSE_BYTE) -
             iohandle_to_phys(handle + 0,
                              IOH_SPARSE_BYTE);

/* Call ws_map_region to map the frame buffer in
 * user space, and place the address of the memory
 * map in the ws_depth_descriptor structure.
 */

dp->pixmap = ws_map_region(temp, NULL, nbytes,
                           0600, (int *)NULL);
```

```
if (myvga_developer_debug)
   printf("myvga_map_unmap_screen: fb: ");
   printf("nbytes 0x%x handle 0x%lx kseg 0x%lx virt 0x%lx\n",
          nbytes, handle, temp, dp->pixmap);

if (dp->pixmap == (caddr_t) NULL)
    return(ENOMEM);

/*
 * Perform the same conversions for the I/O registers --
 * from I/O handle to physical address to KSEG.
 */

handle = busphys_to_iohandle(IOREGS_BASE, BUS_IO, ctlr);
temp = (caddr_t) iohandle_to_phys(handle, IOH_SPARSE_BYTE);
temp = (caddr_t) PHYS_TO_KSEG(temp);

/*
 * Calculate the size of the register space.
 */

nbytes = iohandle_to_phys(handle + IOREGS_SIZE,
            IOH_SPARSE_BYTE) -
             iohandle_to_phys(handle + 0,
                              IOH_SPARSE_BYTE);

/*
 * Call ws_map_region to map the registers in
 * user space, and place the address of the memory
 * map in the ws_depth_descriptor structure.
 */

dp->plane_mask = ws_map_region(temp, NULL, nbytes,
                 0600, (int *)NULL);

if (myvga_developer_debug)
   printf("myvga_map_unmap_screen: regs: ");
   printf("nbytes 0x%x handle 0x%lx kseg 0x%lx virt 0x%lx\n",
          nbytes, handle, temp, dp->plane_mask);

if (dp->plane_mask == (caddr_t)NULL)
    return(ENOMEM);

SET_MYVGA_MAPPED(scp);
scp->mapped_pid = u.u_procp->p_pid;
```

```
    return (0);
}
```

RELATED INFORMATION

Appendix A, Data Structures: `ws_depth_descriptor`, `ws_map_control`, `ws_screen_descriptor`, `ws_screen_functions`

Appendix C, Driver Routines: `close`, `init_screen`, `init_screen_handle`, `ioctl`

Files: `/usr/sys/include/sys/workstation.h`, `/usr/sys/include/sys/wsdevice.h`

NAME

`recolor_cursor` – Changes the foreground and background colors of the cursor

SYNOPSIS

```
int (*ws_color_map_functions->recolor_cursor)(
        caddr_t cursor_handle,
        ws_screen_descriptor *screen,
        ws_color_cell *foreground,
        ws_color_cell *background);
```

ARGUMENTS

cursor_handle
: Specifies the virtual address (handle) of device-specific information. Typically this is a pointer to a private data structure that may contain information such as the address of the hardware, state information, and other information that may be shared between drivers. The Workstation Subsystem allows separate handles for the cursor, colormap, and the screen functions.

screen
: Specifies a pointer to the `ws_screen_descriptor` structure, which describes the attributes of the screen.

foreground
: Specifies a pointer to the `ws_color_cell` structure, which defines the foreground color for the cursor specified by *cursor_handle*.

background
: Specifies a pointer to the `ws_color_cell` structure, which defines the background color for the cursor specified by *cursor_handle*.

DATA STRUCTURES

The `recolor_cursor` function accesses the `ws_screen_descriptor` and `ws_color_cell` structures defined in `/usr/sys/include/sys/workstation.h`.

recolor_cursor

DESCRIPTION

The `recolor_cursor` function recolors the foreground and the background of the cursor specified by **foreground* and **background*, respectively.

RETURN VALUES

If successful, the `recolor_cursor` function returns 0 (zero). If unsuccessful, it returns –1.

NOTES

This function is necessary on both gray scale and black-and-white monitors.

EXAMPLES

The following example shows how the `recolor_cursor` function is written for an ATI Mach64 graphics adapter:

```
ati64_recolor_cursor(caddr_t cursor_handle,
        ws_screen_descriptor screen,
        ws_color_cell fg, bg)
{
    register struct vga_info *vp =
     (struct vga_info *)cursor_handle;

    vp->cursor_fg = *fg;
    vp->cursor_bg = *bg;

    ati64_restore_cursor_color(cursor_handle);

    return(0);
}

ati64_restore_cursor_color(caddr_t cursor_handle)
{
    register struct vga_info *vp =
            (struct vga_info *)cursor_handle;
    register struct ati64_type *ap =
            &ati64_info[vp->unit];
    ws_color_cell entry;
    int rshift = 24, gshift = 16, bshift = 8;
```

```
    unsigned char cur_clr0_b, cur_clr0_g, cur_clr0_r;
    unsigned char cur_clr1_b, cur_clr1_g, cur_clr1_r;
    unsigned int cur_clr0, cur_clr1;

    /* Set up new color. */
    cur_clr0_r = (vp->cursor_bg.red >> bshift);
    cur_clr0_g = (vp->cursor_bg.green >> bshift);
    cur_clr0_b = (vp->cursor_bg.blue >> bshift);
    cur_clr0 = ( (cur_clr0_r << rshift) |
            (cur_clr0_g << gshift) |
            (cur_clr0_b << bshift) );

    cur_clr1_r = (vp->cursor_fg.red >> bshift);
    cur_clr1_g = (vp->cursor_fg.green >> bshift);
    cur_clr1_b = (vp->cursor_fg.blue >> bshift);
    cur_clr1 = ( (cur_clr1_r << rshift) |
            (cur_clr1_g << gshift) |
            (cur_clr1_b << bshift) );

    /* Update cursor color with write to chip. */
    REGW(vp, ap, CUR_CLR0, cur_clr0);
    REGW(vp, ap, CUR_CLR1, cur_clr1);
    break;

    return(0);
}
```

RELATED INFORMATION

Appendix A, Data Structures: `ws_color_cell`, `ws_cursor_functions`, `ws_screen_descriptor`

Appendix C, Driver Routines: `cursor_on_off`, `init_cursor_handle`, `load_cursor`, `set_cursor_position`

Files: `/usr/sys/include/sys/workstation.h`, `/usr/sys/include/sys/wsdevice.h`

NAME

`set_cursor_position` – Moves the cursor hotspot to a location on the screen

SYNOPSIS

```
int (*ws_cursor_functions->set_cursor_position)(
        caddr_t cursor_handle,
        ws_screen_descriptor *screen,
        int x,y);
```

ARGUMENTS

cursor_handle
: Specifies the virtual address (handle) of device-specific information. Typically this is a pointer to a private data structure that may contain information such as the address of the hardware, state information, and other information that may be shared between drivers. The Workstation Subsystem allows separate handles for the cursor, colormap, and the screen functions.

screen
: Specifies a pointer to the `ws_screen_descriptor` structure, which describes the attributes of the screen.

x, y
: Specify the absolute position of the cursor's hotspot within the screen's resolution.

DATA STRUCTURES

The `set_cursor_position` function accesses the `ws_screen_descriptor` structure defined in `/usr/sys/include/sys/workstation.h`.

DESCRIPTION

The `set_cursor_position` function moves the cursor hotspot to a location on the screen specified by *x, y*. The function must deal with any offsets required for video or the cursor's hotspot. In addition, the

Workstation Subsystem guarantees that the hotspot will be confined to the screen and cursor handles by updating the screen coordinates in the screen structure after this function has been called.

RETURN VALUES

On success, the set_cursor_position function returns 0 (zero). If an error occurs, it returns –1.

EXAMPLES

The following example shows how the set_cursor_position function is written for an ATI Mach64 graphics adapter:

```
ati64_set_cursor_position(caddr_t cursor_handle,
        ws_screen_descriptor sp,
        int x,
        int y)
{
    register struct vga_info *vp =
            (struct vga_info *)cursor_handle;
    register struct ati64_type *ap =
            &ati64_info[vp->unit];
    register int xt, yt, xo, yo, offset, temp, viol;

    /* Initialized the first time only */
    static int previol = 0;


    /*
     * Bias position by where the hotspot is.
     * Want to move hot spot to new location x.
     */
    xt = x - vp->x_hot;
    yt = y - vp->y_hot;

    /*
     * Location in memory where cursor
     * definition area (64x64) begins
     */
    offset = vp->cursor_offset;
    viol = 0;

    /* Cursor display will begin at xo (origin) */
```

```
if (xt < 0) {
    xo = -xt;
    xt = 0;
    viol = 1;
} else
    xo = 0;

/*
 * If yt < 0, cursor y_hot is to move down.  For each
 * y0 unit moved, add 1 scanline to location in memory
 * where cursor position begins.  A scanline is 64
 * pixels (64*2 bits=16 bytes)
 */

/*
 * For each y0 unit, add 1 scanline
 * a scanline is 64*2 bits=16 bytes
 */
if (yt < 0) {
    yo = -yt;
    yt = 0;
    offset += (yo << 4);
    viol = 1;
} else
    yo = 0;

/*
 * The following code keeps the cursor POSTION
 *  up-to-date, even while the cursor is off,
 *  so that code (ati64_cursor_on_off) that
 *  simply turns the cursor on will find it at
 *  the desired spot.
 */

if (viol || previol) {
    temp = REGR(vp, ap, GEN_TEST_CNTL);
  /* Turn OFF cursor. */
  REGW(vp, ap, GEN_TEST_CNTL, temp & ~0x80);
  /* Offset is QWORD index. */
  REGWZ(vp, ap, CUR_OFFSET, offset >> 3);
  REGWZ(vp, ap, CUR_HORZ_VERT_OFF, (yo << 16) | xo);
    mb();
  /* Turn ON cursor. */
  REGW(vp, ap, GEN_TEST_CNTL, temp | 0x80);
}
```

```
    previol = viol;

    /* Write cursor position to chip. */
    REGW(vp, ap, CUR_HORZ_VERT_POSN, (yt << 16) | xt);

    return(0);
}
```

RELATED INFORMATION

Appendix A, Data Structures: `ws_screen_descriptor`, `ws_cursor_functions`

Appendix C, Driver Routines: `cursor_on_off`, `init_cursor_handle`, `load_cursor`, `recolor_cursor`

Files: `/usr/sys/include/sys/workstation.h`, `/usr/sys/include/sys/wsdevice.h`

video_on, video_off

NAME

video_on, video_off – Turn video on and off

SYNOPSIS

```
int (*ws_color_map_functions->video_on)(
        caddr_t colormap_handle,
int (*ws_color_map_functions->video_off)(
        caddr_t colormap_handle);
```

ARGUMENTS

colormap_handle
: Specifies the virtual address (handle) of device-specific information. Typically this is a pointer to a private data structure that may contain information such as the address of the hardware, state information, and other information that may be shared between drivers. The Workstation Subsystem allows separate handles for the cursor, colormap, and the screen functions.

DESCRIPTION

The video_on function turns the video on and the video_off function turns the video off, which may involve such things as saving and restoring a number of colormap entries for the cursor and X clients as well as accessing the graphics hardware specified by the *colormap_handle* parameter.

RETURN VALUES

If successful, the video_on and video_off functions return 0 (zero). If unsuccessful, they return –1.

EXAMPLES

The following example shows how the myvga example driver implements the video_on and video_off functions:

```
int
myvga_video_on(caddr_t colormap_handle)
{
```

```
   register struct myvga_type *scp =
      (struct myvga_type *)colormap_handle;
   unsigned char state;

   OUTB(MYVGA_SEQ_ADDRESS, 0x01);
   state = INB(MYVGA_SEQ_DATA);

   state &= 0xDF;

   /*
    * Turn on video in Clocking Mode register.
    */
   OUTB(MYVGA_SEQ_ADDRESS, 0x01);
   OUTB(MYVGA_SEQ_DATA, state); /* change mode */

   return(0);
}

int
myvga_video_off(caddr_t colormap_handle)
{
   register struct myvga_type *scp =
      (struct myvga_type *)colormap_handle;
   unsigned char state;

   OUTB(MYVGA_SEQ_ADDRESS, 0x01);
   state = INB(MYVGA_SEQ_DATA);

   state |= 0x20;

   /*
    * Turn off video in Clocking Mode register.
    */
   OUTB(MYVGA_SEQ_ADDRESS, 0x01);
   OUTB(MYVGA_SEQ_DATA, state); /* change mode */

   return(0);
}
```

RELATED INFORMATION

Appendix A, Data Structures: `ws_color_map_functions`

Appendix C, Driver Routines: `clean_color_map`, `init_color_map`, `init_colormap_handle`, `load_color_map_entry`

video_on, video_off

Files: /usr/sys/include/sys/workstation.h,
/usr/sys/include/sys/wsdevice.h

NAME

ws_get_screen – Returns a pointer to the screen descriptor for a screen

SYNOPSIS

```
ws_screens *ws_get_screen(
          short screen_number);
```

ARGUMENTS

screen_number
: Specifies the number of the screen whose screen descriptor is to be returned.

DATA STRUCTURES

The ws_get_screen routine accesses the ws_screens structure defined in /usr/sys/include/sys/wsdevice.h.

DESCRIPTION

The ws_get_screen routine returns a pointer to the screen descriptor for the screen specified by the *screen_number* argument.

RETURN VALUES

If successful, the ws_get_screen routine returns the screen descriptor associated with the specified *screen_number*. If no screen descriptor is found, ws_get_screen returns NULL.

RELATED INFORMATION

Appendix A, Data Structures: ws_screens

Appendix C, Driver Routines: install_vga_console, ws_is_mouse_on, ws_map_region, ws_register_screen

Files: /usr/sys/include/sys/wsdevice.h

ws_is_mouse_on

NAME

ws_is_mouse_on – Determines whether the server has opened /dev/ws0

SYNOPSIS

```
int ws_is_mouse_on(
        );
```

ARGUMENTS

None

DESCRIPTION

The ws_is_mouse_on routine is a convenience routine that determines whether the server is running. You can use this routine to determine if you are at the console.

RETURN VALUES

If the server is running, the ws_is_mouse_on routine returns 1. If the server is not running, it returns 0.

EXAMPLES

The following example shows how to use ws_is_mouse_on to exit from a routine if the server is running:

```
/* Exit only if in text mode */
if (ws_is_mouse_on())
    return(0);
```

RELATED INFORMATION

Appendix C, Driver Routines: install_vga_console, ws_get_screen, ws_map_region, ws_register_screen

NAME

ws_map_region – Maps kernel addresses into the process virtual address space

SYNOPSIS

```
caddr_t ws_map_region(
          caddr_t kaddr,
          caddr_t uaddr,
          register int nbytes,
          int how,
          int *erroraddr);
```

ARGUMENTS

kaddr
Specifies the kernel address of the memory to be mapped.

uaddr
Specifies the address of the current process where *kaddr* is to be mapped. This argument should be NULL unless you need to map kernel addresses to a specific location in the process' address

nbytes
Specifies the size (in bytes) of the memory to be mapped.

how
Specifies the access mode in octal for the physical memory segment.

erroraddr
Specifies a pointer to the error number (errno) address, which returns NULL if the memory cannot be mapped.

DESCRIPTION

The ws_map_region routine takes the physical addresses that *kaddr* specifies and maps them into the current process's virtual address space. This routine is typcially called by map_unmap_screen to map the physical address of the defined screen into virtual address space.

NOTES

Each call to this routine allocates one memory segment. Since shared memory segments are expensive system resources, you should minimize the number of separate areas mapped.

RETURN VALUES

On success, the ws_map_region routine returns the address of the memory map. If the memory could not be mapped, it returns a NULL pointer.

EXAMPLES

The following example shows how the myvga_map_unmap_screen function calls ws_map_region to map I/O registers to user space:

```
int
myvga_map_unmap_screen(caddr_t screen_handle,
                       ws_depth_descriptor *depths,
                       ws_screen_descriptor *screen,
                       ws_map_control *mp)
{
.
.
.
    dp->pixmap = ws_map_region(temp, NULL, nbytes,
            0600, (int *)NULL);
.
.
.
}
```

RELATED INFORMATION

Appendix C, Driver Routines: map_unmap_screen

Files: /usr/sys/include/sys/workstation.h, /usr/sys/include/sys/wsdevice.h

NAME

`ws_register_screen` – Returns the index of the screen as it is used by the Workstation Subsystem

SYNOPSIS

```
int ws_register_screen(
        ws_screen_descriptor *sp,
        ws_visual_descriptor *vp,
        ws_depth_descriptor *dp,
        ws_screen_functions *f,
        ws_color_map_functions *cmf,
        ws_cursor_functions *cf,
        controller *ctlr);
```

ARGUMENTS

sp
Specifies a pointer to the `ws_screen_descriptor` structure, which describes the attributes of the screen.

vp
Specifies a pointer to the array of `ws_visual_descriptor` structures, which describe the visual classes supported by the screen.

dp
Specifies a pointer to the array of `ws_depth_descriptor` structures, which describe the depths that are present on the screen.

f
Specifies a pointer to the `ws_screen_functions` structure, which contains functions to manipulate the screen structure.

cmf
Specifies a pointer to the `ws_color_map_functions` structure, which contains functions to manipulate the colormap structure.

cf
Specifies a pointer to the `ws_cursor_functions` structure, which contains functions to manipulate the cursor structure.

ctlr
Specifies a pointer to the driver's `controller` structure.

DATA STRUCTURES

The ws_register_screen routine accesses the ws_screen_descriptor, ws_depth_descriptor, ws_visual_descriptor, ws_cursor_functions, ws_screen_functions, and ws_color_map_function structures defined in /usr/sys/include/sys/workstation.h and /usr/sys/include/sys/wsdevice.h.

DESCRIPTION

The ws_register_screen routine registers a display screen with the Workstation Subsystem and returns the ID of the screen as it is used by the Workstation Subsystem. (This ID is not necessarily the same as the screen number assigned to the screen by the server.)

You call ws_register_screen from your graphics driver during autoconfiguration in your driver's probe routine.

RETURN VALUES

On success, the ws_register_screen routine returns the screen number for the display device. It returns NULL if an error occurs.

EXAMPLES

The following example shows how the myvga example device calls the ws_register_screen routine in its probe routine to make the driver known to the Workstation Subsystem:

```
int
myvga_probe(vm_offset_t addr,
     register struct controller *ctlr)
{
⋮
  /*
   *  Initialize function handles
   */
   scp->sf.screen_handle = (*(scp->sf.init_screen_handle))
      ((caddr_t)myvga_softc,ctlr->physaddr,ctlr->ctlr_num,0);
   scp->cf.cursor_handle = (*(scp->cf.init_cursor_handle))
      ((caddr_t)myvga_softc, ctlr->physaddr, ctlr->ctlr_num, 0);
```

```
   scp->cmf.colormap_handle = (*(scp->cmf.init_colormap_handle))
      ((caddr_t)myvga_softc,ctlr->physaddr,ctlr->ctlr_num,0);

   status = ws_register_screen(&scp->screen,
                               scp->visual, scp->depth,
                               &scp->sf, &scp->cmf,
                               &scp->cf, ctlr);
   if (status == -1) {
      printf("myvga driver: could not register screen\n");
      FREE(scp,M_DEVBUF);
      return 0;
   }
.
.
.
}
```

RELATED INFORMATION

Appendix A, Data Structures: `ws_color_map_functions`, `ws_cursor_functions`, `ws_depth_descriptor`, `ws_screen_functions`, `ws_screen_descriptor`, `ws_visual_descriptor`

Appendix C, Driver Routines: `install_vga_console`, `ws_get_screen`, `ws_is_mouse_on`, `ws_map_region`

Files: `/usr/sys/include/sys/workstation.h`, `/usr/sys/include/sys/wsdevice.h`

D

DDX Loadable Services Routines

This appendix contains reference pages for the routines that interface to the DIGITAL UNIX X Server loadable subsystem. The DDX can call the loadable services routines, for example, to load and unload libraries or to check the options that the user has specified on the command line.

Routine descriptions contain the following sections:

Name

This section lists the name of the routine along with a summary of its purpose.

Synopsis

This section contains the function prototype, which gives you the following information:

- Return type

  The data type of the return value, in bold font, or `void` if the routine does not return a value.

- Routine name

  The name of the routine, in bold font. Note that routine names are case sensitive.

- Argument data type

  The argument's C type definition. Data type keywords are in bold font.

- Argument name

  The argument name, in italic font.

Arguments

This section contains a description of each argument.

Description

This section explains the tasks that the routine performs.

Return value

This section shows the return values that the routine can return, or “None” if no value is returned. If the routine returns an error value, this value is also described in the Return Value section.

Notes

This section discusses information that falls into the following categories:

- Hardware-specific information

  Some interfaces behave differently depending on the architecture of the hardware.

- Operating system-specific information

  Some interfaces behave differently depending on the implementation of the operating system.

- Information pertinent to device drivers

  Some interfaces require specific information important to the device driver writer.

Related Information

This section lists related structures and routines. You can refer to the reference pages for these structure and routines for additional information.

NAME

LS_ForceSymbolResolution – Forces all symbols for a library to be resolved

SYNTAX

```
LS_Status LS_ForceSymbolResolution(
          );
```

ARGUMENTS

None

DESCRIPTION

The LS_ForceSymbolResolution routine forces all symbols for a loaded library to be resolved by opening the server library and all libraries that depend upon it. It calls the dlopen function with the RTLD_NOW mode so that the loader resolves all symbolic references in those libraries.

RETURN VALUE

On success, the LS_ForceSymbolResolution routine returns LS_Success. If an error occurs, it returns LS_Failure.

RELATED INFORMATION

Appendix D, Loadable Services Routines: LS_GetSymbol, LS_GetSymbolInLibrary

LS_FreeMarkedLibraries

NAME

LS_FreeMarkedLibraries – Closes all libraries marked for unloading

SYNTAX

```
void LS_FreeMarkedLibraries(
          );
```

ARGUMENTS

None

DESCRIPTION

The LS_FreeMarkedLibraries routine closes all libraries that were previously marked by the LS_MarkForUnloadLibraryReqs routine.

RETURN VALUE

None

RELATED INFORMATION

Appendix D, Loadable Services Routines:
LS_MarkForUnloadLibraryReqs

NAME

LS_GetDeviceName – Returns a pointer to the device name

SYNTAX

```
char * LS_GetDeviceName(
          LS_LibraryReq *libraries,
          int index);
```

ARGUMENTS

libraries
Specifies a set of library records. This list may specify individual libraries or a range of libraries.

index
Specifies the starting index into the list of library records.

DESCRIPTION

The LS_GetDeviceName routine accesses the list of libraries specified in *libraries* and returns a pointer to the device name at the specified *index*.

RETURN VALUE

The LS_GetDeviceName routine returns a pointer to the device name.

RELATED INFORMATION

Appendix A, Data Structures: LS_LibraryReqs

Appendix D, Loadable Services Routines: LS_GetInitProc, LS_GetInitProcName, LS_GetLibFileName, LS_GetLibName, LS_GetLibraryReqBy

NAME

LS_GetInitProc – Returns the address of the library's initialization procedure

SYNTAX

```
LS_InitProcPtr LS_GetInitProc(
        LS_LibraryReq *libraries,
        int index);
```

ARGUMENTS

libraries
Specifies a set of library records. This list may specify individual libraries or a range of libraries.

index
Specifies the starting index into the list of library records.

DESCRIPTION

The LS_GetInitProc routine accesses the library at the specified *index* in the list, *libraries*, and returns the address of the initialization procedure for the library.

RETURN VALUE

The LS_GetInitProc routine returns the address of the initialization procedure.

RELATED INFORMATION

Appendix A, Data Structures: LS_InitProcPtr, LS_LibraryReq

Appendix D, Loadable Services Routines: LS_GetDeviceName, LS_GetInitProcName, LS_GetLibFileName, LS_GetLibName, LS_GetLibraryReqBy

NAME

LS_GetInitProcName – Returns the name of the library's initialization procedure

SYNTAX

```
LS_InitProcPtr LS_GetInitProcName(
          char *libraries,
          int index);
```

ARGUMENTS

libraries
Specifies a set of library records. This list may specify individual libraries or a range of libraries.

index
Specifies the starting index into the list of library records.

DESCRIPTION

The LS_GetInitProcName routine accesses the library at the specified *index* in the list, *libraries*, and returns the name of the initialization procedure for the library.

RETURN VALUE

The LS_GetInitProcName routine returns the name of the initialization procedure.

RELATED INFORMATION

Appendix A, Data Structures: LS_InitProcPtr, LS_LibraryReq

Appendix D, Loadable Services Routines: LS_GetDeviceName, LS_GetInitProc, LS_GetLibFileName, LS_GetLibName, LS_GetLibraryReqBy

LS_GetLibFileName

NAME

LS_GetLibFileName – Returns a pointer to the library file name

SYNTAX

```
char * LS_GetLibFileName(
          LS_LibraryReq *libraries,
          int index);
```

ARGUMENTS

libraries
Specifies a set of library records. This list may specify individual libraries or a range of libraries.

index
Specifies the starting index into the list of library records.

DESCRIPTION

The LS_GetLibFileName routine accesses the list of libraries specified in *libraries* and returns a pointer to the library file name at the specified *index*.

RETURN VALUE

The LS_GetLibFileName routine returns a pointer to the library file name.

RELATED INFORMATION

Appendix A, Data Structures: LS_LibraryReq

Appendix D, Loadable Services Routines: LS_GetDeviceName, LS_GetInitProc, LS_GetInitProcName, LS_GetLibName

NAME

LS_GetLibName – Returns a pointer to the library name

SYNTAX

```
char *LS_GetLibName(
          char *libraries,
          int index);
```

ARGUMENTS

libraries
Specifies a set of library records. This list may specify individual libraries or a range of libraries.

index
Specifies the starting index into the list of library records.

DESCRIPTION

The LS_GetLibName routine accesses the list of libraries specified in *libraries* and returns a pointer to the library name at the specified *index*.

RETURN VALUE

The LS_GetLibName routine returns a pointer to the library name.

RELATED INFORMATION

Appendix A, Data Structures: LS_LibraryReq

Appendix D, Loadable Services Routines: LS_GetDeviceName, LS_GetInitProc, LS_GetInitProcName, LS_GetLibFileName

LS_GetLibraryReqByDeviceName

NAME

LS_GetLibraryReqByDeviceName – Returns the index of a library name within a library list

SYNTAX

```
int LS_GetLibraryReqByDeviceName(
        LS_LibraryReq *libraries,
        int count,
        char *name);
```

ARGUMENTS

libraries
Specifies a set of library records. This list may specify individual libraries or a range of libraries.

count
Specifies the number of libraries in the list.

name
Specifies the name of the device to search for in the list.

DESCRIPTION

The LS_GetLibraryReqByDeviceName routine searches the specified libraries in a list of library records and returns the index of the library with the specified device name.

RETURN VALUE

On success, the LS_GetLibraryReqByDeviceName routine returns the index of the library. If the library is not in the list, it returns –1.

RELATED INFORMATION

Appendix A, Data Structures: LS_LibraryReq

Appendix D, Loadable Services Routines: LS_GetLibraryReqByExtensionName, LS_GetLibraryReqByLibName

LS_GetLibraryReqByExtensionName

NAME

LS_GetLibraryReqByExtensionName – Returns the index of a library within a library list

SYNTAX

```
int LS_GetLibraryReqByExtensionName(
        LS_LibraryReq *libraries,
        int count,
        char *name);
```

ARGUMENTS

libraries
Specifies a set of library records. This list may specify individual libraries or a range of libraries.

count
Specifies the number of libraries in the list.

name
Specifies the name of the extension to search for in the list.

DESCRIPTION

The LS_GetLibraryReqByExtensionName routine searches the specified libraries in a list of library records and returns the index of the library with the specified extension name.

RETURN VALUE

On success, the LS_GetLibraryByExtensionName routine returns the index of the library. If the library is not in the list, it returns –1.

RELATED INFORMATION

Appendix A, Data Structures: LS_LibraryReq

Appendix D, Loadable Services Routines: LS_GetLibraryReqByDeviceName, LS_GetLibraryReqByLibName

LS_GetLibraryReqByLibName

NAME

LS_GetLibraryReqByLibName – Returns the index of a library within a library list

SYNTAX

```
int LS_GetLibraryReqByLibName(
        LS_LibraryReq *libraries,
        int count,
        char *name);
```

ARGUMENTS

libraries
Specifies a set of library records. This list may specify individual libraries or a range of libraries.

count
Specifies the number of libraries in the list.

name
Specifies the name of the library to search for in the list.

DESCRIPTION

The LS_GetLibraryReqByLibName routine searches the specified libraries in a list of library records and returns the index of the library with the specified name.

RETURN VALUE

On success, the LS_GetLibraryReqByLibName routine returns the index of the named library. If the library is not in the list, it returns –1.

RELATED INFORMATION

Appendix A, Data Structures: LS_LibraryReq

Appendix D, Loadable Services Routines:
LS_GetLibraryReqByDeviceName,
LS_GetLibraryReqByExtensionName

NAME

LS_GetSubLibList – Returns information about a library's sublibrary list

SYNTAX

```
Boolean LS_GetSubLibList(
          LS_LibraryReq *libraries,
          int index,
          LS_LibraryReq **subliblist,
          int *count);
```

ARGUMENTS

libraries
: Specifies a set of library records. This list may specify individual libraries or a range of libraries.

index
: Specifies the starting index into the list of library records.

subliblist
: Contains a pointer to a pointer to the list of sublibraries associated with the specified library. This argument value is returned by the routine.

count
: Contains the number of libraries in the *subliblist*. The routine returns this argument value.

DESCRIPTION

The LS_GetSubLibList routine finds the library at the specified *index* offset from the beginning of *libraries* and initializes a pointer to the list of the library's sublibraries and the number of libraries in the list.

A sublibrary is a library that depends on another library. In the Xserver.conf file, for example, all VGA-compliant DDXs are sublibraries to the VGA library. The server must load the VGA library before loading its sublibraries.

RETURN VALUE

When the sublibrary list is greater than 0 (zero), the `LS_GetSubLibList` routine returns `TRUE`. If the library has no sublibraries, it returns `FALSE`.

RELATED INFORMATION

Appendix A, Data Structures: `LS_LibraryReq`

NAME

LS_GetSymbol – LS_GetSymbol

SYNTAX

```
void *LS_GetSymbol(
          char *symbolName);
```

ARGUMENTS

symbolName
Specifies the name of the symbol in global name space.

DESCRIPTION

The LS_GetSymbol routine finds the symbol in the global name space and returns a pointer to its address.

RETURN VALUE

The LS_GetSymbol routine returns the address of the symbol, or a NULL pointer if the symbol is not found.

RELATED INFORMATION

Appendix D, Loadable Services Routines: LS_ForceSymbolResolution, LS_GetSymbolInLibrary

LS_GetSymbolInLibrary

NAME

LS_GetSymbolInLibrary – Returns the address of a symbol in the name space of a library

SYNTAX

```
void *LS_GetSymbolInLibrary(
        char *symbolName);
```

ARGUMENTS

symbolName
Specifies the name of the symbol in the library name space.

DESCRIPTION

The LS_GetSymbolInLibrary routine finds the symbol in the library name space and returns a pointer to its address.

RETURN VALUE

The LS_GetSymbolInLibrary routine returns the address of the symbol, or a NULL pointer if the symbol is not found.

RELATED INFORMATION

Appendix D, Loadable Services Routines: LS_ForceSymbolResolution, LS_GetSymbol

NAME

LS_GetVideoMode – Determines the video mode for the screen

SYNTAX

```
short LS_GetVideoMode(
          ScreenPtr pScreen,
          ValidModePtr pModes);
```

ARGUMENTS

pScreen
Specifies a pointer to the ScreenRec structure for the screen.

pModes
Specifies the valid modes for the screen.

DESCRIPTION

The LS_GetVideoMode routine determines the video modes that the screen should use, based on any command line arguments that the user specified and on the valid modes for the screen. The routine returns the valid video mode that most closely matches the modes that the user requested. If the user does not specify any command line arguments, the routine returns the default video mode.

RETURN VALUE

The LS_GetVideoMode routine returns an index into the list of valid video modes, specifying the video mode that the screen should use.

LS_IsLibraryInited

NAME

LS_IsLibraryInited – Checks to see if a library is initialized

SYNTAX

```
Boolean LS_IsLibraryInited(
          LS_LibraryReq *libraries,
          int index);
```

ARGUMENTS

libraries
Specifies a set of library records. This list may specify individual libraries or a range of libraries.

index
Specifies the starting index into the list of library records.

DESCRIPTION

The LS_IsLibraryInited routine checks to see if the specified library is initialized.

RETURN VALUE

If the library is initialized, the LS_IsLibraryInited routine returns TRUE. If the library is not initialized, it returns FALSE.

RELATED INFORMATION

Appendix A, Data Structures: LS_LibraryReq

Appendix D, Loadable Services Routines: LS_MarkLibraryInited

NAME

`LS_ListOpenLibraries` – Displays a list of all opened libraries

SYNTAX

```
void LS_ListOpenLibraries(
         );
```

ARGUMENTS

None

DESCRIPTION

The `LS_ListOpenLibraries` sends to `stderr` all of the opened libraries.

RETURN VALUE

None

RELATED INFORMATION

Appendix D, Loadable Services Routines: `LS_LoadLibraryReqs`

LS_LoadLibraryReqs

NAME

LS_LoadLibraryReqs – Loads a set of libraries into the loadable server

SYNTAX

```
LS_Status LS_LoadLibraryReqs(
          LS_LibraryReq *libraries,
          int index,
          int number,
          Boolean enforceVersion);
```

ARGUMENTS

libraries
Specifies a set of library records for the libraries to be loaded. This list may specify individual libraries or a range of libraries.

index
Specifies the starting index into the list of library records.

number
Specifies the number of libraries to load from the list, starting at *index*.

enforceVersion
Specifies TRUE or FALSE to indicate whether the routine should set the library version number in the library record. The DIGITAL UNIX X Server uses this argument when it needs to distinguish between major versions.

DESCRIPTION

The LS_LoadLibraryReqs routine searches the specified libraries in a list of library records and opens any libraries that are not already opened. It calls the dlopen function with the RTLD_LAZY mode so that the loader does symbol resolution only as needed.

RETURN VALUE

On success, the routine returns LS_Success. If it encounters a library it cannot load, it returns LS_Failure.

RELATED INFORMATION

Appendix A, Data Structures: `LS_LibraryReq`

Appendix D, Loadable Services Routines: `LS_UnLoadLibraryreqs`

LS_MarkForUnloadLibraryReqs

NAME

MarkForUnloadLibraryReqs – Puts a library on a list for unloading at a future time

SYNTAX

```
void
void MarkForUnloadLibraryReqs(
         LS_LibraryReq *libraries,
         int index,
         int number);
```

ARGUMENTS

libraries
Specifies a set of library records for the libraries to be marked for unloading. This list may specify individual libraries or a range of libraries.

index
Specifies the starting index into the list of library records.

number
Specifies the number of libraries to mark for unloading, starting at *index*.

DESCRIPTION

The LS_MarkForUnloadLibraryReqs routine sets an attribute in the master library list to mark the library for unloading. The LS_FreeMarkedLibraries routine searches the master library list and unloads those libraries that have been marked.

The LS_MarkForUnloadLibraryReqs routine marks a library for unloading only if the library's reference count is 0. That is, the library cannot be marked for closing if users are still accessing it.

RETURN VALUE

None

RELATED INFORMATION

Appendix A, Data Structures: `LS_LibraryReq`

Appendix D, Loadable Services Routines: `LS_FreeMarkedLibraries`

LS_ParseArguments

NAME

LS_ParseArguments – Parses the command line argument list for the specified options

SYNTAX

```
void LS_ParseArguments(
        apOptionDescList options,
        int num_options,
        int *argc,
        char **argv);
```

ARGUMENTS

options

Specifies the list of options to search for.

num_options

Specifies the number of options in the list.

argc

Specifies the number of command line arguments in the argument list.

argv

Specifies the address of the argument list.

DESCRIPTION

The LS_ParseArguments routine searches the argument list (*argv*) to see if it contains any of the options in the *options* list and removes those arguments from argv. When the server starts up, it calls this routine to remove the –config, –errfile, –debug, –showDefaults, –showConfigs, and –showUsed options from the argument list. Using LS_ParseArguments ensures that arguments do not remain on the argument list when they are not needed.

RETURN VALUE

None

NAME

LS_UnLoadLibraryReqs – Unloads a set of libraries from the server

SYNTAX

```
void LS_UnLoadLibraryReqs(
        LS_LibraryReq *libraries,
        int index,
        int num);
```

ARGUMENTS

libraries
Specifies a set of library records for the libraries to be unloaded. This list may specify individual libraries or a range of libraries.

index
Specifies the starting index into the list of library records.

num
Specifies the number of libraries to unload from the list, starting at *index*.

DESCRIPTION

The LS_UnLoadLibraryReqs routine searches for the specified libraries in the master library list and closes any libraries that are opened.

RETURN VALUE

None

RELATED INFORAMTION

Appendix A, Data Structures: LS_LibraryReq

Appendix D, Loadable Services Routines: LS_LoadLibraryReqs

E

DIGITAL UNIX X Server Components

This appendix lists the components that make up the DIGITAL UNIX X Server. Table E–1 lists the core components. Table E–2 lists the loadable DDX libraries. Table E–3 lists the loadable extensions.

Note

All directory specifications are relative to `/usr/X11R6/xc/programs`.

Table E–1: Server Loadable Core Components

Name	Core Component
DIX	Xserver/dix/libdix.so
OS	Xserver/os/libos.so
MI	Xserver/mi/libmi.so
MFB	Xserver/mfb/libmfb.so
CFB	Xserver/cfb/libcfb.so
CFB16	Xserver/cfb16/libcfb16.so
CFB32	Xserver/cfb32/libcfb32.so
Workstations	Xserver/hw/dec/ws/lib_dec_ws.so

Table E–2: Loadable DDX Libraries

Name	Core DDX
VGA	Xserver/hw/dec/vga/lib_dec_vga.so
Qvision	Xserver/hw/dec/triton/lib_dec_triton.so
GEN	Xserver/hw/dec/gen/lib_dec_gen.so
TX	Xserver/hw/dec/tx/lib_dec_tx.so

Table E–2: Loadable DDX Libraries (cont.)

Name	Core DDX
ATI	Xserver/hw/dec/ati/lib_dec_ati.so
ATI64	Xserver/hw/dec/ati64/lib_dec_ati64.so
CIRRUS	Xserver/hw/dec/cirrus/lib_dec_cirrus.so
S3	Xserver/hw/dec/S3/lib_dec_S3.so

Table E–3: Loadable Extensions

Name	Extension
BIG-REQUESTS	Xserver/Xext/libextbigreq.so
DOUBLE-BUFFER	Xserver/dbe/libdbe.so
DPMS	Xserver/Xext/libextdpms.so
DEC-XTRAP	Xserver/Xext/libextxtrap.so
MIT-SCREEN-SAVER	Xserver/Xext/libextScrnSvr.so
MIT-SHM	Xserver/Xext/libextshm.so
MIT-SUNDRY-NONSTANDARD	Xserver/Xext/libextMITMisc.so
Multi-Buffering	Xserver/Xext/libextMultibuf.so
SHAPE	Xserver/Xext/libextshape.so
SYNC	Xserver/Xext/libextSync.so
XC-MISC	Xserver/mfb/libextMisc.so
XTEST	Xserver/Xext/libextxtest.so
Keyboard Management Extension	Xserver/Xext/libextkme.so
XKEYBOARD	Xserver/Xext/libxkb.so
Shared Memory Transport	Xserver/dec_smt/lib_dec_smt.so
XInputExtension	Xserver/Xi/libxinput.so
XIE	Xserver/XIE/dixie/libdixie.so Xserver/XIE/mixie/libmixie.so

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825) before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-bps modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	—	Local Digital subsidiary or approved distributor
Internal (submit an Internal Software Order Form, EN-01740-07)	—	SSB Order Processing – NQO/V19 *or* U.S. Software Supply Business Digital Equipment Corporation 10 Cotton Road Nashua, NH 03063-1260

Reader's Comments

DIGITAL UNIX
Writing a Graphics Device Driver and DDX for the DIGITAL UNIX X Server
AA-R5NHA-TE

Digital welcomes your comments and suggestions on this manual. Your input will help us to write documentation that meets your needs. Please send your suggestions using one of the following methods:

- This postage-paid form
- Internet electronic mail: `readers_comment@zk3.dec.com`
- Fax: (603) 881-0120, Attn: UEG Publications, ZK03-3/Y32

If you are not using this form, please be sure you include the name of the document, the page number, and the product name and version.

Please rate this manual:

	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Usability (ability to access information quickly)	☐	☐	☐	☐

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____
_____
_____
_____

What version of the software described by this manual are you using? _____

Name, title, department _____
Mailing address _____
Electronic mail _____
Telephone _____
Date _____

Do Not Cut or Tear − Fold Here and Tape

digital™

NO POSTAGE
NECESSARY IF
MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST−CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
UEG PUBLICATIONS MANAGER
ZKO3−3/Y32
110 SPIT BROOK RD
NASHUA NH 03062−9987

Do Not Cut or Tear − Fold Here

Cut on
Dotted
Line