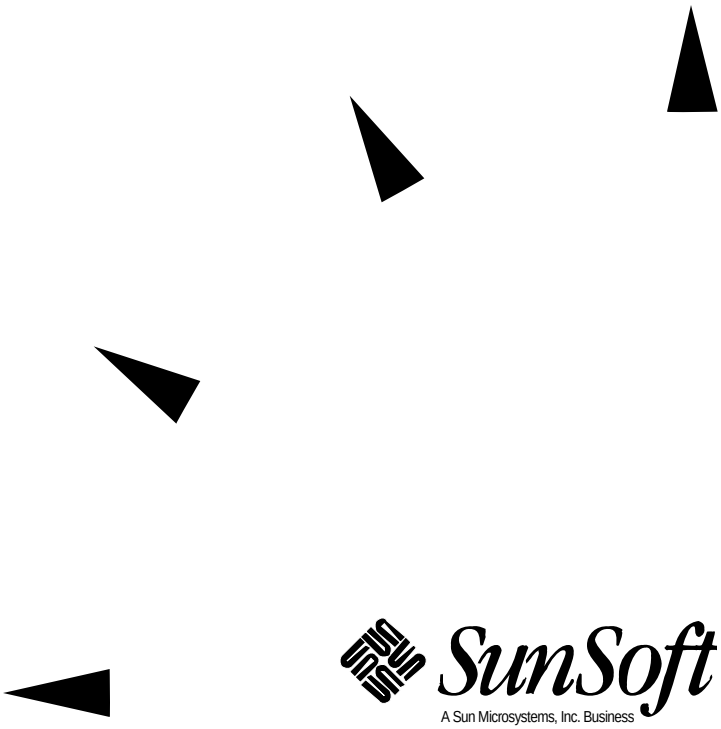
X Server Device Developer's Guide



2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

SunSoft
A Sun Microsystems, Inc. Business

Please
Recycle

Contents

Figures

Tables

Preface

The *X Server Device Developer's Guide* provides detailed information on writing device drivers that run with the OpenWindows™ environment. These device drivers are DDX handlers that interface with the OpenWindows server.

Who Should Use This Book

If you are an Independent Hardware Vendor (IHV) interested in writing device drivers, you should read this book.

Before You Read This Book

Check the *Solaris 2.5.1: Driver Developer Kit Introduction* and *Solaris 2.5.1: Driver Developer Kit Installation Guide* for any corrections or updates to information in this manual.

See Appendix B, "Packaging and Installation Hints" for important information on packaging issues and installation hints.

This manual assumes that the reader has a programming background and familiarity with, or access to, appropriate documentation for:

- Solaris 2.5 and 2.5.1
- The X Window System; specifically the MIT sample server and the DDX (Device Dependent X) porting layer.
- C Language
- X, Xlib

How This Book Is Organized

Chapter 1, “DDX Porting Overview,” provides an overview of porting features and requirements of the DDX layer.

Chapter 2, “The Loadable DDX Interface,” explains how the server interfaces to a loadable DDX handler.

Chapter 3, “Screen Initialization,” describes some aspects of Screen initialization common to many devices.

Chapter 4, “Cursors,” discusses software and hardware cursor implementations and helps you decide which cursor layer to use for your purposes.

Chapter 5, “Multiple Plane Group Interface,” provides an architectural overview and describes the feature of the multiple plane group (MPG) DDX module.

Chapter 6, “Overlay Window Interface,” describes the overlay window interface (OVL) for your DDX handler.

Chapter 7, “Window ID Interface,” defines the window management interface routines that are part of the MPG package.

Chapter 8, “Colormap Interface,” describes all of the routines that are part of the CMAP package. It also provides several examples.

Chapter 9, “Multibuffering Extension to X Interface,” describes how to port your DDX handler to the MBX (Multi-buffering) Extension.

Chapter 10, “Direct Graphics Access Drawable Client Interface,” describes the DGA library interface for clients.

Chapter 11, “Direct Graphics Access Drawable DDX Interface,” describes the DGA library interface for DDX handlers.

Chapter 12, “Input Devices,” explains how to add an extension input device to the server and how to access the extension with MIT’s XInput Extension.

Chapter 13, “Direct Pixel Access DDX Interface,” describes the direct pixel access (DPA) interface for DDX handlers.

Chapter 14, “Debug Server Modules,” provides information about the debug server modules.

Chapter 15, “MIT Shared Memory Extension,” explains how to implement the MIT Shared Memory extension.

Appendix A, “The OWconfig File,” includes the default `OWconfig` file and explains its content.

Appendix B, “Packaging and Installation Hints,” discusses packaging and installation issues pertaining to loadable modules.

Appendix C, “Virtual User Input Device Interface,” explains the mechanism that sets up input devices to generate event codes and what a device driver needs to do in order to conform to the vuid interface.

Appendix D, “Dynamically Loadable Extensions,” discusses requirements X extensions must meet to be dynamically loadable by the server.

Related Books

Solaris Release Information

For information on this release, see the following:

- *“New Features and Changes” on page xxiii*
- *Solaris 2.5.1: Driver Developer Kit Introduction*
- *Solaris 2.5.1: Driver Developer Kit Installation Guide*
- *Solaris 2.5.1 Software Developer Kit Installation Guide*

OpenWindows Start Up Information

For information on how to start up the OpenWindows environment, see the following manuals:

- *Solaris 2.5.1: Driver Developer Kit Introduction*
- *Solaris Advanced User’s Guide*

OpenWindows Environment Information

To learn how to use the OpenWindows environment, see the following manuals:

- *Solaris User's Guide*
- *Solaris Advanced User's Guide*

X Window System Information

The following X Window System manuals are available through SunExpress or your local bookstore. Contact your SunSoft representative for information about ordering.

- *XView Reference Manual*, O'Reilly & Associates
- *XView Programming Manual*, O'Reilly & Associates
- *Xlib Reference Manual*, O'Reilly & Associates
- *Xlib Programming Manual*, O'Reilly & Associates
- *Programmer's Supplement for Release 5*, O'Reilly & Associates
- *X Toolkit Intrinsics Reference Manual*, O'Reilly & Associates
- *The X Window System, Third Edition*, Digital Press
- *The X Window System Server*, Digital Press

MIT Sample Server Porting Information

The following manuals are available online in the `/doc/Server` directory of the SUNWxwddk package. The default installation directory of this package is `/opt/SUNWddk/xserver`. These manuals are recommended if you are new to X11 server development. The associated filename is in parentheses.

- *Strategies for Porting the X v11 Sample Server* (`strat.ms`)
- *Definition of the Porting Layer for the X v11 Sample Server* (`ddx.tbl.ms`)

MIT Sample Server Information on `ftp.x.org`

The following MIT documentation is available to systems on the Internet. The MIT documentation resides on the `ftp.x.org` machine. Use the File Transfer Protocol (`ftp`) to download files from this system. If you need help using `ftp`, refer to the `ftp`(1) man page. To determine if your system is connected to the Internet, see your system administrator.

The directory and filename is given in parenthesis for the document.

- *X Window System, Version 11, Release 5. Release Notes* (`/pub/R5untarred/mit/RELNOTES.TXT`)

What Typographic Changes and Symbols Mean

The following table describes the type changes and symbols used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	**Meaning**	**Example**
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `system% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`system% su` `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.
Code samples are included in boxes and may display the following:		
`%`	UNIX C shell prompt	`system%`
`$`	UNIX Bourne and Korn shell prompt	`system$`
`#`	Superuser prompt, all shells	`system#`

New Features and Changes

The following sections describe new features and changes in this release of the OpenWindows X server.

Debug Server Modules

The server includes source files for debugging in the SUNWxwdes (SPARC), SUNWxwdex (x86), and SUNWxwdep (PPC) packages. See Chapter 14, “Debug Server Modules,” for more information.

New DGA Overlay Interface

The client DGA interface now supports direct access to windows in overlay planes. The following are new DGA API functions:

- `dga_draw_ovlstatechg`
- `dga_draw_ovlstatesetnotify`
- `dga_draw_ovlstategetnotify`
- `dga_draw_ovlstate`

See Chapter 10, “Direct Graphics Access Drawable Client Interface,” for detailed information.

DGA Backward Compatibility

The DGA drawable interface is compatible with DGA clients written with the window grabber interface in OpenWindows 3.4. Some of the routines described in the drawable interface are new, and some are reworked from last release. The DGA routines from the previous release are still supported. Specifically, the following interfaces are still provided:

- Colormap Grabber
- Miscellaneous Grabber
- Window Grabber

  This includes direct access window functions, cursor grabber functions, and retained window grabber functions.
- Multibuffer Grabber

Note – The window grabber and the multibuffer grabber interfaces are being phased out—they will be removed in a future release. Any new DDX handlers and clients should avoid using these interfaces; instead use the DGA drawable interface.

Window Grabber Supported Functions

Existing clients that use the older Window Grabber interface (`XDgaGrabWindow`) will continue to work with some types of windows. It is strongly recommended that you convert your client to use the new DGA drawable interface to directly access the newly supported drawable types and new DGA features.

The following Window Grabber routines are supported in this release:

- `XDgaGrabWindow`
- `XDgaUnGrabWindow`
- `dga_win_grab`
- `dga_win_ungrab`
- `dga_win_fbname`
- `dga_win_devfd`
- `DGA_WIN_LOCK`
- `DGA_WIN_UNLOCK`

- DGA_WIN_MODIF
- dga_win_clipchg
- dga_win_bbox
- dga_win_singlerect
- dga_win_empty
- dga_win_obscured
- dga_win_depth
- dga_win_borderwidth
- dga_win_set_client_infop
- dga_win_get_client_infop
- dga_win_clipinfo
- dga_win_cursactive
- dga_win_curschg
- dga_win_cursupdate
- dga_rtn_grab
- dga_rtn_ungrab
- dga_win_rtnchg
- dga_rtn_active
- dga_rtn_cached
- dga_rtn_devinfop
- dga_rtn_devtype
- dga_rtn_dimensions
- dga_rtn_pixels
- dga_db_grab
- dga_db_ungrab
- dga_db_display
- dga_db_interval
- dga_db_interval_check
- dga_db_display_done
- dga_db_write
- dga_db_read
- dga_db_write_inquire
- dga_db_read_inquire
- dga_db_display_inquire
- dga_win_dbinfop

Note – Do *not* use the window grabber interface with the new DGA drawable interface in the same application.

Note – Use the DGA drawable interface to grab multibuffered windows with the MBX extension. Do not use the window grabber interface to grab these windows; it is not guaranteed to work properly with multibuffer windows whether the window becomes multibuffered through MBX before or after it is grabbed.

Multibuffer Grabber Supported Functions

XGL Compatibility Interface

XGL provides a buffer control interface independent of MBX. It allows its clients to create multibuffers on a window and to switch the display of these buffers. The following routines are provided so that XGL can continue to provide this functionality. This interface, however, is deprecated; it will be removed in a future release. XGL is the only client that should use these routines.

Each one of these routines corresponds to an existing DGA buffer interface `dga_db_xxx` routine with the same suffix. Refer to the DGA client interface documentation on the `dga_db_xxx` routines for a complete description of the routine arguments and semantics.

Use these routines only with windows (that is, *not* pixmaps or multibuffered windows) *and* only when the window is locked.

- `dga_draw_db_grab`
- `dga_draw_db_ungrab`
- `dga_draw_db_write`
- `dga_draw_db_read`
- `dga_draw_db_interval`
- `dga_draw_db_display`
- `dga_draw_db_interval_wait`
- `dga_draw_db_interval_check`
- `dga_draw_db_display_done`
- `dga_draw_db_write_inquire`
- `dga_draw_db_read_inquire`
- `dga_draw_db_display_inquire`
- `dga_draw_db_dbinfop`

GPI Compatibility

The new DGA drawable interface is compatible with older DDX handlers that only support window grabbing. The old DGA screen initialization function, `DgaDevFuncsInit`, is still supported.

The new call, `dgaScreenInit` initializes DGA on the screen the same way as `DgaDevFuncsInit` (see "Initializing Drawable Grabs" on page 209). In addition to window grabbing, this provides support for pixmap and multibuffer grabbing.

Note – `DgaDevFuncsInit` and `dgaScreenInit` are mutually exclusive; a DDX handler should call only one of these.

Combining Client Interfaces

Client-Side Issues

An application can allow its window(s) to be directly accessed through the new DGA drawable client interface, as well as through the DGA window grabber interface. Only the drawable client interface can be used to grab pixmaps or multibuffered (MBX) windows.

The purpose of mixing new and old features is to enable applications to combine the use of graphics libraries with different revision levels, some using the old interface and some using the new interface. Individual libraries, like XGL, must only use one of these interfaces; it cannot mix functions from the old interface with functions from the new interface. It is strongly recommended that new applications or clients use the new drawable interface.

Server-Side Issues

The device-dependent DGA screen function used is determined by which screen initialization routine your DDX handler calls. Your DDX handler can call either the new screen initialization routine (`dgaScreenInit`) or the old routine (`DgaDevFuncsInit`).

Note – New DDX handlers should use `dgaScreenInit`.

If your DDX handler uses the old routine, `DgaDevFuncsInit`, only windows can still be directly accessed through the drawable interface, as well as the window grabber interface—pixmap and multibuffered windows cannot be grabbed.

DDX handlers that use `dgaScreenInit`, can access windows, pixmaps and multibuffered windows with the new DGA drawable interface.

Direct Pixel Access Interface

Direct pixel access (DPA) interface enables the window server to directly manipulate pixels in drawables that you control in your DDX handler. The Display PostScript (DPS) extension uses DPA to improve compositing performance.

Performance Enhancements

If you NFS mount the window server, mount it `setuid` allowable. This enables the server to take advantage of performance features in the Solaris operating system.

x86 *In-line Assembly Language Note*

The SunPro™ C Compilation system includes in-line assembly language provides direct access to x86 I/O instructions, as well as optimized in-line expansion templates. See the manual pages for `cc`(1) and `inline`(1), and SunPro's *ProCompiler C 2.0.1 Programmer's Guide* for more information.

If you want to include in-line assembly language in your code, place the in-line assembly definition file (with the `.il` extension) first in the `cc` command line:

```
cc -O inline.il bitblt.c
```

Common in-line examples are included in the file below.

```
////////////////////////////////////////////////////////////
/ File: inline.il
/
////////////////////////////////////////////////////////////
```

```
/ in and out
/       int ioaddr = 0x3c4;
/
/       Called as:
/               char data;
/               data = inb(ioaddr);
/
        .inline inb,4
        movl    (%esp), %edx
        xorl    %eax, %eax
        inb     (%dx)
        .end

/       Called as:
/               short data;
/               data = inw(ioaddr);
/
        .inline inw,4
        movl    (%esp), %edx
        xorl    %eax, %eax
        inw     (%dx)
        .end

/       Called as:
/               int data;
/               data = inl(ioaddr);
/
        .inline inl,4
        movl    (%esp), %edx
        xorl    %eax, %eax
        inl     (%dx)
        .end

/       Called as:
/               char data;
/               outb(ioaddr,data);
/
        .inline outb,8
        movl    (%esp), %edx
        movl    4(%esp), %eax
        outb    (%dx)
        .end

/       Called as:
/               short data;
```

```
/               outw(ioaddr,data);
/
        .inline outw,8
        movl    (%esp), %edx
        movl    4(%esp), %eax
        outw    (%dx)
        .end

/       Called as:
/               int data;
/               outl(ioaddr,data);
/
        .inline outl,8
        movl    (%esp), %edx
        movl    4(%esp), %eax
        outl    (%dx)
        .end

////////////////////////////////////////////////////////////
/ Set and clear direction flags
/

/       Called as:  cld();
/
        .inline cld,0
        cld
        .end

/       Called as:  std();
/
        .inline std,0
        std
        .end
```

DPS Extension Graphics Rendering

Due to a bug in this release of the DPS code, pixmaps used by DPS must have their `pPixmap->devKind` field equal to the width of the pixmap in bytes. This means that frame buffers that cache pixmaps in off-screen video memory need to use regular memory under certain conditions.

A flag has been added to inform DDX handlers when they should force pixmaps into regular memory. Make the following declaration in your DDX handler's `pScreen->CreatePixmap` routine:

```
extern int sunCreateDFBPixmap
```

Check this variable before creating a pixmap in off-screen memory. If the variable is `TRUE`, your DDX handler should force the pixmap into regular processor memory.

Note – This DPS bug workaround is unchanged from OpenWindows 3.4.

Test/Verify Recommendation

To test and verify a DDX handler, it is recommended that you run the UniSoft Test Suite. This test suite is available from the X Consortium.

You can access X Consortium information if your system is connected to the Internet. The UniSoft Test Suite information resides in the `/pub/XTEST` directory on the `ftp.x.org` machine. Use the File Transfer Protocol (`ftp`) to download files from this system. If you need help using `ftp`, refer to the `ftp`(1) man page. To determine if your system is connected to the Internet, see your system administrator.

DDX Porting Overview 1

The OpenWindows server is based on the X11R5 sample server from the MIT X Consortium. The OpenWindows server dynamically loads DDX handler modules at run time. This enables you, an Independent Hardware Vendor (IHV), to develop DDX modules that can be delivered as separate components.

Sun also provides DDX utility libraries to help you port the server to new graphics devices. These libraries contain functions common across devices.

See “Related Books” on page xix for recommended reading on the DDX layer.

Note – All porting interfaces documented in this manual are *uncommitted* interfaces; therefore, they might change in future releases in ways that could require you to change your DDX port.

1

The DDX Interface

As shown in Figure 1-1 on page 2, the DDX interface is quite extensive: the `Screen` structure alone contains approximately 70 functions.

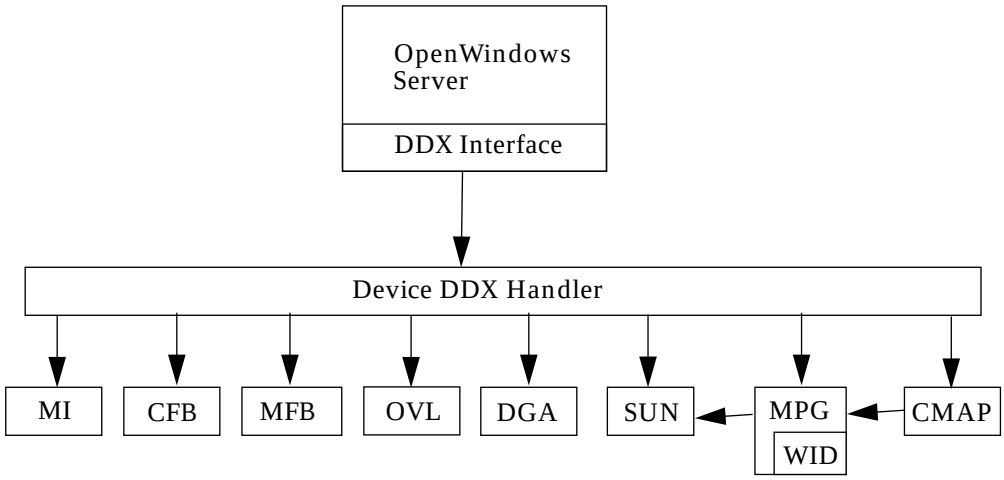


Figure 1-1 DDX Handler Utility Library Interfaces

Many of these functions do not need to be specialized for every device. Table 1-1 on page 2 describes general-purpose utility libraries that can be used to simplify your DDX handler implementation. The MI, CFB, and MFB libraries are from the X Consortium.

Table 1-1 Utility Libraries

Utility Library	Description
MI	Machine independent, high-level
CFB	Memory-mapped color frame buffers
MFB	Monochrome frame buffers
OVL	Transparent window overlay
DGA	Direct graphics access for client and DDX handler
SUN	Sun-specific `ioctl`s for frame buffers

Table 1-1 Utility Libraries *(Continued)*

Utility Library	Description
MPG	Multiple plane groups and multiple hardware colormaps
WID	Window Identification that is part of the MPG library
CMAP	Hardware colormap control

The Loadable DDX

The loadable DDX allows the server to dynamically load DDX modules at runtime instead of having to relink the server to add support for new devices. A DDX module is a shared object that is loaded by the server at runtime through an explicit call to `dlopen`(3X). The location of the DDX modules and their mappings between devices is determined by the `OWconfig` (OpenWindows configuration) file.

See Appendix A, “The OWconfig File” for more information about the `OWconfig` file.

Simple Frame Buffer Support

The OpenWindows server provides a set of general-purpose support routines for simple memory-mapped frame buffers. This includes the MFB library for monochrome frame buffers and the CFB library for color frame buffers.

Note – Although CFB code can be compiled to support depths of 2, 4, 8, 16, and 32 bits, only the 8, 16 and 32-bit depths are supported in this release.

Multiple-Plane Group Support

For devices with multiple-plane groups there is a utility library that provides most of the management functions necessary for MPG devices. This library also includes functions to minimize exposure events between windows that reside in different plane groups. The MPG interface is designed so that the CFB and MFB libraries can be used to render and manipulate windows.

1

The Loadable DDX Interface 2

The server interfaces to a loadable DDX handler. This chapter discusses the following topics:

- How the Server Interfaces With the Loadable DDX Handler
- The Initialization Function
- Device Self-Identification
- DDX Module Versioning

How the Server Interfaces With the Loadable DDX Handler

The server interfaces with the loadable DDX handler in the following manner.

1. The module containing the device's DDX handler is installed in the *modules* directory—the default directory is `/usr/openwin/server/modules`. Since the `/usr/openwin` path could be an NFS mount point, a parallel directory structure can be created on the local machine under `/etc/openwin/server/modules`. The DDX handler modules that are distributed with the standard OpenWindows packages are always installed in `/usr/openwin/server/modules`. DDX handlers supplied by Independent Hardware Vendors (IHVs) can be installed in either the machine local path (preferred, if the installation will not be shared between different machines), or under the default path `/usr/openwin`. (See Appendix B, "Packaging and Installation Hints" for more details).

The server searches for DDX handlers using the following path: `/etc/openwin/server/modules:/usr/openwin/server/modules`. This search path cannot be altered by the end user.

Note – For debugging purposes, create symbolic links from `/etc/openwin` to point to your development environment where you want to debug your code. You will need to edit the `/etc/openwin/server/etc/OWconfig` file to load/configure your DDX handler. Use `/etc/openwin` because it is intended to be local to the target machine (your development environment); do *not* use `/usr/openwin` because it is intended to be shared by many machines.

---

2. The devices that are added as Screens are specified with the `-dev` command-line option to `openwin`. For example:

```
example% openwin -dev /dev/cgsix0 -dev /dev/cgthree0 left
```

For SPARC systems – If no devices are specified on the command line, the server defaults to opening the `/dev/fb` device. This is a symbolic link to the appropriate driver entry in `/dev/fbs`, created when the system is booted with `boot -r`. See the `openwin` (1) man page for more information.

For x86 systems – If no devices are specified on the command line, the server defaults to values specified in the `OWconfig` file. The argument specified with the `-dev` command line option is the name of a supported display (such as `8514`, `v256`, or `vga4`). This name is matched against the `name` attribute specified in a resource line. See Appendix A, “The OWconfig File” for more details.

For PowerPC systems – Use the `kdmconfig` utility included with this release to configure your keyboard, display, and mouse. To properly configure your interface, you need to know if your display is type p9000 or p9100. To find out, enter the following command: `dmesg | grep screen`
The display type will appear on your screen. Within `kdmconfig`, select the appropriate graphic card choice using this information. For example, if the

`dmesg` command returns p9000, pick the Diamond Viper -P9000 selection. In this release of Solaris *PowerPC edition*, the multi-headed system is not supported.

3. The server reads a configuration file (`OWconfig`) to determine the filename of the dynamically loadable DDX handler. This file is searched using the search path `/etc/openwin/server/etc:/usr/openwin/server/etc`. If the configuration file is found in both locations, the server constructs a database combining the two files. This search path cannot be altered by the end user.

   For debugging purposes only, an alternate directory in which the `OWconfig` file can be found can be specified by setting the environment variable `OW_CONFIG_PATH` before running the server. This environment variable is not documented in any end-user documentation and should not be recommended to end users.

```
example% setenv OW_CONFIG_PATH /home/joe
```

   (See Appendix A, “The OWconfig File” for more information on the `OWconfig` file).

4. The server loads the appropriate DDX handler module and calls `ddxInitFunc`. The `ddxInitFunc` initializes the device and data structures so that the server can run. The `ddxInitFunc` has the same specification as the `scrInitProc` defined in the MIT document, *Definition of the Porting Layer for the X v11 Sample Server.*

The Initialization Function

Each graphics adapter’s DDX handler defines an initialization function that is called at server restart. This function initializes the device and the Screen data structure associated with it.

```
Bool xxxInit(int index, ScreenPtr pScreen, int argc,
       char **argv);
```

For SPARC systems – For a sample implementation of the `ddxInitFunc` and a complete sample implementation of a minimal DDX handler for a simple-memory frame buffer, see the sample cg3 DDX handler online in `ddk_2.5.1/xserver/server/ddx/solaris/sparc/cg3`.

For x86 systems – For a sample implementation of the `ddxInitFunc` and a complete sample implementation of an equivalent DDX handler for a simple 256 color VGA display adapter, see the sample v256 DDX handler online in `ddk_2.5.1/xserver/server/ddx/solaris/i386/displays/v256`.

For PowerPC systems – For a sample implementation of the `ddxInitFunc` and a complete sample implementation of an equivalent DDX handler for a color display adapter, see the sample p9100 DDX handler online in `ddk_2.5.1/xserver/server/ddx/solaris/prep/displays/p9100`.

Device Self-Identification

As noted in Step 2 on page 6, devices added as X screens are specified by using the `openwin` command line and the `-dev` option. The server opens each device specified with `-dev` in its `InitOutput` routine, in turn. (If no devices are specified, the default device is `/dev/fb`). It then issues an `ioctl` (`VIS_GETIDENTIFIER`) to the device driver. The device driver for the graphics device is expected to implement this `ioctl` to identify the device uniquely. The `ioctl` returns a unique string name. The server looks up this string name in the `OWconfig` file under the class `XSCREEN`. The DDX handler filename specified in this entry is then dynamically loaded by the server, and the `ddxInitFunc` symbol specified in the entry is called by the DIX routine `AddScreen`. For a complete specification of the device identification `ioctl`, see *Writing Device Drivers*.

For x86 systems – This release does not automatically self-identify the various video card adapters supported. The video cards are not able to specify the card type and supported resolutions and features on most Intel architecture machines. Default video adapter types, initialization and resolution information is stored in the `OWconfig` file for Intel machines. This information

is created during installation with input from the user. The default video display selection is also determined during installation and stored in the `OWconfig` file.

For PowerPC systems – This release does not automatically self-identify the various video card adapters supported. The video cards are not able to specify the card type and supported resolutions and features on most Intel architecture machines. Default video adapter types, initialization and resolution information is stored in the `OWconfig` file for Intel machines. This information is created during installation with input from the user. The default video display selection is also determined during installation and stored in the `OWconfig` file.

DDX Versioning

A versioning scheme is required to ensure that the server and the DDX handler it dynamically loads are compatible. The OpenWindows server component of the Device Developer's Kit (DDK) contains the header files and documentation that define the DDX interface (consisting of data structures and functions) between the server and the dynamically loaded DDX handlers. This component is used to build a DDX handler and has a version number, referred to as the DDK version number. The DDK version number is available as a manifest constant in the header file `sun.h` that every DDX handler must include. The following are the important defines from the `sun.h` header file:

```
/*
 * Server Device Developer's Kit (DDK) Version number
 */

#define DDK_MAJOR_VERSION 1
#define DDK_MINOR_VERSION 2

typedef struct {
    CARD16 majorVersion;
    CARD16 minorVersion;
} sunDDKVersionRec, *sunDDKVersionPtr;

sunDDKVersionPtr sunGetDDKVersion(void);
```

2

Each release of OpenWindows is accompanied by a release of the DDK that was used to build the server. This DDK is used by IHVs to build DDX handlers that are compatible with the OpenWindows server in that release. IHVs supplying DDX handlers must follow these versioning rules:

- The DDK `majorVersion` used to build the DDX handler is stamped in the filename of the handler, such as, `ddxSUNWcg6.so.1`. The convention used in naming DDX handlers is:

```
ddx<organization><device>.so.<majorVersion>
```

- The server is also stamped internally with the DDK version number used to build the server. The server never dynamically loads a module with a `majorVersion` greater than its own. For example, a server built with a DDK version 1.0 will never load a DDX handler built with a DDK version 2.0.
- The server dynamically loads a DDX handler with a DDK `majorVersion` less than its own DDK `majorVersion`, *only if* the server has explicitly decided to emulate that lesser `majorVersion` interface. Every time a new version of OpenWindows and a new version of the server DDK are released, this DDK document specifies which, if any, DDK `majorVersions` are emulated by the server.

Note – For this release of the server DDK, no prior versions are emulated.

- The server dynamically loads a module that has the same DDK `majorVersion` as itself. If the DDX module depends on functionality that was added in a particular `minorVersion` of the DDK, it is up to you to check for the existence of that functionality, by checking the server's DDK version number.

  A DDX module can provide its own workaround if the functionality does not exist, or it can fail with an appropriate error message indicating the server version number it requires.

The functionality differences between `minorVersion` releases of the DDK will be documented in future releases of this manual. A DDX handler module can check the server's DDK version number by calling the sun library function `sunGetDDKVersion`.

```
#include “sun.h”

sunDDKVersionRec serverVersion = sunGetDDKVersion();

if (serverVersion->majorVersion == 1
    && serverVersion->minorVersion < 5) {
    ....
}
else {
    ....
```

2

Screen Initialization 3

The `ddxInitFunc` device function should initialize the `Screen` structure and all of its function vectors. See “The Initialization Function” on page 7 for information on `ddxInitFunc`. This chapter provides information on some aspects of `Screen` initialization common to many devices. Additional initialization steps might be required depending on the utility layers you use in your DDX handler. These steps are documented in subsequent chapters describing the utility layers provided by the server.

A set of common utility functions is provided in the server that:

- Allocate private data structures
- Inquire current command-line options
- Advertise pixmap formats and supported visuals

For SPARC systems – A complete sample implementation of the `ddxInitFunc` for a simple-memory frame buffer is available on line. See the cg3 DDX handler in the following directory:
`ddk_2.5.1/xserver/server/ddx/solaris/sparc/cg3`

For x86 systems – A complete sample implementation of the `ddxInitFunc` for a simple 256 color VGA display adapter is available on line. See the v256 DDX handler in the following directory:
`ddk_2.5.1/xserver/server/ddx/solaris/i386/displays/v256`

3

For PowerPC systems – A complete sample implementation of the `ddxInitFunc` for a color display adapter is available online. See the p9100 DDX handler in the following directory:
`ddk_2.5.1/xserver/server/ddx/solaris/prep/displays/p9100`

Initialization Steps

Your device handler's `ddxInitFunc` function should perform the following steps to initialize the `Screen` structure:

- Initialize the `ScreenRec` structure fields
- Initialize the device
- Map device registers and the frame buffer into the address space (if the device is memory-mappable)
- Allocate required private data structures
- Query command-line options that affect your DDX handler
- Advertise pixmap formats, visuals, and depths the device supports
- Initialize various utility layers you plan to use

It is important to know that `ddxInitFunc` could be called more than once during the lifetime of the server. The server is capable of restarting, and the `ddxInitFunc` is called again when this happens. This is why it is important to wrap `pScreen->CloseScreen` in your `ddxInitFunc`, and free all data structures allocated in the `ddxInitFunc` and elsewhere in the DDX handler.

Additionally, on multi-screen configurations which have multiple Screens of the same device type (hence served by a common DDX handler), the DDX handler module only needs to be loaded once into the server. Thereafter, the index of the `Screen` is used to distinguish between `Screen`s. In this case, the `ddxInitFunc` will be called once for each `Screen`. It is recommended that any `Screen` private data required by the DDX handler be stored by allocating a `devPrivate` index on the `Screen` structure. The use of global variables in the DDX handler is discouraged for the same reason.

sunGetDDKVersion

```
sunDDKVersionPtr
sunGetDDKVersion();
```

One of the first things your DDX handler might do is check the DDK version number of the server that is attempting to load it. This is useful if your DDX handler depends on server functionality that was added in a specific minor version of the server DDK. Call the server function `sunGetDDKVersion` to obtain this information. See “DDX Versioning” on page 9 for a complete specification.

Note – The sample DDX handlers provided on line do not call `sunGetDDKVersion` because they are not dependent on any minor version functionality in the server DDK.

Initialize the ScreenRec Functions

Since some utility layers *wrap* the functions in the `ScreenRec`, it’s important that your DDX handler initialize all the functions in the `ScreenRec` with valid function pointers, or `NULL` pointers for functions that are expected to be wrapped by other utility layers. The `ScreenRec` that is passed to the `ddxInitFunc` is uninitialized. It is the responsibility of `ddxInitFunc` to initialize `ScreenRec` with valid data or `NULL` as appropriate. To do this, at the beginning of your `ddxInitFunc`, include code that `NULL`s out all the `Screen` functions that are not supplied in your DDX handler**.** This could help prevent bugs due to uninitialized `ScreenRec` function pointers in your DDX handler.

Note – This step is not required if your `ddxInitFunc` provides valid function pointers for all the `ScreenRec` functions.

```
/* For example, if your DDX handler does not provide an
 * implementation of pScreen->BlockHandler or
 * pScreen->WakeupHandler, but these are expected to be wrapped
 * from the sunKbd device handler (for the keyboard) later in the
 * Initialization sequence.
*/
pScreen->BlockHandler = NULL;
pScreen->WakeupHandler = NULL;
```

sunScreenAllocate

```
Bool
sunScreenAllocate(ScreenPtr pScreen)
```

Purpose This function allocates a `Screen` private index (`sunScreenIndex`) and allocates the `sunScreenRec` data structure used by various utility layers (defined in `server/ddx/solaris/sun.h`).

Called by Your `ddxInitFunc` before initializing any utility layers.

Results A pointer to the `sunScreenRec` structure is stored in `pScreen->devPrivates[sunScreenIndex].ptr`.

Returns `TRUE` on success
else `FALSE`

The `sunScreenRec` data structure must be freed in the `CloseScreen` routine of your DDX handler. Some of the fields of this data structure are filled by various Sun utility layers; however, a few fields need to be filled in by your `ddxInitFunc`.

Note – A future release of the server might provide interfaces that will make this data structure opaque to the DDX handler.

Access the private data structure using the macros `GetScreenPrivate` and `SetupScreen` defined in `sun.h`.

```
#define GetScreenPrivate(s) \
((sunScreenPtr) ((s)->devPrivates[sunScreenIndex].ptr))
#define SetupScreen(s) \
sunScreenPtr pPrivate = GetScreenPrivate(s)
```

Device-Dependent Initialization

Device-dependent initialization typically consists of the following steps:

- Opening the device-special file for the graphics device
- Mapping the device registers or the frame buffer into the server address space (if the device is memory-mappable)
- Storing the file descriptor and memory mapping information in the private `sunScreenRec` data structure

---

Note – The sample DDX handlers (such as the cg3) use a private helper function called `sunOpenFrameBuffer` to open the device. This routine is called for example only; do not call it from your `ddxInitFunc`. It relies on `ioctl`s that are private to the cg3 device driver, and are not required to be implemented in your device driver.

The device-special filename you should `open` in your `ddxInitFunc` can be obtained by calling the `GetDevname` macro in `sun.h`.

```
char *
GetDevname(int index);    /* The Screen's index */
```

The file descriptor and device name should be stored in the `sunScreenRec` private structure. These are used by other utility layers (such as DGA) in the server. The code in your `ddxInitFunc` might look like this:

```
{
SetupScreen(pScreen);
    ...
    pPrivate->sunFbs.fd = open(GetDevname(index), O_RDWR, 0);
    strcpy(pPrivate->sunFbs.devName, GetDevname(index));
    ...
}
```

If your cursor implementation uses the `sunPointerScreenFuncs` utility functions that implement `Screen` crossings and cursor warping, you should initialize the `pPrivate->sunFbs.EnterLeave` field to NULL in your `ddxInitFunc`. See Chapter 4, “Cursors” for information on `sunPointerScreenFuncs`.

Note – The sample DDX handlers store device-dependent information about the device memory-mappings in some of the other private fields of the `sunScreenRec` data structure, for use in the `CloseScreen` routine. It is recommended that you minimize dependencies on the `sunScreenRec` private data structure, and store device-dependent information in data structures that are private to your own DDX handler. These data structures can be stored by allocating a `devPrivate` index on the `Screen` that is private to your DDX handler.

sunSetPixmapFormat

```
Bool
sunSetPixmapFormat(PixmapFormatRec *request)
```

Purpose This function is used by each device to advertise the pixmap formats supported for each depth. If there are multiple Screens supporting the same depth, they should support a common pixmap format for that depth. The first pixmap format defined for that depth is the one used for all Screens that are added.

Called by — Your `ddxInitFunc` calls this routine once for each depth that it plans to export in the `pScreen->allowedDepths` field.

Returns — `TRUE` if it is the first pixmap format definition for specified depth, or if it is a repeat definition that agrees with the existing one

`FALSE` for any attempt to define a new format for an existing depth. The `request` variable is set to the defined format for that depth; use the format returned in your new Screen's DDX handler.

Table 3-1 lists the pixmap formats supported by some devices.

Table 3-1 Pixmap Formats

Depth	BitsPerPixel	ScanlinePad
1	1	BITMAP_SCANLINE_PAD
4	4	BITMAP_SCANLINE_PAD
4	8	BITMAP_SCANLINE_PAD
8	8	BITMAP_SCANLINE_PAD
24	32	BITMAP_SCANLINE_PAD
32	32	BITMAP_SCANLINE_PAD

If you want your new device to support one of these depths, use one of the pixmap formats specified in Table 3-1 so that your device can be used with devices by other IHV's in a multi-screen configuration.

Note – The two 4-bit deep screen formats may not coexist simultaneously with other IHV's devices. The 4-bit deep, 4 BitsPerPixel format is the only 4-bit deep screen format supported during an X server session.

If a new depth is exported by a device, register the pixmap format with Sun for inclusion in this table, or be prepared to handle differing pixmap formats (that is, `sunSetPixmapFormat` returns `FALSE`) in your DDX handler.

sunGetMonitorRes

```
void
sunGetMonitorRes(int screenIndex, int *dpix int *dpiy)
```

Purpose This function gets the monitor's resolution.

Results The default value, 90 DPI, is used if a monitor resolution is not specified.

Note – Currently the monitor's resolution is specified with the `-dev` command-line option. Future releases of the OpenWindows server might offer alternate mechanisms to query the monitor resolution, such as specifying it in the `OWconfig` database.

sunGetVisualInfo

```
void
sunGetVisualInfo(int screenIndex, int *defClass, int *defDepth,
       Bool *grayVis);
```

Purpose This function gets the command-line options for Visual information specified by the user for the `Screen`. Since the user can specify the default visual class, the default depth, or gray visual, the DDX handler must query these values before setting up the visuals to be exported for this `Screen`.

Returns The default visual class specified as `defclass` in the `-dev` command-line option, if specified; else the default specified with the `-cc` option; else `-1`.

The `defDepth` specified with the `-dev` option.

`TRUE` for `grayVis`, if the user specified the `grayvis` modifier to the `-dev` option. This suppresses color visuals and is useful if a grayscale monitor is connected to the

device. If `grayVis` is `TRUE`, this function ensures that the user has selected a gray `defClass`, if a `defClass` has been specified; else `defClass` is set to `-1`.

Export Supported Visuals

The `ddxInitFunc` should advertise the visuals it supports, based on device capabilities and user preferences selected with command-line options.

Note – The sample cg3 DDX handler uses the cfb utility layer to select and advertise its visual list. See the `sunCG3C.c` file in the `server/ddx/solaris/reference/cg3` directory for details.

Initialize Utility Layers

The various utility layers used by your DDX handler should be initialized in your `ddxInitFunc`. Depending on the utility layers used, the order of initialization might be important, as a number of the utility layers wrap the DDX functions.

Initialize the Banner Code

```
extern int noBanner;
extern void sunInitBanner(ScreenPtr pScreen);
```

Purpose This function initializes the banner display code.

Called by The following code in your `ddxInitFunc`:

```
{
    extern int noBanner;
    extern void sunInitBanner(ScreenPtr pScreen);
    ...
if (!noBanner) {
    sunInitBanner(pScreen);
}
    ...
}
```

Results A banner is displayed by the server on every Screen, unless `openwin` is started with the `-nobanner` command-line option.

Note – The sample cg3 DDX handler does not implement this directly. It calls a private helper function, `sunScreenInit`, to initialize the banner code and perform other miscellaneous initialization. `sunScreenInit` is called for example only; do not call it from your `ddxInitFunc`. It has the undesirable effect of installing a `SaveScreen` routine that relies on `ioctl`s private to the cg3 device driver.

Supply a SaveScreen Function

```
Bool
pScreen->SaveScreen(ScreenPtr pScreen, int on)
```

The field `on` has the following values:

`SCREEN_SAVER_ON`	Turns on the screen saver; disables video
`SCREEN_SAVER_OFF`	Turns off the screen saver; enables video
`SCREEN_SAVER_FORCER`	Updates time of last screen saver mode change

Note – The sample DDX handlers install a private helper routine called `sunSaveScreen` as the `pScreen->SaveScreen` routine. Do not use this implementation in your DDX handler; it relies on `ioctl`s private to the sample device implementation. Instead, implement your own `SaveScreen` routine.

The following is a simple `SaveScreen` implementation:

```
Bool
xxxSaveScreen(ScreenPtr pScreen, int on)
{
    if (on == SCREEN_SAVER_FORCER) {
        SetTimeSinceLastInputEvent();
    }
    else {
        if (on == SCREEN_SAVER_ON) {
            VIDEO_OFF(); /* Device specific video disable */
        }
        else {
            VIDEO_ON(); /* Device specific video enable */
        }
    }
    return TRUE;
}
```

Supply a CloseScreen Function

The `CloseScreen` function should be *wrapped* by `ddxInitFunc`. The `CloseScreen` routine should clean-up all the device state, to the extent required by the device. For example, you might follow these steps in your `CloseScreen` function:

- Enable video, if the `ScreenSaver` disabled video
- Clear the `Screen` before exiting
- Reset the device's LUT with colors appropriate for displaying console messages, if the device also acts as a system console
- Call the `CloseScreen` functions that were wrapped
- Unmap the device registers and frame buffer, if it is a memory-mapped frame buffer
- Close all file descriptors opened by the DDX handler
- Free all allocated memory

For SPARC systems – For a sample `CloseScreen` implementation, see the `ddk_2.5.1/xserver/server/ddx/solaris/sparc/cg3` directory.

For x86 systems – For a sample `CloseScreen` implementation, see the `ddk_2.5.1/xserver/server/ddx/solaris/i386/displays/v256` directory.

For PowerPC systems – For a sample `CloseScreen` implementation, see the `ddk_2.5.1/xserver/server/ddx/solaris/prep/displays/p9100` directory.

Initializing Visual Gamma

If your device supports linear and nonlinear visuals, you might want to advertise the `XSolarisGetVisualGamma` property; otherwise, it is optional.

Gamma-Corrected Visuals

Some devices have linear, or gamma corrected visuals. Applications can distinguish between linear visuals and nonlinear visuals by calling `XSolarisGetVisualGamma`(3). For more information on this routine see the *OpenWindows Server Programmer's Guide*, which is part of the SDK, and the manual page.

Devices that have linear visuals should export these visuals by adding them to the `pScreen->visuals` list just like any other visual. A root window property distinguishes it from the nonlinear visuals.

Note – If a device has a linear visual with a nonlinear counterpart having a gamma of approximately 2.22, it is a good idea to place the nonlinear one before the linear one on the screen visual list. Most X11 applications prefer a nonlinear visual with this gamma value. Make the server default visual nonlinear as well.

The Monitor Intensity Response Property

Linear and nonlinear visuals are differentiated by describing their gamma value through a root window property, `XDCCC_LINEAR_RGB_CORRECTION`. It is a standard X11 ICCCM property originally created for the X Color Management System. The routine `XSolarisGetVisualGamma` also reads it. This property specifies for a visual a set of tables (one for each of the red, green, and blue color channels) that describe how the intensity of colors coming out of the frame buffer map to actual display colors on the monitor screen. This is the *intensity response* of colors displayed in the visual. If the intensity response of more than one visual is described, the property contains more than one set of tables. See the *X Window System* for detailed information on `XDCCC_LINEAR_RGB_CORRECTION`.

Here are some guidelines for creating the property:

1. Create the property with type `XA_INTEGER` and format 16.
2. Visuals with a gamma of exactly 2.22 may be omitted from the property. In this case, `XSolarisGetVisualGamma` assumes a value of 2.22. This is the most efficient way to specify this value.
3. Visuals with a gamma of exactly 1.0 should be represented using a 2-entry type 0 table. For each channel, the first entry should be (`0`, `0`) and the second entry should be (`numIntensities` - 1, `0xffff`), where `numIntensities` is (`1 << visual->bitsPerRGBValue`).
4. All other visuals should be represented using a type 1 table. To create this type of table, the following expression should be evaluated for each color channel and for each value `x` between `0` and `xmax`:

```
y = (unsigned short) (( 65535.0 * pow((double)x/(double)xmax, γ)) + 0.5)
```

   where γ is the gamma of the visual and xmax is numIntensities -1 (see guideline #3).

5. bpr is the `bitsPerRGBValue` member of the visual structure.
6. If the gamma of all visuals is exactly 2.22, the property does not need to be created at all.

Note – XDCCC_LINEAR_RGB_CORRECTION describes the intensity response of the entire path from the frame buffer through the monitor, rather than just the gamma correction function.

Note – It may be acceptable if the intensity response described in this property is only approximate. The DDX may not know the specific monitor attached to the device and may need to provide an estimate. A gamma value of 2.22 is a good estimate for most monitors.

The next section describes how to create a root window property from within a DDX handler screen initialization function.

Initializing a Root Window Property

A root window property cannot be directly created from a DDX screen initialization routine because at the time this routine is called the root window has not yet been created. However, the initialization routine can arrange for the property to be created at a later time, after the root window has been created.

The first call to `pScreen->CreateWindow` is for the root window. This screen function should be wrapped. On the first call to the wrapper function, the property should be created on the argument window. This is guaranteed to be the root window.

A property is created by first determining the atoms for the property's name and type strings. If the string has a predefined atom, simply use the defined symbol for that atom (see `/usr/openwin/include/Xatom.h` for the list of predefined atoms). Otherwise, call `MakeAtom` to intern the string and receive back an atom.

```
Atom
MakeAtom (char *string, unsigned len, Bool makeit)
```

`string` is the name of the string to be interned, `len` is its length (in bytes), and `makeit` should be `TRUE`. A numeric value (the atom) is returned.

Next, the property is added to the window by calling `ChangeWindowProperty`:

```
int
ChangeWindowProperty (WindowPtr pWin, Atom property, Atom type,
        int format, int mode, unsigned long len, pointer value,
        Bool sendevent)
```

`pWin` is the argument to the `CreateWindow` wrapper routine, `property` is the interned atom for the string "`XDCCC_LINEAR_RGB_CORRECTION`", `type` is `XA_INTEGER`, format is 16, mode is `PropModeReplace`, `len` is the length of the property (in units of 16-bit short words), `value` is pointer to the property data and `sendevent` should be `FALSE`. `Success` is returned if the property creation succeeded.

Note – It is a good idea to unwrap `pScreen->CreateWindow` after the property has been created so other calls to `CreateWindow` do not incur extra overhead.

3

Cursors 4

Cursor implementations for most device handlers fall into one of these categories:

- Software cursor
- Limited-size hardware cursor

You can use a number of software layers to help with your cursor implementation, depending on your graphics adapter hardware. This chapter helps you choose the cursor layer that is best for your hardware. The porting interface for each of the available layers is also discussed in detail.

The Basic DDX Interface

The basic DDX interface describing cursor routines for a screen is defined in the MIT sample server document *Definition of the Porting Layer for the Xv11 Sample Server.* This interface consists of the following functions:

```
pScreen->RealizeCursor(pScr, pCurs)
pScreen->UnrealizeCursor(pScr, pCurs)
pScreen->DisplayCursor(pScr, pCurs)
pScreen->RecolorCursor(pScr, pCurs, displayed)
pScreen->ConstrainCursor(pScr, pBox)
pScreen->PointerNonInterestBox(pScr, pBox)
pScreen->CursorLimits(pScr, pCurs, pHotBox, pTopLeftBox)
pScreen->SetCursorPosition(pScr, newx, newy, generateEvent)
```

It is possible for your DDX handler to port directly at this level. You can do this by supplying fully customized versions of these functions in your screen initialization routine.

A DDX implementation of these cursor functions is provided in utility layers discussed in the remainder of this chapter. If your graphics device is an MPG (multiple plane group) device and your cursor implementation is in a separate plane group, refer to Chapter 5, “Multiple Plane Group Interface.”

Note – Due to implementation constraints in the server, the Sun mouse implementation requires you to initialize the `mipointer` code in your DDX handler. The following `miPointer` routines are used by the `ddxSUNWmouse` device handler.

- `miPointerGetMotionEvents`
- `miPointerGetMotionBufferSize`
- `miPointerDeltaCursor`
- `miPointerPosition`
- `miPointerAbsoluteCursor`

Software Cursor

This section describes the software cursor porting interface for your DDX handler.

miDC Layer

The mi utility layer provides a software cursor implementation in the miDC (mi Display Cursor) layer. If your display adapter does not have any hardware cursor capability, a complete software cursor implementation can be enabled by calling the `miDCInitialize` function in your screen initialization routine.

For SPARC systems – For an example of a software cursor implementation, see the cg3 reference DDX handler in the following directory:
`ddk_2.5.1/xserver/server/ddx/solaris/sparc/cg3`

For x86 systems – For an example of a software cursor implementation, see the v256 reference DDX handler in the following directory:
`ddk_2.5.1/xserver/server/ddx/solaris/i386/displays/v256`

For PowerPC systems – For an example of a software cursor implementation, see the p9100 reference DDX handler in the following directory:
`ddk_2.5.1/xserver/server/ddx/solaris/prep/displays/p9100`

Call the `miDCInitialize` function after most of the screen functions have been initialized. It uses the miSprite layer that wraps most of the screen functions. See the sample cg3, v256, or p9100 handler for an example of the order in which to call the screen initialization functions.

Call the `miDCInitialize` routine with the following parameters:

```
#include “mipointer.h”
...
miDCInitialize(ScreenPtr pScreen,
       miPointerScreenFuncPtr screenFuncs);
```

The Sun layer provides a set of `screenFuncs` that is an array of pointers to functions required by the miPointer layer (such as `CursorOffScreen`, `CrossScreen` and `WarpCursor`).

The following example is all that is required in your DDX handler to enable the software cursor implementation in the mi layer.

```
#include “sun.h”
...
#include “mipointer.h”
...
...
extern miPointerScreenFuncRec sunPointerScreenFuncs;
...
miDCInitialize(pScreen, &sunPointerScreenFuncs)
```

The following sections describe in more detail the mi layers that the miDC layer uses to provide a software cursor. If you are in a hurry to get a software cursor working on your graphics adapter, you do not need to know all of the mi layer details.

The miDC layer internally uses the miSprite and miPointer layers to implement the software cursor.

miPointer Layer

The miPointer layer offers a set of the basic DDX cursor interface. This means that it supplies an implementation of the DDX eight discussed in “The Basic DDX Interface” on page 29. To get the miPointer layer to work however, you must provide an implementation of `miPointerSpriteFuncs` and `miPointerScreenFuncs`. Each of these is an array of four functions that you pass to `miPointerInitialize`.

```
miPointerInitialize(ScreenPtr pScreen,
       miPointerSpriteFuncPtr spriteFuncs,
       miPointerScreenFuncPtr screenFuncs, Bool waitForUpdate)
```

`miPointerSpriteFuncs` is a set of four functions that implement the sprite software.

```
RealizeCursor(pScr, pCurs)
UnrealizeCursor(pScr, pCurs)
SetCursor(pScr, pCurs, x, y)
MoveCursor(pScr, x, y)
```

`miPointerScreenFuncs` is a set of functions that implement `Screen` crossings and cursor warping.

```
CursorOffScreen(pScr, x, y)
CrossScreen(pScr, entering)
WarpCursor(pScr, x, y)
EnqueueEvent(xEvent)
NewEventScreen(pScr)
```

Irrespective of which sprite implementation you choose, use the `miPointerScreenFuncs` implementation provided in the `sun` layer. The `sunPointerScreenFuncs` array provides implementations for `CursorOffScreen`, `CrossScreen`, and `WarpCursor`. It has `NULL` pointers for `EnqueueEvents` and `NewEventScreen`; these are initialized by `miPointerInitialize` to the routines `mieqEnqueue` and `mieqSwitchScreen`. The `sunPointerScreenFuncs` array is used by including the following code in your DDX handler.

```
#include “sun.h”
...
#include “mipointer.h”
...
...
extern miPointerScreenFuncRec sunPointerScreenFuncs;
```

miSprite Layer

The miSprite layer provides a set of the `miPointerSpriteFuncs` required to drive the miPointer layer. The `miSprite` layer offers a software *sprite*—a software overlay that can be moved around on the screen, while preserving other images on the screen.

The `miSprite` layer does this by wrapping all the `Screen` rendering functions and all the GC functions. It saves areas under the sprite, and restores them when the sprite moves. It removes the sprite while rendering occurs to areas under the sprite, and restores the sprite when required. To get `miSprite` to work, `miSpriteInitialize` needs to be passed an array of `miSpriteCursorFuncs`.

```
miSpriteInitialize(ScreenPtr pScreen,
       miSpriteCursorFuncPtr cursorFuncs,
       miPointerScreenFuncPtr screenFuncs);
```

`miSpriteCursorFuncs` is an array of these functions:

```
RealizeCursor(pScr, pCurs)
UnrealizeCursor(pScr, pCurs)
PutUpCursor(pScr, pCurs, x, y)
SaveUnderCursor(pScr, x, y, w, h)
RestoreUnderCursor(pScr, x, y, w, h)
MoveCursor(pScr, x, y, w, h, dx, dy)
ChangeSave(pScr, x, y, w, h, dx, dy)
InCursorPlanes(pWin)
```

An implementation of these functions is provided by the `miDC` layer. This layer draws the software cursor image.

miSetZeroLineBias Function

```
extern void miSetZeroLineBias (ScreenPtr pScreen, unsigned int
bias);
```

Purpose This function allows the developer to specify the device line renderering bias. Each device may specify its own line bias based on a bias byte. This bias is honored by all thin line rendering in `cfb`, `mfb` and `mi`.

Arguments `bias` is an 8-bit mask indicating which octants to step axially when the error term is 0. The preprocessor definitions needed to construct a bias byte are defined in the header file `mipixel.h` and are named OCTANT1 through OCTANT8.

Results If this function is not called when needed to tune the software thin line bias for a device, a default value is automatically provided.

Hardware Cursor

This section describes the porting interface for your DDX handler if you have a hardware cursor. The hardware cursor is limited by the size of the cursor image registers.

The X Protocol leaves it up to the server implementation to decide what the cursor looks like if the cursor defined for the `Screen` exceeds the physical limits imposed by the cursor hardware. Some server implementations choose to trim the cursor image around the *hotspot* such that it fits into the size limits imposed by the hardware.

Another strategy, and one that is followed by the OpenWindows server, is to revert to a software cursor implementation whenever a cursor defined for a Window does not fit in the hardware. This means that if there are multiple cursors defined on the same screen, some small enough to fit in the hardware cursor registers, and some larger, the cursor dynamically switches between hardware and software forms as the pointer is moved across the screen. This hardware and software cursor switching is implemented in a utility layer in the server, called `sunSprite`.

The sunSprite Layer

The sunSprite layer implements a sprite that can switch between hardware and software forms. It uses the software cursor layers described in “Software Cursor” on page 30 whenever the cursor does not fit into hardware.

In your DDX handler, you might want to use the `sunSprite` layer to handle your cursor if you want to switch between hardware and software cursors on the same screen. It is recommended that the cursor defined by the application be displayed as actual size, even if this means that it cannot fit into hardware. This is motivated by the desire to keep the application’s look and feel consistent across all graphics adapters supported by the OpenWindows server.

The `sunSprite` code is initialized in the DDX handler’s screen initialization function by calling the following function:

```
#include “sun.h”
...
...
Bool sunSpriteInitialize(ScreenPtr pScreen,
       Bool (*putInHardware)(),
       miPointerSpriteFuncPtr hardwareSpriteFuncs,
       miPointerScreenFuncPtr screenFuncs)
```

To make the sunSprite layer work, you must pass the sunSprite layer a set of four functions that implement a hardware cursor on your device (miPointerSpriteFuncPtr) and a function that is called by the sunSpriteLayer to check if a defined cursor should be put in hardware or software (putInHardware). An implementation of screenFuncs is already available:

```
#include “sun.h”
....
#include “mipointer.h”
....
....
extern miPointerScreenFuncRec sunPointerScreenFuncs;
```

The four functions that implement the hardware cursor and the putCursorInHardware function are needed to port to your hardware.

```
Bool xxxPutInHardware(ScreenPtr pScr, CursorPtr pCurs)
```

This function returns TRUE if the cursor should be placed in hardware; FALSE if the cursor should be drawn by software (miDC).

The following code is a sample implementation of this function on a device that has a 32x32 cursor register.

```
Bool
XXXPutInHardware(pScreen, pCursor)
    ScreenPtr pScreen;
    CursorPtr pCursor;
{
    if (pCursor->bits->width > 32 || pCursor->bits->height > 32)
        return FALSE;
    return TRUE;
}
```

Examples of miPointerSpriteFuncs

The following code is a sample pseudo-implementation of the four `miPointerSpriteFuncs` that implement a hardware cursor on the same device.

Code Example 4-1 Hardware Cursor Pseudocode

```
#include “sun.h”
#include “dixfontstr.h”
#include “mipointer.h”
#include “cursorstr.h”
#include “XXXhardware.h”
...
...
static Bool
XXXRealizeCursor (pScreen, pCursor)
    ScreenPtrpScreen;
    CursorPtrpCursor;
{
    pCursor->bits->devPriv[pScreen->myNum] = NULL;
    return TRUE;
}
static Bool
XXXUnrealizeCursor (pScreen, pCursor)
    ScreenPtrpScreen;
    CursorPtrpCursor;
{
    return TRUE;
}

/*
 * XXXLoadCursor -- Load the cursor into XXX hardware registers. When the
 * sunSprite layer is used, this routine is passed a cursor to install
 * into hardware only if the cursor fits into hardware (in this case <= 32x32).
 * However, just in case it is not the sunSprite layer calling this
 * routine, or if for DGA reasons you decide you want to force the cursor into
 * hardware regardless of its size, this routine is able to accept a
 * cursor larger than 33x32, trim it around the hotspot, and fit it into the
 * cursor register. You can either trim the cursor exactly around the
 * hotspot (bitBlt), or trim it so that you use the
 * 32-bit word of each scanline that the hotspot falls within. Do the latter
 * because it is faster. (The protocol says “The components of the cursor
 * can be transformed arbitrarily to meet display limitations...”)
 */
```

Code Example 4-1 Hardware Cursor Pseudocode *(Continued)*

```
static void
XXXLoadCursor (pScreen, pCursor, x, y)
    ScreenPtr  pScreen;
    CursorPtr  pCursor;
    int        x, y;
{
    SetupScreen(pScreen);
    int    w, h;
    Unsgn32 source[32], mask[32], *pSource, *pMask;
    int i;

    w = pCursor->bits->width;
    h = pCursor->bits->height;
    xhot = pCursor->bits->xhot;
    yhot = pCursor->bits->yhot;
    /* Assumes BITMAP_SCANLINE_PAD == 32 in the non-trim case */
    pSource = (Unsgn32 *)pCursor->bits->source;
    pMask = (Unsgn32 *)pCursor->bits->mask;

    /* Do I need to trim the cursor? */
    if (w > 32 || h > 32) { /* trim ! */
        int scanline = ((BitmapBytePad((int)(pCursor->bits->width))) >> 2);
        int startWord = 0, startscan = 0, endscan = h - 1;
        if (w > 32) {
            xhot = pCursor->bits->xhot % 32;
            startWord = pCursor->bits->xhot / 32;
            w = 32;
        }
        if (h > 32) {
            yhot = 16; /* easy to center around yhot */
            endscan = pCursor->bits->yhot + 15;
            while (endscan > h) {
            endscan--;
            yhot++;
            }
            startscan = endscan - 31;
            while (startscan < 0) {
            startscan++;
            yhot--;
            }
            h = 32;
        }
        pSource = pSource + startWord + startscan * scanline;
        pMask = pMask + startWord + startscan * scanline;
```

Code Example 4-1 Hardware Cursor Pseudocode *(Continued)*

```
        for (i = 0; i < h; i++) {
            source[i] = *pSource; pSource += scanline;
            mask[i] = *pMask; pMask += scanline;
        }
        pSource = source;
        pMask = mask;
    }

    /* By the time we reach this point, w <= 32 && h <=32 */

    /* Set the hardware cursor position and image here */
    /* This is where hardware-specific code is added... */
    XXXDOSETCURSORIMAGEANDPOSITION(pSource, pMask, x, y);
}

static void
XXXSetCursor (pScreen, pCursor, x, y)
    ScreenPtr  pScreen;
    CursorPtr  pCursor;
    int        x, y;
{

if (pCursor)
    XXXLoadCursor (pScreen, pCursor, x, y);
else
    XXXDisableCursor (pScreen);
}

static void
XXXMoveCursor (pScreen, x, y)
    ScreenPtr  pScreen;
    int        x, y;
{
    XXXMOVECURSOR((((x - xhot) << 16) | (( y - yhot) & 0xffff)));
}

static void
XXXQueryBestSize (class, pwidth, pheight, pScreen)
    int        class;
    short      *pwidth, *pheight;
    ScreenPtr  pScreen;
{
```

Code Example 4-1 Hardware Cursor Pseudocode *(Continued)*

```
switch (class)
    {
    case CursorShape:
        if (*pwidth > 32)
            *pwidth = 32;
        if (*pheight > 32)
            *pheight = 32;
        break;
    default:
        mfbQueryBestSize (class, pwidth, pheight, pScreen);
        break;
    }
}

static miPointerSpriteFuncRec XXXPointerSpriteFuncs = {
    XXXRealizeCursor,
    XXXUnrealizeCursor,
    XXXSetCursor,
    XXXMoveCursor,
};

/*
 * This function is called from the DDX handler's Screen Init routine. */
void
XXXCursorInitialize (pScreen)
    ScreenPtrpScreen;
{
    extern miPointerScreenFuncRec sunPointerScreenFuncs;

pScreen->QueryBestSize = XXXQueryBestSize;
    sunSpriteInitialize (pScreen, XXXPutInHardware,
        &XXXPointerSpriteFuncs,
        &sunPointerScreenFuncs);
}

void
XXXDisableCursor (pScreen)
    ScreenPtrpScreen;
{
    XXXSWITCHOFFCURSOR();
}
```

Kernel Cursor Tracking - The sunHWCursor Layer

The preceding section outlined examples of a hardware cursor implementation in which the hardware cursor was tracked by the X server process—that is, the cursor position was updated in user-domain code. Under conditions of heavy system load, this approach of tracking the cursor in the X server process might result in a considerable latency between pointer motion and corresponding cursor motion on the screen. One way to improve the interactive performance of the cursor is to track the cursor in the kernel-domain.

The sunHWCursor layer offers an implementation of a hardware cursor that is tracked in the kernel. To use this layer, the device driver for your graphics adapter must implement a set of kernel cursor tracking `ioctl`s that are documented in *Writing Device Drivers*. If your device driver implements these `ioctl`s, and you use the sunHWCursor layer utilities for your cursor implementation, a module (called *hwc*) is pushed on the mouse stream that intercepts mouse events and sends them directly to the graphics adapter's device driver via the Kernel Cursor Tracking `ioctl`s issued from the kernel-domain.

Additionally, the sunHWCursor implementation is layered over the sunSprite layer. This means that when this layer is used for your cursor implementation, the cursor switches to a software form (tracked in the user-domain) over windows that define a cursor that is too large to fit in the hardware cursor image registers.

The sunHWCursor code is initialized in the DDX handler's Screen initialization function by calling the following function:

```
#include "sun.h"
...
...
Bool sunCursorInitialize(ScreenPtr pScreen)
```

`sunCursorInitialize` initializes `pScreen->QueryBestSize` with `sunQueryBestSize`, and then calls `sunSpriteInitialize`. As mentioned in "The sunSprite Layer" on page 35, the sunSprite layer requires an implementation of the `PutInHardware`, `hardwareSpriteFuncs` and `screenFuncs` functions.

Note – In this release, the ability to specialize these functions for the sunSprite layer is not available when using the sunHWCursor layer; the sunHWCursor layer has built-in implementations of these functions and the `sunQueryBestSize` function. The ability to specialize some of these functions when using the sunHWCursor layer might be offered in a future release of the OpenWindows server.

Invoking `sunCursorInitialize` in your DDX handler's initialization routine, and implementing the `ioctl`s in the device driver is sufficient to obtain a kernel-tracked cursor. If you are in a hurry to get a kernel-tracked hardware cursor implementation going on your graphics adapter, you do not need to know all of the sunHWCursor layer details that follow.

sunHWCursor Functions

The functions provided in the sunHWCursor layer are described in this section.

sunQueryBestSize

```
static void sunQueryBestSize(int class, short *pWidth,
       short *pHeight, ScreenPtr pScreen)
```

Results If `class` is `CursorShape`, this function issues an `ioctl` to the device driver to determine the maximum hardware cursor size. For all other values of `class`, this function calls `mfbQueryBestSize`.

Returns If the hardware cursor size is smaller than the maximum screen bounds, this function returns these values in `pWidth` and `pHeight`, else it returns the maximum screen bounds.

If this implementation of `pScreen->QueryBestSize` is not desired, supply an equivalent function in your DDX handler after `sunCursorInitialize` has been called.

sunPutInHardware

```
static Bool sunPutInHardware(ScreenPtr pScreen,
       CursorPtr *pCursor)
```

Purpose This function is the sunHWCursor layer's implementation of the `PutInHardware` routine required by the sunSprite layer.

Results This function issues an `ioctl` to the device driver to determine the maximum hardware cursor size.

Returns If the cursor passed in `pCursor` is larger than the hardware size, this function returns `FALSE`, else it returns `TRUE`.

screenFuncs

```
extern miPointerScreenFuncRec sunPointerScreenFuncs;
```

Purpose This is an implementation of the `screenFuncs` functions that is passed to the sunSprite layer. See "miPointer Layer" on page 32.

hardwareSpriteFuncs

```
miPointerSpriteFuncRec sunPointerSpriteFuncs = {
       sunRealizeCursor, sunUnRealizeCursor, sunSetCursor,
       sunMoveCursor,
};
```

Purpose This is the sunHWCursor layer's implementation of the `hardwareSpriteFuncs` array required by the sunSprite layer. These functions load the hardware cursor, and enable kernel cursor tracking via the *hwc* module that has been pushed onto the mouse stream. The `sunMoveCursor` function is a stub that does not get called while kernel cursor tracking is active. If the cursor is switched to a software form by the sunSprite layer (this might happen when the pointer

traverses a window that has a large cursor defined, which does not fit in the hardware cursor image registers), the cursor is tracked in user-domain by the `miDC` layer.

Multiple Plane Group Interface 5

Some devices contain multiple plane groups (MPG) to support overlays and visuals of varying depths. The MPG utility library provides the following features for those devices:

- Windowing Operations

  These functions are necessary to operate on windows with multiple plane groups. When a window is moved, all of its physical plane groups need to be moved; when a window is exposed, all of its damaged plane groups need to be repaired.

- Minimizing Exposure Events

  These functions minimize exposure events between windows that reside in separate plane groups. See “CopyPlanes and AggregatePlanes” on page 61 for more information.

- Leveraging of Existing DDX Interfaces

  MPG is designed to use existing rendering and windowing libraries, such as CFB or MFB.

MPG Architectural Overview

MPG is data-driven; DDX handlers need to inform MPG which plane groups are used by which windows and how they are used within the windows. Then the MPG windowing operations take care of moving, preparing and computing exposures to the plane groups.

Figure 5-1 shows the MPG library's interfaces to other DDX utility libraries.

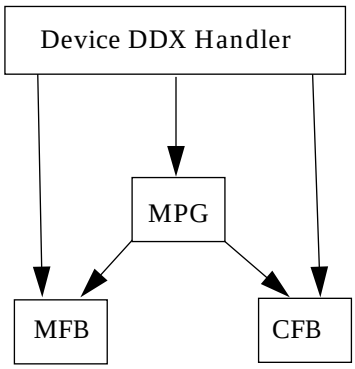


Figure 5-1 MPG DDX Library Interfaces

The MPG DDX library does not actually do any rendering. Instead, it is designed to lie on top of other DDX libraries, such as CFB and MFB or device-specific code, which provide all of the rendering and some of the windowing functions. This way a frame buffer with a 24-bit color plane group and a 1-bit overlay plane group can use CFB32 and MFB for its depth-specific rendering and windowing functions. MPG manages the depth-specific setup and switching between the underlying DDX libraries, and provides the rest of the windowing functions. MPG does not explicitly call CFB or MFB, and can use any device-specific functions.

Each physical plane group requires a *screen pixmap*, which is a pixmap structure that points to an on-screen data area. Each window uses one or more plane groups. Two windows can share the same plane group, but use it differently.

The *MPG info* of a window is comprised of its plane group combination and usage. The MPG info is stored in the `mpgInfoRec` structure that may be shared among windows. The flavor of a window is defined by its MPG info and visual. There is a one-to-many relationship between MPG infos and visuals. A sample device, such as the CG8, might have:

- three plane groups: 24-bit color, 1-bit overlay, 1-bit overlay enable

and might provide:

- two MPG infos: color underlay and monochrome overlay, and
- three visuals: `StaticGray`, `TrueColor`, and `DirectColor`

In the above example, windows with `TrueColor` or `DirectColor` visuals share the same color underlay MPG info. Each supported visual is matched by an MPG info in the `mpgPerVisInfo` structure. Each window is assigned to an MPG info based on its visual.

Data Structure Initialization

In a single plane group (SPG) device, some members of the screen structure apply to only a single depth. In an MPG device that supports various depths, this depth-specific information must be stored somewhere else. Currently, most of this information is stored in the `mpgInfoRec` structure; the rest stored in the `mpgPerDepthInfo` structure which is arranged by depth. Pointers to all `mpgInfoRec` structures are listed in the `mpgPerVisInfo` structure arranged by visual.

The `mpgPerVisInfo` and `mpgPerDepthInfo` structures are initialized directly in the device's DDX handler and attached to the screen private structure via the `mpgScreenInit` function. Each `mpgInfoRec` structure is

initialized indirectly via `mpgGetScreenState` and `mpgInsertPlanegroup` functions. See “MPG Functional Interface” for a detailed description of these functions.

Code Example 5-1 MPG Data Structure Direct Initialization

```
#define NUMVISUALS 3
#define NUMVISUALS1 1
#define NUMVISUALS24 2
#define NUMDEPTHS 2 /* 1 and 24 bit */

static mpgInfoRec overlay_info, color_info;

static mpgPerVisInfo cg8MPGPerVisInfo[NUMVISUALS] = {
    (VisualID)0, &overlay_info,
    (VisualID)0, &color_info,
    (VisualID)0, &color_info,
};

static const mpgPerDepthInfo cg8MPGPerDepthInfo[NUMDEPTHS] = {
    {1, mfbCreateGC, mfbCreatePixmap, mfbDestroyPixmap,
        mfbGetImage, mfbGetSpans},
    {24, cfb32CreateGC, cfb32CreatePixmap, cfb32DestroyPixmap,
        cfb32GetImage, cfb32GetSpans}
};
```

MPG Functional Interface

initPixmap

```
void
initPixmap(ScreenPtr pScreen, int width, int height,
       int linebytes, int depth,
       PixmapPtr pScreenPixmap, pointer data)
```

Purpose This function initializes the screen pixmap of a plane group.

Arguments `width`, `height` and `depth` are the plane group dimensions.

`linebytes` is the number of bytes to pad a scan line on the plane group of a given `width` and `depth`.

`data` is a pointer to a memory-mapped on-screen data area that is used to initialize the `devPrivate` field of the screen pixmap.

The following code shows you a few samples of how to use `initPixmap`.

Code Example 5-2 initPixmap

```
initPixmap(pScreen, width, height, PixmapBytePad(width, 1), 1,
    &cg8Private->pixmaps[CG8_ENABLE], overlay_enable_data);
initPixmap(pScreen, width, height, PixmapBytePad(width, 1), 1,
    &cg8Private->pixmaps[CG8_OVERLAY], overlay_data);
initPixmap(pScreen, width, height, PixmapBytePad(width, 24), 24,
    &cg8Private->pixmaps[CG8_COLOR_24], color_data);
```

mpgGetScreenState

```
Bool
mpgGetScreenState(ScreenPtr pScreen, mpgInfoPtr pMPGInfo,
       void (*SetupScreen)(), miBSFuncPtr pBSFuncs)
```

Purpose This function stores depth-specific information about the screen in the `mpgInfoRec` structure pointed to by `pMPGInfo`. It stores the `blackPixel` and `whitePixel` values, a set of depth-specific screen functions, a plane group-specific `SetupScreen` function, and a set of depth-specific backing store functions pointed to by `pBSFuncs`.

The following depth-specific screen functions are currently stored by `mpgGetScreenState`:

- `GetImage`
- `GetSpans`
- `ResolveColor`
- `CreateColormap`
- `DestroyColormap`
- `CopyWindow`
- `CreateWindow`
- `DestroyWindow`
- `RealizeWindow`
- `PositionWindow`
- `UnrealizeWindow`

- PaintWindowBorder
- PaintWindowBackground
- ChangeWindowAttributes

SetupScreen

```
void
(* SetupScreen)(ScreenPtr pScreen)
```

Purpose This function normally initializes the devPrivate field of the screen structure to point to the screen pixmap of a specific plane group. It may also perform other software set up for rendering on that specific plane group.

The following code shows you a few samples of how to set up screens.

Code Example 5-3 SetupScreen

```
static void
cg8MFBSetup(ScreenPtr pScreen)
{
    pScreen->devPrivate = (pointer)&pCG8Private->pixmaps[CG8_OVERLAY];
}

static void
cg8CFB32Setup(ScreenPtr pScreen)
{
    pScreen->devPrivate = (pointer)&pCG8Private->pixmaps[CG8_COLOR_24];
    pScreen->devPrivates[cfb32ScreenPrivateIndex].ptr = pScreen->devPrivate;
}
```

mpgGetScreenState extracts most of its information from the current state of the screen. Do not over-initialize the screen before calling mpgGetScreenState. Routines like mfbScreenInit and cfbScreenInit usually do too much, such as bringing in much of the MI library that might not be necessary or allocating a lot of redundant memory. Use routines like mfbSetupScreen and cfbSetupScreen instead.

The following code shows you a few samples of how to get the screen state.

Code Example 5-4 mpgGetScreenState

```
mfbSetupScreen(pScreen, pCG8Private->pixmaps[CG8_OVERLAY].devPrivate,
    pScreen->width, pScreen->height, monitorResolution,
    monitorResolution, pScreen->width);
mpgGetScreenState(pScreen, &overlay_info, cg8MFBSetup,
    &mfbBSFuncRec);
cfb32SetupScreen(pScreen,
    pCG8Private->pixmaps[CG8_COLOR_24].devPrivate, pScreen->width,
    pScreen->height, monitorResolution, monitorResolution,
    pScreen->width);
mpgGetScreenState(pScreen, &color_info, cg8CFB32Setup,
    &cfb32BSFuncRec);
```

`mpgGetScreenState` returns `TRUE` if it's successful, `FALSE` otherwise.

mpgInsertPlanegroup

```
Bool
mpgInsertPlanegroup(mpgInfoPtr pMPGInfo, mpgPlaneId iid,
       mpgPlaneId eid, mpgType type, mpgOp op, unsigned long val)
```

Purpose This function builds the MPG info by filling the `mpgInfoRec` structure pointed to by `pMPGInfo` with information on plane group combination and usage.

Arguments `iid` and `eid` are the plane group internal and external identifiers. Plane group identifiers are unique small integers. Each device can enumerate its own plane groups to uniquely identify them. Plane group identifiers are normally used to index arrays of screen pixmaps. They are also bit-encoded and combined together to create plane group bit masks that express the plane group combination in each window and

facilitate the plane group interaction among windows. MPG provides the following macros to create and perform set operations on plane group bit masks:

```
#define mpg_bit_encoded(i) (1<<(i))
#define mpg_union(a,b) ((a)|(b))
#define mpg_intersect(a,b) ((a)&(b))
#define mpg_subtract(a,b) ((a)&(~(b)))
#define mpg_subset(a,b) ((a)==((a)&(b)))
```

Currently the bit-encoding scheme limits plane group identifiers to be between 0 and 31 inclusive. `iid` is used to represent a plane group internally within the window, while `eid` is used to represent a plane group externally with respect to other windows. For example, `iid` is used in rendering and preparing plane groups in each window, while `eid` is used in checking plane group interference among windows and moving a family of windows across the screen. Windows that share the same `eid` damage each other on that plane group. Normally the `eid` of a plane group is identical to its `iid`. For backward compatibility, entering 0 for the `eid` currently forces it to be identical to the `iid`.

`type` describes the usage of each plane group within its window. Entering `MPG_VISIBLE` for `type` means the plane group is used for describing visibility. Entering `MPG_DRAWABLE` for `type` means the plane group is used for client rendering or to assist client rendering, for example, as the Z buffer in 3D rendering or the WID (window ID) buffer in hardware clipping. (See Chapter 7, “Window ID Interface” for detailed information on WIDs.) Entering `MPG_VISIBLE_DRAWABLE` for `type` means the plane group is used for all of the purposes stated above. Each window has one plane group of type `MPG_VISIBLE` or `MPG_VISIBLE_DRAWABLE` to describe visibility. Entering `MPG_OTHER` for `type` means the plane group is used for purposes other than the ones stated above, such as clearing buffers or switching colormaps.

Each plane group with a unique `eid` has a region that represents the area of the screen pixmap claimed by its window with respect to other windows. The region of a plane group of type `MPG_VISIBLE` or `MPG_VISIBLE_DRAWABLE` is used in processing `VisibilityNotify` events—it is used to describe if its window is unobstructed, fully obscured, or partially obscured by other windows that share the same plane group. The region of a plane group of type `MPG_DRAWABLE` or

MPG_VISIBLE_DRAWABLE is used in processing Expose events—it is used to compute the effective rendering clip of its window. A window does not receive an Expose event until all of its plane groups of type MPG_DRAWABLE or MPG_VISIBLE_DRAWABLE are exposed.

op is performed on each plane group when it is exposed. Entering MPG_NOOP for op means the plane group is not filled or rendered—it does not contain data. A plane group with MPG_NOOP operation can be viewed as a virtual plane group. It is normally used to force interference among windows with different plane group combinations. A virtual plane group is not copied when its window is moved.

Entering MPG_DRAW for op means the plane group is rendered by clients—it contains data. Multiple plane groups can have the MPG_DRAW operation. The last plane group inserted is the *drawing* plane group. The iid of this plane group is used to render color data.

Note – In the current release, use MPG_DRAW with plane groups of type MPG_DRAWABLE or MPG_VISIBLE_DRAWABLE.

Entering MPG_FILL for op means the plane group is filled with the value supplied in val, which is constant throughout the window's existence. Entering MPG_FILL_WID for op means the plane group is filled with the *window id* value associated with its window. Window ids are a finite resource that can be shared and rotated among windows.

val is the value to fill the plane group with when op is MPG_FILL. It is ignored for all other cases.

Plane Group Aliasing

In addition to supporting plane groups with multiple purposes, MPG also supports multiple ways of addressing them. MPG allows *plane group aliasing*—the ability to address a plane group partially, internal or external to the window. This enables a plane group to be split into several disjoint partitions or aggregated with other plane groups to form a larger cohesive entity. For example, a 24-bit color plane group is internally addressed as an 8-bit color plane group to support 8-bit windows, or is split into three disjoint 8-bit color

plane groups, in which mutually non-interfering 8-bit windows coexist. Enter a different `iid` and `eid` per plane group with `mpgInsertPlanegroup` to use plane group aliasing.

Note – Currently a one-to-many relationship between `iid`s and `eid`s in each window is supported.

The following examples show you how to implement plane group aliasing with `mpgInsertPlanegroup`. Each example gets more complex—the first example shows the most common ways to plane group alias, while the last example shows a disjointed plane group.

Code Example 5-5 Common use of mpgInsertPlanegroup

```
mpgInsertPlanegroup(&overlay_info, CG8_OVERLAY, CG8_OVERLAY,
    MPG_DRAWABLE, MPG_DRAW, 0);
mpgInsertPlanegroup(&overlay_info, CG8_ENABLE, CG8_ENABLE,
    MPG_VISIBLE, MPG_FILL, 1);

mpgInsertPlanegroup(&color_info, CG8_COLOR_24, CG8_COLOR_24,
    MPG_DRAWABLE, MPG_DRAW, 0);
mpgInsertPlanegroup(&color_info, CG8_OVERLAY, CG8_OVERLAY,
    MPG_OTHER, MPG_FILL, 0);
mpgInsertPlanegroup(&color_info, CG8_ENABLE, CG8_ENABLE,
    MPG_VISIBLE, MPG_FILL, 0);
```

Code Example 5-6 Complex use of mpgInsertPlanegroup

```
mpgInsertPlanegroup(&overlay_info, CG8_OVERLAY, CG8_OVERLAY,
    MPG_DRAWABLE, MPG_DRAW, 0);
mpgInsertPlanegroup(&overlay_info, CG8_ENABLE, CG8_ENABLE,
    MPG_VISIBLE, MPG_FILL, 1);

mpgInsertPlanegroup(&color8_info, CG8_COLOR_8, CG8_COLOR_24,
    MPG_DRAWABLE, MPG_DRAW, 0);
mpgInsertPlanegroup(&color8_info, CG8_OVERLAY, CG8_OVERLAY,
    MPG_OTHER, MPG_FILL, 0);
mpgInsertPlanegroup(&color8_info, CG8_ENABLE, CG8_ENABLE,
    MPG_VISIBLE, MPG_FILL, 0);

mpgInsertPlanegroup(&color24_info, CG8_COLOR_24, CG8_COLOR_24,
    MPG_DRAWABLE, MPG_DRAW, 0);
mpgInsertPlanegroup(&color24_info, CG8_OVERLAY, CG8_OVERLAY,
    MPG_OTHER, MPG_FILL, 0);
mpgInsertPlanegroup(&color24_info, CG8_ENABLE, CG8_ENABLE,
    MPG_VISIBLE, MPG_FILL, 0);
```

Code Example 5-7 More Complex use of mpgInsertPlanegroup

```
mpgInsertPlanegroup(&overlay_info, CG8_OVERLAY, CG8_OVERLAY,
    MPG_DRAWABLE, MPG_DRAW, 0);
mpgInsertPlanegroup(&overlay_info, CG8_ENABLE, CG8_ENABLE,
    MPG_VISIBLE, MPG_FILL, 1);

mpgInsertPlanegroup(&color8A_info, CG8_COLOR_8A, CG8_COLOR_8A,
    MPG_DRAWABLE, MPG_DRAW, 0);
mpgInsertPlanegroup(&color8A_info, CG8_OVERLAY, CG8_OVERLAY,
    MPG_OTHER, MPG_FILL, 0);
mpgInsertPlanegroup(&color8A_info, CG8_ENABLE, CG8_ENABLE,
    MPG_VISIBLE, MPG_FILL, 0);

mpgInsertPlanegroup(&color8B_info, CG8_COLOR_8B, CG8_COLOR_8B,
    MPG_DRAWABLE, MPG_DRAW, 0);
mpgInsertPlanegroup(&color8B_info, CG8_OVERLAY, CG8_OVERLAY,
    MPG_OTHER, MPG_FILL, 0);
mpgInsertPlanegroup(&color8B_info, CG8_ENABLE, CG8_ENABLE,
    MPG_VISIBLE, MPG_FILL, 0);

mpgInsertPlanegroup(&color24_info, CG8_COLOR_24, CG8_COLOR_8A,
    MPG_DRAWABLE, MPG_DRAW, 0);
mpgInsertPlanegroup(&color24_info, CG8_COLOR_24, CG8_COLOR_8B,
    MPG_DRAWABLE, MPG_DRAW, 0);
mpgInsertPlanegroup(&color24_info, CG8_COLOR_24, CG8_COLOR_8C,
    MPG_DRAWABLE, MPG_DRAW, 0);
mpgInsertPlanegroup(&color24_info, CG8_OVERLAY, CG8_OVERLAY,
    MPG_OTHER, MPG_FILL, 0);
mpgInsertPlanegroup(&color24_info, CG8_ENABLE, CG8_ENABLE,
    MPG_VISIBLE, MPG_FILL, 0);
```

`mpgInsertPlanegroup` returns `TRUE` if successful, `FALSE` otherwise.

mpgScreenInit

```
Bool
mpgScreenInit(ScreenPtr pScreen, int numPlanes,
       PixmapPtr pScreenPixmaps, mpgPlanes dispPlanes,
       mpgPerVisInfoPtr pMPGPerVisInfo,
       mpgPerDepthInfo pMPGPerDepthInfo,
       void (* SwitchScreen)());
```

Purpose This function completes the MPG screen initialization.

Arguments

`numPlanes` is the total number of plane groups in the device.

`pScreenPixmaps` is a pointer to an array of screen pixmaps.

`dispPlanes` is the displayable plane groups in the device. Displayable plane groups are plane groups that are visible at one time or another on the screen. For example, in CG8, the 24-bit color and 1-bit overlay plane groups are displayable, but not the 1-bit overlay enable plane group. `dispPlanes` is entered as a plane group bit mask, created by combining bit-encoded displayable plane group identifiers.

`pMPGPerVisInfo` is a pointer to the `mpgPerVisInfo` structure, which is an arranged-by-visual array of MPG infos.

`pMPGPerDepthInfo` is a pointer to the `mpgPerDepthInfo` structure, which is an arranged-by-depth array of depth-specific screen functions.

SwitchScreen

```
void
(* SwitchScreen)(ScreenPtr pScreen, mpgPlaneId pid)
```

Purpose This function is a pointer to a function that performs the hardware set up for rendering on a specific plane group. Entering `NULL` means the device does not need it. `pid` is the identifier of a plane group to which the screen has to be switched.

Returns `TRUE` if successful; `FALSE` otherwise

The following fields in the screen structure should be initialized before calling `mpgScreenInit`:

- `visuals`
- `numDepths`
- `numVisuals`
- `CloseScreen`
- `allowedDepths`

The following code shows you a sample of how to use `mpgScreenInit`.

```
mpgScreenInit(pScreen, NUM_CG8_PLANEGROUPS, pCG8Private->pixmaps,
       mpg_union(mpg_bit_encoded(CG8_OVERLAY),
       mpg_bit_encoded(CG8_COLOR_24)), cg8MPGPerVisInfo,
       cg8MPGPerDepthInfo, NULL);
```

Note – The initialization order for devices that use *both* MPG and DGA is: MPG, DGA, and then the screen pixmap `devPrivates` at the end of your DDX handler initialization,

getMpgInfoFromVisual

```
mpgInfoPtr
getMpgInfoFromVisual(ScreenPtr pScreen, VisualID vid)
```

Purpose This function uses `vid` to search the arranged-by-visual `mpgPerVisInfo` structure, which is attached to the screen private structure.

Returns A pointer to the matching `mpgInfoRec` structure.

mpgChangeInfo

```
void
mpgChangeInfo(WindowPtr pWin, mpgInfoPtr pNewMPGInfo)
```

Purpose This function replaces the MPG info of a window with a new `mpgInfoRec` structure pointed to by `pNewMPGInfo`. It can be used to change the flavor of a window at any given time. Changing the MPG info is similar to adding, subtracting, or replacing plane groups, or changing their types and operations.

The following code shows you a sample of how to use `mpgChangeInfo`.

```
/* migrate pWin from 8-bit color plane group A to 8-bit color */
/* plane group B */
if (getMpgInfoFromVisual(pScreen, pWin->optional->visual) ==
       &color8A_info)
   mpgChangeInfo(pWin, &color8B_info);
```

freeMpgInfo

```
void
freeMpgInfo(mpgInfoPtr pMPGInfo)
```

Purpose This function frees the memory associated with the `mpgInfoRec` structure pointed to by `pMPGInfo`, but not the structure itself. The freed memory has been previously allocated by `mpgGetScreenState` and `mpgInsertPlanegroup`.

The following code shows you a few samples of how to use `freeMpgInfo`.

```
freeMpgInfo(&overlay_info);
freeMpgInfo(&color_info);
```

mpgCursorInitialize

```
Bool
mpgCursorInitialize(ScreenPtr pScreen,
       mpgPlaneId cid, mpgPlaneId eid, Bool isDedicated)
```

Purpose This function sets up the screen to use the MPG software cursor. If the device has a hardware cursor there is no need to call `mpgCursorInitialize`.

Arguments `cid` is the identifier for the cursor plane group, on which the cursor image is rendered with the default foreground and background colors of 1 and 0, respectively.

`eid` is the identifier for the cursor enable plane group, on which the cursor mask is filled with the default value of 1.

`isDedicated` is `TRUE` if the cursor and the cursor enable plane groups are dedicated to the cursor and not used by any window. Otherwise, MPG has to lift the cursor for any conflicting rendering operation and drop it again afterwards.

Returns `TRUE` if successful, `FALSE` otherwise

mpgSetCursorValues

```
void
mpgSetCursorValues(ScreenPtr pScreen, unsigned long eval,
       unsigned long fval, unsigned long bval)
```

Purpose This function resets the cursor enable plane group's fill value, the cursor's foreground color, and the cursor's background color with `eval`, `fval` and `bval`, respectively.

mpgSetCursorHasEnable

```
void
mpgSetCursorHasEnable(ScreenPtr pScreen, Bool hasEnable)
```

Purpose This function resets the need for the cursor enable plane group.

Arguments `hasEnable` is `FALSE` if the cursor enable plane group is not needed.

The following code shows you a sample of how to use `mpgSetCursorHasEnable`.

```
mpgCursorInitialize(pScreen, CG8_OVERLAY, CG8_ENABLE, FALSE);
mpgSetCursorValues(pScreen, 1, 0, 1);/* reverse */
mpgSetCursorHasEnable(pScreen, FALSE);
```

CopyPlanes and AggregatePlanes

To minimize window exposures, MPG wraps, or replaces the existing X windowing screen functions. For example, it cannot use the basic `CopyWindow` screen function for moving a family of windows with various depths and other attributes across the screen, since this operation involves copying different regions on several plane groups. Instead, it allocates two function pointers in the MPG screen private structure, `CopyPlanes` and `AggregatePlanes`, and uses them. `AggregatePlanes` is a complement to `CopyPlanes`, and is called

inside any CopyPlanes implementation. AggregatePlanes notifies CopyPlanes if the device can copy several plane groups simultaneously, so that CopyPlanes adjusts accordingly and improves its performance; otherwise, CopyPlanes copies those plane groups one-by-one.

CopyPlanes

```
void
(* CopyPlanes)(ScreenPtr pScreen, WindowPtr pWin,
       RegionPtr pRegions[], mpgPlanes planes, int dx, int dy)
```

Note – MPG provides a generic implementation of CopyPlanes in mpgCopyPlanes. It is highly recommended that you use mpgCopyPlanes directly, or wrap it in conjunction with AggregatePlanes, instead of providing your own implementations.

Arguments

pWin is a pointer to the highest window in the window subtree being moved—it is the root of the subtree. Currently it serves as a flag to override AggregatePlanes. When pWin is NULL, CopyPlanes still copies plane groups one at a time, even though AggregatePlanes insists that the device is capable of copying them simultaneously. In mpgCopyPlanes, pWin is used as a starting point to repair the damage on the window subtree being moved that may be caused by copying plane groups simultaneously.

pRegions is a pointer to an indexed-by-plane group array of regions to be copied. These regions often differ from each other.

planes is a plane group bit mask indicating which entries are valid in the array of regions pointed to by pRegions.

dx and dy are the horizontal and vertical distances to copy those regions on their plane groups.

AggregatePlanes

```
int
(* AggregatePlanes)(ScreenPtr pScreen, mpgPlanes planes)
```

Purpose — MPG does not provide a generic implementation of `AggregatePlanes`. By default, `mpgCopyPlanes` copies plane groups one-by-one. Providing an implementation of `AggregatePlanes` and attaching it to the screen private structure are sufficient to allow `mpgCopyPlanes` to copy plane groups simultaneously. Some devices might also need to wrap `mpgCopyPlanes`.

Arguments — `planes` is a plane group bit mask indicating which plane groups have regions to be copied.

Returns — A plane group identifier representing the aggregate of all plane groups in `planes` if they can be aggregated; a negative number otherwise.

Note – Currently `CopyPlanes` and `AggregatePlanes` are initialized by `mpgScreenInit` to `mpgCopyPlanes` and `NULL`, respectively. These default function assignments should be sufficient for a lot of devices.

When a device needs to reset `AggregatePlanes`, wrap `mpgCopyPlanes` or implement your own `CopyPlanes`,

MPG provides a macro, `mpg_priv_scr`, to access the screen private structure:

```
#define mpg_priv_scr(pScreen) ((mpgPrivScreenPtr)(
(pScreen)->devPrivates[mpgScreenPrivateIndex].ptr))
       (pScreen)->devPrivates[mpgScreenPrivateIndex].ptr))
```

The following code shows you samples of how to use `CopyPlanes` and `AggregatePlanes`.

Code Example 5-8 CopyPlanes and AggregatePlanes

```
/* after calling mpgScreenInit, wrap mpgCopyPlanes and initialize */
/* AggregatePlanes */
{
mpgPrivScreenPtr pMPGPrivScreen = mpg_priv_scr(pScreen);
    pMPGPrivScreen->CopyPlanes = cg8CopyPlanes;
    pMPGPrivScreen->AggregatePlanes = cg8AggregatePlanes;
}
int
cg8AggregatePlanes(ScreenPtr pScreen, mpgPlanes planes)
{
    switch (planes) {
        case mpg_union(mpg_bit_encoded(CG8_COLOR_8A),
            mpg_union(mpg_bit_encoded(CG8_COLOR_8B),
                mpg_bit_encoded(CG8_COLOR_8C))):
            return CG8_COLOR_24;
        default:
            return -1;
    }
}

void
cg8CopyPlanes(ScreenPtr pScreen, WindowPtr pWin,
        RegionPtr pRegions[], mpgPlanes planes, int dx, int dy)
{
    mpgPlanes plns = mpg_union(mpg_bit_encoded(CG8_COLOR_8A),
    mpg_union(mpg_bit_encoded(CG8_COLOR_8B),
    mpg_bit_encoded(CG8_COLOR_8C)));

    if (mpg_subset(plns, planes)) {
        mpgCopyPlanes(pScreen, pWin, pRegions, plns, dx, dy);
        mpgCopyPlanes(pScreen, pWin, pRegions,
            pg_subtract(planes, plns), dx, dy);
    } else
        mpgCopyPlanes(pScreen, pWin, pRegions, planes, dx, dy);
}
```

mpgSetScreenFuncs

```
long
mpgSetScreenFuncs(pScreen, funcs, mask, oldfuncs)
```

Purpose This function allows the device developer to supply an arbitrary number of wrapper functions.

Arguments `funcs` is a structure containing the wrapper functions.

`mask` indicating which of the wrapper functions is valid.

`oldfuncs` contains previous wrapper functions.

Returns The previous values of the indicated function vectors so that devices may make use of the more generalized default implementation to handle the more obscure cases of the particular function they are wrapping.

The `mpgSetScreenFuncs()` function examines the `mask` parameter to determine which functions are being wrapped. For each wrapper indicated, this function stores the previous wrapper function (or NULL if there was no default value) into the appropriate member of the `oldfuncs` structure (if supplied) and then loads the new wrapper function from the appropriate member of the `funcs` structure into the internal MPG function vector.

The `oldfuncs` parameter may be NULL if the device does not need to refer to the previous versions of any of the functions which it is overriding. The `oldfuncs` parameter may also be a pointer to the same structure as the `funcs` parameter, in which case `mpgSetScreenFuncs()` safely swaps the two function values.

5

Overlay Window Interface 6

This chapter discusses the Overlay window (OVL) graphics programming interface (GPI). It includes an introduction, how to setup your device, how to initialize overlays, and defines all of the functions and data types in this interface.

Introduction

The OpenWindows server provides the basic infrastructure for the OVL GPI in the OVL package. Your X11 client can create and configure overlay windows, and use backing store and gravity. These features are exported by the X11 client libraries `libX11` (the core Xlib library) and `libXext` (the Xlib extension library).

In addition to overlay window manipulation, the server provides a means for rendering transparent pixels into overlay windows. An extension routine that specifies an X11 GC *paint type* attribute is provided. The behavior of the core X11 rendering routines is extended to use this attribute while rendering. For more specific information, see the *Solaris X Window System Developer's Guide* which is part of the SDK (Software Developer's Kit).

These capabilities are made available on all device types. However, some devices can *optimize* the overlay window manipulation and rendering. This is exported to the client through a visual in the screen's list of visuals. The client then creates *optimal* overlay windows on these visuals. However, the client still

needs to know what is the *best* visual to use as a matching overlay/underlay visual for the exported visual. The Overlay Window API provides this information, but the server gets this information from the device.

Also, some devices specify their own functions to process the requests in the overlay extension. This interface, called the Overlay GPI, presents a solution to these problems.

Note – The OVL package is dependent on the Multiple Plane Group (MPG) package (see Chapter 5, "Multiple Plane Group Interface").

Device Setup

The OpenWindows server fully implements overlay windows and renders transparency. Device setup for overlay windows is done with the MPG package. This section provides examples of different device types and how to set them up for optimal performance.

The four basic types of devices are as follows.

1. Transparent Pixel

   The transparent pixel device renders into a drawable plane group with a special value to provide transparency. The special value causes a different drawable plane group to show through.

2. Control Plane Group

   The control plane group device has a special plane group that specifies which *drawable* plane group is currently visible. This plane group is often referred to as the control plane group. It could be a 1-bit enable plane, a multi-bit WID plane group, or some other type of control plane group.

3. Shared

   The shared device has the overlay windows and the underlay windows coexisting in the same drawable plane group.

4. Custom

   The custom device is different than the above device types—it could be a device with some or all overlay and underlay plane groups are not memory mapped, or a device that can render into image and control plane groups simultaneously.

Overlay window processing and rendering transparency is dependent on how the devices different physical plane groups are presented to MPG. In general, rendering transparency can be thought of as making the window *behind* the overlay window visible. So, all mpg setup should follow the guideline of attaching all plane groups to an MPG info structure that would allow a window associated with that MPG info to be visible. In the following sections, each device type is presented with the appropriate plane group partitioning that would facilitate overlay window processing and rendering transparency.

Transparent Pixel

A transparent pixel device has the following plane groups:

- a 24-bit drawing plane group (DRAW_A),
- an 8-bit drawing plane group (DRAW_B), and
- another 8-bit drawing plane group that can render transparency by rendering one of several set pixel values (OVERLAY).

Also, a given transparent pixel value may be different depending on what plane group is expected to *show through*. For DRAW_A, the pixel value is 254 and for DRAW_B, the pixel value is 255. The question now is what should the mpg setup look like.

The transparent pixel device has three MPG infos. The overlay MPG info has just the OVERLAY plane group with a type of `MPG_VISIBLE_DRAWABLE` and an `op` of `MPG_DRAW`. The other two MPG infos have specific `MPG_DRAWABLE` plane groups and an OVERLAY plane group as well; however, the OVERLAY plane group is of type `MPG_VISIBLE` and the `op` is `MPG_FILL`. For DRAW_A, the fill value is 254 corresponding to the pixel value needed to make DRAW_A visible. For the same reason, the fill value for DRAW_B should be 255. The calls to `mpgInsertPlaneGroup` are shown below.

```
MPG infoRec pseudo_color_info, true_color_info, overlay_info;

/* Overlay Window Plane group */
mpgInsertPlaneGroup(&overlay_info, OVERLAY, 0, MPG_VISIBLE_DRAWABLE,
       MPG_DRAW, 0);

/* 24 bit plane group */
mpgInsertPlaneGroup(&true_color_info, DRAW_A, 0, MPG_DRAWABLE,
       MPG_DRAW, 0);
mpgInsertPlaneGroup(&true_color_info, OVERLAY, 0,
       MPG_VISIBLE, MPG_FILL, 254);

/* 8 bit plane group */
mpgInsertPlaneGroup(&pseudo_color_info, DRAW_B, 0, MPG_DRAWABLE,
       MPG_DRAW, 0);
mpgInsertPlaneGroup(&pseudo_color_info, OVERLAY, 0,
       MPG_VISIBLE, MPG_FILL, 255);
```

A transparent pixel device is one of the more difficult devices to set up. The other device types should be easier.

Control Plane Group

The control plane group device requires no special MPG setup for overlay window processing. Use the standard MPG setup facilities and overlay window processing and rendering transparency work properly.

For example, a device with a 24-bit image plane group (DRAW_A), an 8-bit image plane group (DRAW_B), an 8-bit overlay plane group (OVERLAY), and a control plane group (WID), has the following segmentation:

```
MPG infoRec pseudo_color_info, true_color_info, overlay_info;

/* Overlay Window Plane group */
mpgInsertPlaneGroup(&overlay_info, OVERLAY, 0, MPG_DRAWABLE,
                        MPG_DRAW, 0);
mpgInsertPlaneGroup(&overlay_info, WID, 0, MPG_VISIBLE,
                        MPG_FILL_WID, 0);

/* 24-bit plane group */
mpgInsertPlaneGroup(&true_color_info, DRAW_A, 0, MPG_DRAWABLE,
                        MPG_DRAW, 0);
mpgInsertPlaneGroup(&true_color_info, WID, 0, MPG_VISIBLE,
                        MPG_FILL_WID, 0);

/* 8-bit plane group */
mpgInsertPlaneGroup(&pseudo_color_info, DRAW_B, 0, MPG_DRAWABLE,
                        MPG_DRAW, 0);
mpgInsertPlaneGroup(&pseudo_color_info, WID, 0, MPG_VISIBLE,
                        MPG_FILL_WID, 0);
```

Shared

If the shared device is a memory-mapped device with the `pScreen->devPrivate` pointing to a screen pixmap that can address the device, the OVL package is automatically initialized. This enables overlays to be available on that screen.

Custom

The custom device is the most difficult to use in the OVL package. If the device *almost* adheres to one of the above device types, it can initialize everything, and then wrap all of the necessary rendering / window manipulation components to complete its processing. For overlay window requests that are not a part of the core protocol, a wrapping mechanism is provided in this GPI. See “ovlWrapDevFuncs” on page 74 for a complete description of this wrapping process.

A device able to port using this method is one that has an extra plane group that requires special processing that MPG does not provide.

6

Initializing Overlays

The server implements all of the functionality for overlay window processing and rendering transparency. There are three basic steps required to use this feature on a device. First, the device must describe its plane groups appropriately to the MPG package. This was discussed in the previous section. The last two steps are described here. They are combined into a single initialization function, `ovlScreenInit`.

Once a device has described its plane groups to the MPG package, the OVL package can create and process overlay windows on any visual supported by the device. However, some of the visuals may be more *optimal* than others for overlay window processing. For example, a device may have a plane group that has special features for rendering transparency or is simply a dedicated overlay plane group to facilitate minimum damage to its underlay plane groups. The device needs a method to *hint* to the client that this visual is more optimal for overlay windows than other visuals.

In the Overlay Window API there are portable visual queries that allow the client to query which visual pairs are optimal for overlay window processing. If the device has specified that there are no optimal visual pairs, the portable visual queries return regular visuals that match the clients request. See the *Solaris X Window System Developer's Guide* for a complete description of the portable visual queries.

The second step for enabling overlay window processing is to describe all of the overlay and underlay combinations that are optimally supported by the device. An overlay/underlay combination is called a *pair*. The second step is combined with the third step, calling the overlay initialization function `ovlScreenInit`.

`ovlScreenInit` is called to initialize overlay window processing and describe the set of optimal overlay/underlay pairs supported by the device. This routine is given a list of pairs and the number of pairs. It must be called during screen initialization and it must be called after the MPG package has been initialized.

Each pair in the list has an overlay and underlay MPG info structure. All visuals pairs that may be derived from the MPG info pairs are then used to signify an optimal pair of overlay/underlay visual pairings. Because of the matching scheme used in the API, devices are encouraged to submit the pair list in most optimal to least optimal order.

Some device may not have any optimal overlay/underlay pairs. This is the case on shared pixel devices described above. If this is the case, `ovlScreenInit`() should still be called to initialize overlay window processing, but there should be no pairs passed into the function. This will indicate to the OVL package that no pairs are optimal.

Overlay GPI Specification

The following functions and data types define the Overlay GPI specification.

OvlPairs

```
typedef struct {
    mpgInfoPtr     pOvMpgInfo;    /* overlay mpgInfo */
    mpgInfoPtr     pUnMpgInfo;    /* underlay mpgInfo */
} OvlPair;
```

Description Specifies to the system a particular overlay/underlay pair that the device optimally supports.

ovlScreenInit

```
Bool
ovlScreenInit (ScreenPtr pScreen, unsigned int numPairs,
       OvlPair *pPairs)
```

Description This is the screen initialization function for Overlay Window support. The given set of pairs is exported to the client as the optimal pairs. If the device has no optimal pairs, pass in 0 for `numPairs` and `null` for `pPairs`.

Results Initializes overlay support on the given screen.

Returns `TRUE` on success
otherwise `FALSE`

Arguments `pScreen` is the screen structure for the device.

`numPairs` is the number of overlay/underlay pairs.

`pPairs` is a list describing the pairs.

`OvlPair` points to the MPG infos of the optimal overlay/underlay pair.

ovlWrapDevFuncs

```
void
ovlWrapDevFuncs (ScreenPtr pScreen, OvlDevFuncs *newfuncs,
                 long funcmask, OvlDevFuncs *oldfuncs)
```

Description

This function allows devices to wrap the requests associated with the overlay window extension. A full description of all the wrappable functions is given below.

This routine should only be needed by custom devices. The default functions handle all processing for devices that are supported by MPG.

Results

Wraps the overlay request dispatch functions.

Arguments

`pScreen` is the screen structure for the device.

`newfuncs` is a pointer to the new `OvlDevFuncs` to be instantiated.

`funcmask` is a mask of all the functions specified in `newfuncs`. `funcmask` indicates which functions in `newfuncs` are to be wrapped. If a given mask bit in `funcmask` is set, the appropriate field in `newfuncs` must be filled in with a valid function pointer. If a given mask bit in `funcmask` is not set, the appropriate field in `newfuncs` will not be accessed

`oldfuncs` (return) A pointer to the `OvlDevFuncs` previously instantiated.

The previously instantiated `OvlDevFuncs` is returned in `oldfuncs`, if provided. `OvlDevFuncs` is a structure containing pointers to the wrappable functions.

Valid values for `funcmask` are:

```
#define CopyPaintTypeMask        (1<<0)
#define CopyAreaAndPaintTypeMask (1<<1)
#define GetClutInfosMask         (1<<2)
#define ReadScreenInitMask       (1<<3)
#define ReadScreenMask           (1<<4)
#define ReadScreenUninitMask     (1<<5)
```

ovlGetPaintType

```
XSolarisOvlPaintType
ovlGetPaintType (GCPtr pGC)
```

Description `XSolarisOvlPaintOpaque` is returned unless a client has explicitly set the paint type to `XSolarisOvlPaintTransparent`.

Returns Current paint type of the given GC.

Arguments GC is the specified GC.

ovlIsOverlay

```
Bool
ovlIsOverlay (WindowPtr pWin)
```

Description Specifies whether the given window is an overlay window.

Returns `TRUE` if the window is an overlay window
`FALSE` otherwise.

Arguments `pWin` is the specified window.

XOvlClutInfo

```
typedef struct {
    VisualID   vid;
    int        pool;
    int        count;
} XOvlClutInfo;
```

Description A structure containing color lookup table information.

OvlDevFuncs

```
typedef struct {
    RegionPtr  (*CopyPaintType)();
    RegionPtr  (*CopyAreaAndPaintType)();
    int        (*GetClutInfos)();
    int        (*ReadScreenInit)();
    int        (*ReadScreen)();
    void       (*ReadScreenUninit)();
} OvlDevFuncs;
```

Description Defines the function vector of DDX handler functions for devices that want to wrap the overlay requests.

The following definitions are of data types in `OvlDevFuncs`.

CopyPaintType

```
RegionPtr
(*CopyPaintType) (OvlDevFuncs * devfuncs, DrawablePtr src,
       DrawablePtr dst, GCPtr pGC, int src_x, int src_y,
       unsigned int width, unsigned int height, int dest_x,
       int dest_y, unsigned long action, unsigned long plane)
```

Description If a device wraps the `CopyPaintType` request, their `CopyPaintType` function should take this form. This function uses the paint type information of the specified rectangle of `src` to control fill operations in the specified

rectangle of dst. src can be any type of drawable. If src is not an overlay window, plane specifies which bit-plane to use for paint type data. dst can be any type of drawable. The region of dst that corresponds to opaque pixels in src is filled with the current fill attributes of pGC. If dst is an overlay, then the region of dst that corresponds to transparent pixels in src is filled with transparent paint. If dst is not an overlay, then the region of dst that corresponds to transparent pixels in src is filled with the fill attributes of pGC, but with the fg and bg pixel values reversed. The function must restrict its fills according to the specified action which is one of XSolarisOvlCopyOpaque, XSolarisOvlCopyTransparent, or XSolarisOvlCopyAll referring to the filling of just the opaque pixels, just the transparent pixels, or both.

Results

Fills the appropriate regions of dst depending on the paint type data of src and the indicated action. Returns the region for which GraphicsExpose events must be generated.

Arguments

devfuncs is the current set of ovldevfuncs.

src is the source drawable.

dst is the destination drawable.

pGC is the GC to use for the fills. It has the same depth as dst.

src_x and src_y are the X and Y coordinates of the upper-left corner of the source rectangle relative to the origin of the source drawable.

width and height are the dimensions in pixels of both the source and destination rectangles.

dest_x and dest_y are the X and Y coordinates of the upper-left corner of the destination rectangle relative to the origin of the destination drawable.

action specifies which regions of dst should be filled.

`plane` specifies which plane of `src` should be used if it is not an overlay window. `1` means opaque, `0` means transparent.

CopyAreaAndPaintType

```
void
(*CopyAreaAndPaintType) (OvlDevFuncs * devfuncs,
        DrawablePtr colorsrc, DrawablePtr painttypesrc,
        DrawablePtr colordst, DrawablePtr painttypedst,
        GCPtr colorgc, GCPtr painttypegc, int colorsrc_x,
        int colorsrc_y, int painttypesrc_x, int painttypesrc_y,
        unsigned int width, unsigned int height, int colordst_x,
        int colordst_y, int painttypedst_x, int painttypedst_y,
        unsigned long action, unsigned long plane,
        RegionPtr *colorexposergn, RegionPtr *painttypeexposergn)
```

Description

If a device wraps the `CopyAreaAndPaintType` request, their `CopyAreaAndPaintType` function should take this form. This function copies the specified area from `colorsrc` to the specified area in `colordst` and copies the paint type area specified in `painttypesrc` to the specified paint type area of `painttypedst`. If `painttypesrc` is not an overlay window, `plane` specifies which bit-plane to use for paint type data. `colordst` may be any drawable of the same depth as `colorsrc`. `painttypedst` may be any type of drawable. If `colordst` is an overlay, then `painttypedst` will be the same overlay. If `painttypedst` is not an overlay, then `painttypegc` is used to fill the opaque and transparent regions of `painttypedst`. Opaque regions are filled according to the fill attributes of `painttypegc` while transparent regions are filled similarly but with the foreground and background pixel values reversed. This function must also handle the specified `action`. An `action` may be one of `XSolarisOvlCopyOpaque`, `XSolarisOvlCopyTransparent`, or `XSolarisOvlCopyAll` referring to the copying of just the opaque pixels, just the transparent pixels, or both. A pointer to a region indicating which areas must be exposed on the `colordst` drawable due to incomplete color or paint type

information is returned in the location pointed to by `colorexposergn`. A pointer to a region indicating which areas must be exposed on the `painttypedst` drawable due to incomplete paint type information is returned in the location pointed to by `painttypeexposergn`.

Results

Copies the given area and paint type data from one drawable to another. Returns the regions for which `GraphicsExpose` events must be generated.

Arguments

`devfuncs` is the current set of `ovldevfuncs`.

`colorsrc` is the color information source drawable. It can be any type of drawable.

`painttypesrc` is the paint type source drawable. It can be any type of drawable.

`colordst` is the color information destination drawable. It must be the same depth as `colorsrc`. It may be any type of drawable.

`painttypedst` is the paint type destination drawable. It can be any type of drawable. If `colordst` is an overlay, this parameter will be the same as `colordst`.

`colorgc` is the GC to use for copying the color information. It has the same depth as `colordst`.

`painttypegc` is the GC to use for rendering the opaque and transparent regions of the paint type information if `painttypedst` is not an overlay.If `colordst` and `painttypedst` are an overlay, this parameter will be the same as `colorgc`. It has the same depth as `painttypedst`.

`colorsrc_x` and `colorsrc_y` are the X and Y coordinates of the upper-left corner of the source rectangle relative to the origin of the color source drawable.

`painttypesrc_x` and `painttypesrc_y` are the X and Y coordinates of the upper-left corner of the source rectangle relative to the origin of the paint type source drawable.

`width` and `height` are the dimensions in pixels of all the source and destination rectangles.

`colordst_x` and `colordst_y` are the X and Y coordinates of the upper-left corner of the destination rectangle relative to the origin of the color destination drawable.

`painttypedst_x` and `painttypedst_y` are the X and Y coordinates of the upper-left corner of the destination rectangle relative to the origin of the paint type destination drawable. If `colordst` and `painttypedst` are an overlay, these values will be the same as `colordst_x` and `colordst_y`.

`action` specifies which portions of the paint type should be copied.

`plane` specifies which `painttypesrc` plane to use as paint type information if it is not an overlay window. `1` means opaque, `0` means transparent.

`colorexposergn` is a pointer to a location in which to store a pointer to the region that is to be exposed on the `colordst` drawable.

`painttypeexposergn` is a pointer to a location in which to store a pointer to the region that is to be exposed on the `painttypedst` drawable.

GetClutInfos

```
int
(*GetClutInfos)(OvlDevFuncs * devfuncs, ScreenPtr pScreen,
XOvlClutInfo ** pClutInfos)
```

Description

If a device does not use the Multiple Hardware Colormap (MHC) package to maintain its hardware colormaps, it needs to wrap this function. This information is used by the portable visual queries documented in the *Solaris X Window System Developer's Guide*.

This function should allocate a XOvlClutInfo structure for each visual that it exports. Each structure should contain the visual id, a unique pool identifier, and the number of hardware color look up tables that are available to the visual. The pool identifier will only be used to uniquely identify the group. This function should return the number of structures that are being returned. The calling function will free the data returned in pClutInfos.

Results Gets hardware color lookup table information.

Arguments devfuncs is the current set of ovldevfuncs.

pScreen points to the ScreenRec structure for which information is needed.

pClutInfos (return) is a pointer to be assigned the array of XOvlClutInfo structures returned.

XOvlClutInfo is a structure containing color lookup table information and is defined on page 76.

ReadScreenInit

```
int
(*ReadScreenInit)(OvlDevFuncs * devfuncs, WindowPtr pWin,
       int x, int y, unsigned int width, unsigned int height,
       Bool includeCursor)
```

Description If a device wants to wrap the ReadScreen request, it should wrap this function, as well as ReadScreen and ReadScreenUninit. If a device wraps the ReadScreen request, their ReadScreenInit function should take this form. This function is responsible for any initialization that the device needs to prepare for the ReadScreen request. The region of interest is specified by x, y, width, and height. x and y are relative to pWin. This function could, for example, take the cursor down if the cursor were a software cursor, intersected the region of interest, and includeCursor was set to xFalse.

Results Prepares for getting the color data displayed in a specified area.

Returns `Success` if no errors were encountered,
`!Success` otherwise

Arguments `devfuncs` is the current set of `ovldevfuncs`.

`pWin` points to the `WindowRec` structure used to compute the area of interest.

`x` and `y` specify the X and Y coordinates of the upper-left corner of the area to be read.

`width` and `height` are the dimensions of the area to be read.

`includeCursor` specifies whether or not to include the cursor image in the image.

ReadScreen

```
int
(*ReadScreen)(OvlDevFuncs * devfuncs, WindowPtr pWin, int x,
       int y, unsigned int width, unsigned int height,
       Bool includeCursor, pointer pBuffer)
```

Description If a device wants to wrap the `ReadScreen` request, it should wrap this function, as well as `ReadScreenInit` and `ReadScreenUninit`. If a device wraps the `ReadScreen` request, their `ReadScreen` function should take this form. This function is responsible for getting the color information of the area specified by `x`, `y`, `width`, and `height`. `x` and `y` are relative to `pWin`. `pBuffer` is a pointer to an area of memory big enough to store `width*height` number of long integers. It is important to note that this function copies into `pBuffer` the actual theoretical colors that can be displayed in the area and not the pixel values. Each long stored in `pBuffer` is of the form *XXBBGGRR*, where *XX* is unused, *BB* is a 16-bit intensity of blue, *GG* is a 16-bit intensity of green, and *RR* is a 16-bit intensity of red. `pBuffer` is allocated and freed by the calling function.

Called by — More than once for a single ReadScreen request. It will always be called within a `ReadScreenInit`/`ReadScreenUninit` block.

Results — Gets the color data displayed in a specified area.

Returns — Success if no errors were encountered, an X protocol error otherwise

Arguments — `devfuncs` is the current set of `ovldevfuncs`.

`pWin` points to the `WindowRec` structure used to compute the area of interest.

`x` and `y` specify the X and Y coordinates of the upper-left corner of the area to be read.

`width` and `height` are the dimensions of the area to be read.

`includeCursor` specifies whether or not to include the cursor image in the image.

`pBuffer` (return) points to an area of memory that is guaranteed to be large enough to hold the color data.

ReadScreenUninit

```
void
(*ReadScreenUninit)(OvlDevFuncs * devfuncs, WindowPtr pWin,
        Bool includeCursor)
```

Description — If a device wants to wrap the `ReadScreen` request, it should wrap this function, as well as `ReadScreenInit` and `ReadScreen`. If a device wraps the `ReadScreen` request, their `ReadScreenUninit` function should take this form. This function is responsible for doing any cleanup necessary after `ReadScreen` processing has completed. This could include putting the cursor back up, if it was previously taken down.

Results — Cleans up after getting the color data displayed in a specified area.

Arguments

`devfuncs` is the current set of `ovldevfuncs`.

`pWin` points to the `WindowRec` structure used to compute the area of interest.

`includeCursor` specifies whether or not to include the cursor image in the image.

Window ID Interface 7

This chapter describes the window identifier (WID) interface visible to Solaris Independent Hardware Vendors (IHVs) writing DDX ports. This interface consists of routines that are part of the MPG package. The MPG package is discussed in Chapter 5, “Multiple Plane Group Interface.”

Hardware Window IDs

Some graphics devices use WIDs to control the video output circuitry and drawing functions of their frame buffer. The term display ID (DID) is also used. For each pixel, a portion of the frame buffer describes how that pixel is to be output to the monitor. Examples of these attributes are: the specific buffer the color data is to be taken from, the other buffers it is to be combined with, and the output lookup tables to use. These video output aspects are called *WID pixel attributes* and are meaningful to the video display circuitry by a distinct bit pattern.

On *indirect WID* devices, the WID value in the frame buffer is used to look up the WID pixel attributes in a hardware table called a *WID lookup table*. On these devices, the WID value serves as an index into this table.

On *direct WID* devices the WID value in the frame buffer is the actual bit pattern of the WID pixel attributes. In this case, there is no indirection through a lookup table.

Usually, the pixels for a given window all share the same pixel attributes. For example, the pixels are all the same depth and all possess Z buffer information. Because of this, a distinct WID is allocated for use by the window and the WID plane group in the window's visible region is filled with the value of this WID.

Note – In this release, the WID interface refers to direct WID devices that are *not* supported in this release.

Software WID Object

The OpenWindows DDX interface provides a software object to represent hardware WIDs. On a direct WID device, each software WID represents a single hardware WID value. On indirect WID devices, a software WID can represent one or more contiguous hardware WID values.

The DDX interface provides functions a device handler can use to allocate WIDs. It also provides routines to initialize WID management. These routines are included in the MPG package.

On indirect WID devices there is a concept of a *WID free pool*. These are the WIDs in the hardware WID table that are not already being used by some window.

An opaque type, `WidPtr`, points to the software WID object. Opaque means that the format of the memory pointed to is known only by MPG. WID object attributes are only accessible with the routines defined in "Window ID Functions" on page 93.

The purpose of the software WID object is to be general enough that all device architectures can share WID properties, and to be extensible enough to accommodate device dependencies.

WID Object Attributes

A WID object has the following attributes. READ ONLY means that the attribute is set at WID allocation time by WID or a device-dependent WID routine. After allocation, the attribute cannot be changed by clients of WID.

- Screen READ ONLY

  The device that owns the WID.

- Visual READ ONLY

  The visual of the window passed to the allocation function.

- Value READ ONLY

  The bit pattern rendered into the WID plane group that uses the WID.

- Number READ ONLY

  The number of contiguous WIDs described by the WID object. For direct WID devices, this will always be `1`. For indirect WID devices, the value of the WID object is the index into the WID table of the first WID in the group. The values of the other WIDs in the group are in sequentially ascending order relative to the first WID. To be specific, if `n` is the value of the WID object, the values of subsequent WIDs in the group are n+1, n+2, ..., n+(number-1).

- Unique READ ONLY

  A Boolean that indicates whether the WID can be shared among multiple windows. A value of `TRUE` means that the WID is not sharable; a value of `FALSE` means that the WID can be shared.

  For example, the unique attribute of the WID of a hardware double-buffered window might be `TRUE`. Another example of a unique WID is for hardware clipping. This type of WID must be unique because if another window shares the WID, drawing to the first window could happen in the other window sharing the WID, which is not the desired behavior.

- Flavor READ ONLY

  A small integer representing the union of all pixel attributes for the device, not including unique fields and colormap control. Unique fields include display buffer control and hardware clipping. Non-unique fields include depth and Z buffer. The values of this attribute are device-dependent. For more information on flavors, see Chapter 8, “Colormap Interface.”

- DevData READ/WRITE

  An opaque handle to arbitrary device-specific data.

- ColorLut READ/WRITE

  The identifier of the hardware color lookup table to use for displaying windows using WID.

For devices supporting only a single hardware color lookup table, the value of this attribute is undefined and setting it is ignored.

Two WID objects are considered to be equal if their values are equal.

Accessing WID

All files using the WID routines of MPG must include the following header file:

```
#include “mpg/wid.h”
```

Dynamically link all shared objects using WID with `libmpg.so`.

Using MPG

Devices that use WIDs are multiple plane group (MPG) devices because there must be a plane group filled with the proper WID values when a window is moved. MPG does this filling with a process called *WID preparation*.

Device handlers that use WIDs must first initialize MPG by calling `mpgScreenInit`, `mpgInsertplanegroup`, and other MPG functions.

How to Use WID

This section describes the purpose for and usage of the WID function listed in “Window ID Functions” on page 93.

DDX Handler

DDX handlers use the WID function to:

- Initialize WID

  `widScreenInit` is used to initialize WID for the screen and should be called before any other WID functions.

- Create windows

The DDX handler wraps `pScreen->CreateWindow`. If the device has a single color lookup table, call `widAllocate` to create a new WID for that window and then call `widSetWindowWid` to attach it.

If the device has multiple color lookup tables, the DDX handler calls `cmapMhcWindowAttachWid`.

See Chapter 8, “Colormap Interface” for more information on devices with multiple color lookup tables.

MPG

MPG uses WID to:

- Change WIDs

  MPG uses `widDecref` to indicate there is one less window using old WID and `widIncref` to indicate there is one more window using the new WID.

- Prepare Window WIDs

  MPG uses `widGetValue` to get the value with which to fill the WID plane group.

CMAP

CMAP uses WID function to:

- Avoid unnecessary preparations

  If CMAP assigns a WID to a window that was the same as the old, it does not try to reprepare the WID. It uses `widGetValue` and the comparison operator `==` to make the necessary test.

- Notify WIDs of color lookup table changes

  When `XInstallColormap` changes the hardware color lookup table assignment of a colormap, the WIDs of all windows using that colormap are notified of the change so that the given color lookup table can be displayed in these windows. To do this notification, CMAP calls `widSetColorLut`. This can also occur in `XUninstallColormap` if it tries to implicitly reinstall a colormap that previously lost its color lookup table because of another installation.

- Manage flavors

  CMAP attempts to share WIDs between windows of the same flavor. It uses `widGetWindowWid`, `widGetFlavor`, `widGetValue`, and the comparison operator `==` to do the necessary tests. When CMAP attempts to share WIDs, it ignores unique WIDs by calling `widGetUnique`.

- Assign new WIDs

  When an `XSetWindowColormap` occurs, CMAP attempts to find an existing WID of the same flavor as the window. If it cannot, it creates a new one, using `widAllocate`, and assigns it to the window using `mpgSetWindowWid`.

See Chapter 8, “Colormap Interface” for more information on WID creation and manipulation by the CMAP package.

WID Data Types

The function that initializes WID is `widScreenInit`. The following WID data types describe the device-dependent WID functions that must be supplied to the `widScreenInit` function.

WidPtr

A pointer to a WID object. A WID object represents one or more device WIDs. This pointer is not passed as an argument to `widScreenInit` (see page 93), but it is central to the set of functions described in this chapter.

```
typedef void *WidPtr;
```

Note – This pointer is opaque. The internal format of `_WidObj` is not exposed to the DDX handler. Use the utility functions provided to access `WidPtr`.

WidAllocFunc

```
typedef WidPtr (*WidAllocFunc)(ScreenPtr pScreen,
        VisualID visual, int count, Bool unique, CARD32 flavor);
```

Purpose This is the WID allocation routine supplied by the device handler.

Results It allocates one or more contiguous WIDs from a WID table. The location and format of the WID table is device, and possibly visual, dependent.

Arguments `visual` is used by devices whose WID allocation depends on the window's visual. This type of device internally associates a visual with device-dependent WID data, such as the location of the WID table. When the allocate function is called, the device data associated with `pWin`'s visual is retrieved and used as appropriate.

`count` is the number of contiguous WIDs to allocate. For direct WID devices, a WID object is limited to a single hardware WID, so this value must always be `1`. The base WID value is aligned on a power-of-two boundary, which is determined by rounding up `count` to the next power of two. If n is the base WID value, subsequent WID values in the sequence are n+1, n+2, ..., n+(`count` - 1).

`unique` is `TRUE` if the WID is non-sharable. This argument is used by devices that allocate unique WIDs in different tables from the non-unique ones.

`flavor` is an additional argument to use for your own purposes. For example, if hardware clipping WIDs are allocated in a different WID table than software WIDs, `flavor` would be used to indicate the allocation of a hardware WID versus a software WID. See "Flavors" on page 122 for a detailed description of how to assign flavor values.

Returns On direct WID devices, this routine returns `NULL` if `count` <> `1`. For indirect WID devices, if `count` > 1, multiple contiguous hardware WIDs are allocated.

On indirect WID devices, this function marks the returned WID(s) as *allocated* and removes them from the free pool.

WidFreeFunc

```
typedef void (*WidFreeFunc)(WidPtr pWid);
```

Purpose The WID free routine supplied by the device handler.

Returns On indirect WID devices, `WidFreeFunc` returns the WID(s) represented by the given WID object to the free pool and frees the WID object memory.

On direct WID devices, this routine frees the WID object memory.

WidSetColorLutFunc

```
typedef void (*WidSetColorLutFunc) (WidPtr pWid, CARD32 clutId);
```

Purpose Specifies the color lookup table ID that a WID is to display. This function is supplied by the device handler.

Results On indirect WID devices, this routine updates the WID table for the WID to display the given color lookup table.

On direct WID devices, this routine changes the Value attribute.

If the WID object consists of more than one hardware WID, the color lookup table selection attributes of all hardware WIDs is set to the same value, the appropriate value for `clutId`. Currently, this is only applicable to indirect WID devices.

Note – No WID preparation is done. The client is expected to call an MPG function to *reprepare*. This only affects direct WID devices.

Window ID Functions

This section lists the WID functions used by other parts of MPG, CMAP, and DDX handlers. “” on page 103 provides a description of the expected use of these routines.

General Routines

These routines are used by several different software components of the server, including MPG, CMAP, and the device handler. The device handler can call some of these routines from screen function wrappers such as `CreateWindow`, or from the device-dependent WID functions supplied to `widScreenInit`.

widScreenInit

```
Bool
widScreenInit (ScreenPtr pScreen, WidAllocFunc allocFunc,
       WidFreeFunc freeFunc, WidSetColorLutFunc setClutFunc)
```

Purpose This function initializes WID management for a screen.

Called by A DDX handler at screen initialization.

Arguments The argument functions are device-dependent functions that understand the device details for managing WIDs. These functions must be non-NULL.

widScreenClose

```
void
widScreenClose (ScreenPtr pScreen)
```

Purpose This function frees resources allocated by `widScreenInit`.

Called by the device's `ScreenClose` procedure

widAllocate

```
WidPtr
widAllocate (ScreenPtr pScreen, VisualID visual, int count,
       Bool unique, CARD32 flavor)
```

Purpose This function allocates a WID object appropriate for the specified visual on `pScreen`. Initially, the reference count for the WID is 0.

Arguments `flavor` must be less than the `maxFlavors` of the WID's plane group, or `NULL` is returned. `maxFlavors` is the value passed to `cmapScreenInit` for the WID's plane group. See Chapter 8, "Colormap Interface" for more information.

If `count` is > 1 on direct WID devices, a WID object is limited to a single hardware WID, so this value must be 1. For indirect WID devices, if `count` > 1, multiple contiguous hardware WIDs are allocated. The base WID value is aligned on a power-of-two boundary, which is determined by rounding up `count` to the next power of two. The base WID value is retrieved by calling `widGetValue`. If this value is n, subsequent WID values in the sequence are n+1, n+2, ..., n+(count - 1).

Returns On direct WID devices, this routine returns `NULL` if `count` > 1.

widIncref

```
void
widIncref (WidPtr pWid)
```

Purpose This function increments the reference count of a WID object.

widDecref

```
void
widDecref (WidPtr pWid)
```

Purpose This function decrements the reference count of a WID object. If the reference count becomes less than or equal to 0, the device-dependent `widFree` function is called. This function frees the WID object memory (see below).

Returns For indirect WID devices, the WID value(s) represented by the WID object are returned to the free pool.

widGetScreen

```
ScreenPtr
widGetScreen (WidPtr pWid)
```

Returns A pointer to the WID object's screen.

widGetVisual

```
VisualID
widGetVisual (WidPtr pWid)
```

Returns Returns the ID of the visual of the window with which the WID was created.

widGetValue

```
unsigned long
widGetValue (WidPtr pWid)
```

Purpose For single WID objects, this is the WID bit pattern to be rendered into the frame buffer. For multiple WID objects, this is the bit pattern of the first WID in the sequence.

Returns The value of the WID object.

widSetValue

```
void
widSetValue (WidPtr pWid, unsigned long value)
```

Purpose For single WID objects, this is the WID bit pattern to be rendered into the frame buffer.

Returns The value of the WID object.

widWinGetValue

```
unsigned long
widWinGetValue (WindowPtr pWin)
```

Returns The value of the WID object for the specified window.

widGetNumber

```
unsigned int
widGetNumber (WidPtr pWid)
```

Returns The number of hardware WID values represented by the argument WID object.

widGetUnique

```
Bool
widGetUnique (WidPtr pWid)
```

Returns Whether a WID is unique.

widGetFlavor

```
CARD32
widGetFlavor (WidPtr pWid)
```

Returns The flavor of a WID.

widSetDevData

```
void
widSetDevData (WidPtr pWid, pointer pDevData)
```

Purpose This function sets device-dependent data on a WID object.

widGetDevData

```
pointer
widGetDevData (WidPtr pWid)
```

Purpose This function gets device-dependent data on a WID object.

widSetColorLut

```
void
widSetColorLut (WidPtr pWid, CARD32 clutId)
```

Purpose This function sets the color lookup table ID for a WID object.

Results If the WID object consists of more than one hardware WID, the color lookup table selection attributes of the hardware WIDs are set to the same value, that is, the appropriate value for `clutId`.

Note – On devices with a single color lookup table, this value is ignored.

widGetColorLut

```
CARD32
widGetColorLut (WidPtr pWid)
```

Purpose This function gets the color lookup table ID for a WID object.

Note – On devices with a single color lookup table, this value is undefined.

widSetWindowWid

```
void
widSetWindowWid (WindowPtr pWin, WidPtr pWid, Bool prepare)
```

Purpose This function specifies a window's WID.

Results The reference count of `pWid` increases and the reference count of the old WID decreases.

If `prepare` is `TRUE`, the WID plane group in the window's visible region is filled with the WID value. This is done even if the old WID is the same as `pWid`.

widGetWindowWid

```
WidPtr
widGetWindowWid (WindowPtr pWin)
```

Returns — The WID of a window. This is `NULL` if `mpgWindowSetWid` has not been called.

Handler-Specific Routines

Call these functions only from the device-dependent WID functions supplied to `widScreenInit`.

widAllocObj

```
WidPtr
widAllocObj ()
```

Purpose — This function allocates memory for a software WID object.

Called by — The device-dependent `allocFunc`.

Results — The reference count of this WID object is set to 0.

widSetValue

```
void
widSetValue (WidPtr pWid, unsigned long value)
```

Purpose — This function sets the value of a WID.

Called by — The device-dependent `allocFunc`.

widFreeObj

```
void
widFreeObj (WidPtr pWid)
```

Purpose Frees memory allocated by `widAllocObj`.

Called by The device-dependent `freeFunc`.

WID Device-Dependent Allocation and Free Functions Implementation

The `widScreenInit` function initializes WID for a device. Before calling this routine, make whatever device-dependent preparations are necessary to start using WIDs. For example, allocate a screen `devPrivate` slot for storing device-specific WID data on the screen.

On indirect WID devices, after `widScreenInit` is called, all of the device WIDs are considered to be unallocated and in the free pool. WID values returned in WID objects allocated by `allocFunc` are removed from this pool until freed.

For some plane groups of a device, there is only a single WID. In this case, the `allocFunc` can return a WID object with this WID as its value; ignore the WID reference count.

Allocation Function

`widScreenInit` takes an `allocFunc` argument. This is the device-dependent WID allocation function. This function calls `widAllocObj`, which returns a partially initialized WID object. `allocFunc` then fills in various device-dependent attributes of the WID. This is illustrated in the following example function.

```
/* Note: required for a bug workaround (described below) */
typedef struct {
    unsigned long  opaque1[6];
    CARD32         clutId;
    unsigned long  opaque2[2];
} *WidInsidePtr;

WidPtr
myAllocFunc (ScreenPtr pScreen, VisualId visual, int count,
               Bool unique, CARD32 flavor)
{
        WidPtr  pWid;

        if (!(pWid = widAllocObj ())
            return (NULL);

        <allocate a hardware WID value>

        widSetValue(pWid, <window ID value>);
        widSetDevData(pWid, <anything the handler wants>);

        /*
        ** Initialize the color LUT by reaching inside the
        ** opaque object. This is a temporary bug workaround.
        ** See note below.
        */
        <initialize color LUT of hardware WID>
        { WidInsidePtr *pWidInside;
         pWidInside = (WidInsidePtr) pWid;
         pWidInside->clutId = <initial color LUT>;
        }
}
```

`value` is of type `unsigned long`, `clutId` is of type `CARD32`, and `devPrivate` is of type `pointer`.

The client is required to initialize the value attribute. It is also required that `clutId` be initialized. Initialization of `devPrivate` is completely optional.

Note – There is a bug in this release: `myAllocFunc` cannot call `widSetColorLut` to initialize `pWid`'s color LUT because the screen of `pWid` has not yet been initialized. `pWid` must have been assigned a screen for `widSetColorLut` to work. The workaround is to access the `clutId` field of the `pWid` object directly. To do this, the device handler must "reach inside" the otherwise opaque object. This implementation is allowed only for this workaround and will be removed in a future release when a `widXXX` function is provided for color LUT setting that does not require the screen to be initialized.

Note – Even if the device-dependent WID `freeFunc` calls `cmapMhcReleaseOverload`, `myAllocFunc` should never call `cmapMhcForceOverload`. This call is invoked at a higher level in the system.

Note – In general, you should not attempt to share WIDs between windows within this routine. Instead, you should use the facilities described in Chapter 8, "Colormap Interface." The only exception to this rule is when there is only a single WID for a visual. In this case, `myAllocFunc` can allocate `pWid` only once and return copies of the pointer to it.

Free Function

The `widScreenInit` function takes a `freeFunc` argument. This is the device-dependent WID free function.

If the device has multiple color lookup tables, this function should call `cmapMhcReleaseOverload` to notify CMAP that it might be possible to remove some overloading conditions. It passes the return value of `widGetVisual` as the argument to this routine. See Chapter 8, "Colormap Interface" for more information.

Next, it performs any device-dependent actions needed to free the WID. Finally, `freeFunc` frees the WID object memory by calling `widFreeObj`.

If the device does not have multiple color lookup tables, this function performs the device-dependent free actions followed by a call to `widFreeObj`.

7

Colormap Interface 8

This chapter describes the colormap interfaces (CMAP) visible to Solaris Independent Hardware Vendors (IHVs) writing DDX ports. The topics discussed are:

- Introduction to CMAP
- CMAP Call Summary
- Compiling and Linking
- MPG and WID Initialization
- CMAP Initialization and Utilities
- Controlling Multiple Hardware Colormap (MHC) device's WIDs
- Changing a Window's WID
- Changing a Window's Colormap

Introduction to CMAP

The CMAP interface provides colormap management for devices with hardware color lookup tables. Call it from your DDX handler to initialize the colormap functions of your device's `pScreen`.

CMAP manages colormaps for devices with both a single hardware color lookup table and multiple hardware color lookup tables.

Note – If you do not use CMAP to manage your colormaps, part of the DGA interface will not work. For information, see "DGA and Colormaps" on page 229.

8

CMAP Call Summary

General Calls

The CMAP interface provides these functions for initializing colormap management for devices, retrieving the device colormap attributes, and releasing memory:

- `cmapScreenInit`
- `cmapCloseScreen`
- `cmapGetDevFuncs`
- `cmapGetMultiple`
- `cmapGetCmapPriv`
- `cmapGetWidType`

When calling `cmapScreenInit`, you must specify whether the device has a single-color lookup table or multiple-color lookup tables.

MHC Calls

When you call `cmapGetMultiple`, multiple color LUT management has been selected and CMAP provides the following additional routines. These routines only operate when multiple color LUT management has been selected; they return error status in the single-color LUT case.

- `cmapMhcForceOverload`
- `cmapMhcReleaseOverload`
- `cmapMhcWindowAttachWid`
- `cmapMhcWindowDetachWid`
- `cmapMhcChangeFlavor`
- `cmapMhcAllocWids`

Compiling and Linking

If you have a color device, use `cmapScreenInit` to initialize CMAP. The interface to these routines is provided by these header files:

- `colormapst.h`
- `cmap.h`

These routines are built into the server, so symbolic references to these routines are resolved when your DDX handler shared object is loaded into the server.

Additionally, MPG DDX handlers should use `mpgScreenInit` to initialize MPG. The interface to this routine, and associated routines, is provided by the following header file:

- `mpg.h`

These routines are provided by `libmpg.so`. Dynamically link the device handler with this shared object.

Finally, dynamically link DDX handlers that use the following routines with `libmhc.so`:

- `cmapMhcForceOverload`
- `cmapMhcReleaseOverload`
- `cmapMhcWindowAttachWid`
- `cmapMhcWindowDetachWid`
- `cmapMhcChangeFlavor`
- `cmapMhcAllocWids`

MPG and WID Initialization

The Multiple Hardware Colormap (MHC) devices supported by CMAP are MPG devices that mostly use window IDs (WIDs). The Solaris DDK provides the WID interface for managing these aspects of device control. See Chapter 7, “Window ID Interface.”

Prior to initializing CMAP for multiple color LUT management, initialize MPG by calling `mpgScreenInit` and `mpgGetScreenState`. For more information, see Chapter 5, “Multiple Plane Group Interface.”

If the device also has WIDs, call `widScreenInit`. For more information see Chapter 7, “Window ID Interface.”

8

CMAP Initialization and Utilities

Screen Initialization Routine

To initialize either single or multiple color lookup table management, call `cmapScreenInit`. For MHC devices, call this routine after the MPG and WID packages have been appropriately initialized.

cmapScreenInit

```
Bool
cmapScreenInit (ScreenPtr pScreen, CmapDevFuncs *pDevFuncs,
       Bool multiple, int numClutPools,
       CmapClutPoolDesc *pClutPoolDescs, CmapWidType widType)
```

Purpose

Initialize colormap management for the given screen. This routine changes the following members of the screen: `CreateColormap`, `DestroyColormap`, `InstallColormap`, `UninstallColormap`, `ListInstalledColormaps`, and `StoreColors`.

The device must supply device-dependent routines for accessing its hardware color LUT(s).

Arguments

`pDevFuncs` points to a structure with pointers to these functions. This pointer must be non-NULL.

If `multiple` is `FALSE`, single hardware color lookup table management is initialized.

If `multiple` is `TRUE`, multiple hardware color lookup table management is selected. If this mode is selected, information describing the configuration of the hardware color lookup tables must be passed in the arguments `numClutPools` and `pClutPoolDescs`.

If `multiple` is `TRUE`, the argument `widType` indicates whether the device uses WIDs and, if so, what type of WID device it is.

If `multiple` is `TRUE`, `mpgScreenInit` must have been already called. If not, this routine returns `FALSE`. Furthermore, if `multiple` is `TRUE` and `widType` is `CmapWidIndirect` or `CmapWidDirect`, `widScreenInit` must have already been called. Otherwise, this routine returns `FALSE`.

Results The contents of `pDevFuncs` and `pClutPoolDescs` are copied into an internal structure rather than copying the pointers.

The data types used by `cmapScreenInit` are described in the following section.

Device-Dependent Color LUT Access Routines

A pointer to the `CmapDevFuncs` structure is passed to `cmapScreenInit`.

CmapDevFuncs

```
typedef struct {
Bool (*writeClutFunc) (ColormapPtr pCmap, CARD32 clutId)
Bool (*storeColorsFunc)(ColormapPtr pCmap, CARD32 clutId,
                        int ndef, xColorItem *pdefs)
/* reserved for future expansion */
pointer reserved[4];
} CmapDevFuncs;
```

Purpose Specifies device-dependent routines for accessing the device's hardware color LUTs. Use `WriteClutFunc` in your device handler to write an entire colormap into one of the hardware color LUTs. This structure member must always be non-NULL.

Arguments `clutId` is the device-dependent hardware identifier of the hardware color LUT into which the color data is written. If a single hardware color LUT operation has been selected, the value of `clutId` is arbitrary.

`storeColorsFunc` is provided by the device handler to update a hardware color LUT with a set of `XColorItem` changes. `ndef` is the number changes specified in the list of changes in `pdefs`.

Returns `TRUE` on success; `FALSE` on failure.

Implementing writeClutFunc

When updating a color LUT, a DDX handler should avoid updating color LUT entries whose corresponding colormap entry is unallocated. This reduces colormap flashing. The following sections discuss the various ways to implement this behavior.

Loading Color Lookup Tables

Some devices are *mapped-access* devices—devices with color LUTs memory-mapped into the X server process. The DDX handler can access the contents of these LUTs quickly. Other devices are *request-access* devices—devices with color LUTs accessed through a request, such as a kernel driver `ioctl`.

For best results, request-access devices require a different color LUT update strategy than mapped-access devices because the time required per access is different.

For request-access devices, the possible strategies are:

- Get the entire color LUT contents, update it with allocated colormap cells, and put the entire color LUT back.
- Get the color data for the allocated colormap cells and the list of allocated cells. Determine contiguous ranges of allocated entries. Invoke several requests to put the color data for these ranges into the hardware.

For mapped-access devices, the best strategy is:

- Get the color data for the allocated colormap cells and the list of allocated cells. Use the allocation information to directly copy the data into the hardware.

Do not use the strategy of caching color LUT contents in the DDX handler because this does not work with *DGA colormap-grabbing clients*. Instead, use one of the above strategies.

cmapGetColorData8

```
int
cmapGetColorData8 (ColormapPtr pCmap, unsigned char *pRmap,
       unsigned char *pGmap, unsigned char *pBmap,
       Bool *pRallocs, Bool *pGallocs, Bool *pBallocs)
```

Purpose

Gets color data and allocation information from a colormap. Use it if the hardware color LUTs have 8 output bits per channel.

Arguments

For indexed colormaps, the data for entry `i` is placed in `pRmap[i]`, `pGmap[i]`, and `pBmap[i]`.

For direct colormaps, the data for red entry `i` is placed in `pRmap[i]`, for green entry `i` in `pGmap[i]`, and for blue entry `i` in `pBmap[i]`.

The `pRmap`, `pGmap`, and `pBmap` locations corresponding to unallocated entries in `pCmap` are unchanged.

If you are not interested in allocation information for `pRallocs`, `pGallocs`, and `pBallocs`, the arguments are `NULL`.

Returns

The value `1` is returned on success, `0` on failure.

In `pRmap`, `pGmap`, and `pBmap` the color data allocated in `pCmap`. It is assumed that the number of output bits per channel is eight. The `pRmap`, `pGmap`, and `pBmap` arrays must be long enough to hold all of the entries of `pCmap`.

Information on allocated entries, if requested. To request allocation information, supply non-NULL arguments to `pRallocs`, `pGallocs`, and `pBallocs`.

For indexed colormaps, if entry `i` is allocated in `pCmap`, `pRallocs[i]` is returned as `TRUE`, otherwise `FALSE`.

For direct colormaps, if red entry `i` is allocated in `pCmap`, `pRallocs[i]` is returned as `TRUE`, otherwise `FALSE`. Likewise, `pGallocs` and `pBallocs` are used to return the allocation status of the green and blue entries.

cmapGetColorData16

```
int
cmapGetColorData16 (ColormapPtr pCmap, unsigned short *pRmap,
           unsigned short *pGmap, unsigned short *pBmap,
           Bool *pRallocs, Bool *pGallocs, Bool *pBallocs)
```

Returns This routine returns the color data for allocated entries in `pCmap` in `pRmap`, `pGmap`, and `pBmap`.

Note – This function returns the full 16 bits of color data for each channel. It is up to the caller to convert this data to the output bits of the hardware color LUT.

Implementing storeColorsFunc

Code Example 8-1 shows how to implement this device-dependent function in your DDX handler.

Note – In Code Example 8-1, the color LUTs are indexed, the pixel size is 8 bits, and hardware color LUT channel outputs size is 8 bits each.

Code Example 8-1 Direct or Indirect Colormap Into Indirect Color LUT

```
Bool
exampleDDstoreColors (ColormapPtr pCmap, CARD32 clutId,
        int ndef, xColorItem *pdefs)
{
        unsigned char rmap[256], gmap[256], bmap[256];
        xColorItem     expanddefs[256];

            /* Since the color LUTs are indexed, if we have a direct
             * colormap, we must translate the pdefs.*/
        if ((pCmap->pVisual->class | DynamicClass) == DirectColor) {
            ndef = cfbExpandDirectColors(pCmap, ndef, pdefs,
                expanddefs);
            pdefs = expanddefs;
            }
```

Code Example 8-1 Direct or Indirect Colormap Into Indirect Color LUT *(Continued)*

```
            /* Optimization: A common case for optimization is for the
             * change to be to all channels of a single entry. This
             * frequently happens when XAllocColor is called on a dynamic
             * colormap. */
        if (ndef == 1 &&
        (pdefs->flags & (DoRed|DoGreen|DoBlue)==(DoRed|DoGreen|DoBlue))) {
            unsigned char red, green, blue;
            red = pdefs->red >> 8;
            green = pdefs->green >> 8;
            blue = pdefs->blue >> 8;

            << put red, green, blue into color LUT clutId at pdefs->pixel >>

            return (TRUE);

        }

        << get entire current contents of color LUT clutId into rmap, gmap, bmap>>

        /* apply changes */
        while (ndef--) {
            if (pdefs->flags & DoRed)
                rmap[pdefs->pixel] = pdefs->red >> 8;
            if (pdefs->flags & DoGreen)
                gmap[pdefs->pixel] = pdefs->green >> 8;
                if (pdefs->flags & DoBlue)
                        bmap[pdefs->pixel] = pdefs->blue >> 8;
                pdefs++;
            }

            <<put entire rmap, gmap, bmap into the color LUT for clutId>>

            return (TRUE);

}
```

Code Example 8-1 shows a special case when only a single entry is being changed *and* all three channels of that entry are being changed. This is a significant optimization because this situation happens very frequently when color applications are started. For devices that use a system call to get the color LUT contents out of the hardware, this optimization avoids an extra system call.

Simulating a Direct Color LUT With an Indirect Color LUT

In Code Example 8-1, something special must be done when the colormap is direct (either TrueColor or DirectColor) and the color LUT is indexed.

When an `XStoreColors` is performed on a single channel of a direct color LUT, it affects the displayed colors for all pixels containing the bit pattern of the channel entry changed. For example, if red entry `0x05` was updated, the colors change for pixels `0x05GGBB`, where `GG` and `BB` are any legal value for the green and blue portions of the pixel. In this example, a single change to the red entry changes the colors of multiple pixels.

When the color LUT is indexed rather than direct, several color LUT entries must be changed to get this same effect. This is done by calling `cfbExpandDirectColors`. It converts the `pdefs` change list describing the changed channel entries into another change list which, when applied, updates an indexed color LUT to achieve the desired effect.

The specification of `cfbExpandDirectColors` is:

```
int
cfbExpandDirectColors (ColormapPtr pCmap, int ndef,
       xColorItem *indefs, xColorItem *outdefs)
```

This DDX function can be used by devices with any arbitrary number of color LUT output bits. It is not limited to devices with eight bits of output per channel.

Simulating an Indirect Colormap With a Direct Color LUT

The preceding section dealt with the case where the device has indexed color LUTs and the device handler chooses to export indexed visuals. It is also possible to simulate indexed visuals if the device color LUTs are direct. This is the subject of the next section.

Code Example 8-2 is a routine that can load either indirect or direct colormaps into a direct color LUT. The only difference is in the treatment of the pixel value. For an indirect colormap, the same pixel value is used to index into all three color channels. For a direct colormap, the pixel value is divided into separate channel indexes.

Code Example 8-2 Direct or Indirect Colormap Into Direct Color LUT

```
Bool
exampleDDstoreColor (ColormapPtr pCmap, CARD32 clutId, int ndef,
            xColorItem *pdefs)
{
    unsigned char rmap[256], gmap[256], bmap[256];
    Pixel pix;
    VisualPtr pVis = pCmap->pVisual;
    int direct = (pVis->class|DynamicClass) == DirectColor;

    <<get entire current contents of color LUT clutId into rmap, gmap, bmap>>

    /* apply changes */
    while (ndef--) {
            pix = pdefs->pixel;
            if (direct) {
                /* Direct colormap */
                if (pdefs->flags & DoRed)
                        rmap[(pix&pVis->redmask)>>pvis->redoffset] = pdefs-
>red>>8;
                if (pdefs->flags & DoGreen)
                        gmap[pix&pVis->greenmask)>>pVis->greenoffset] = pdefs-
>green>>8;
                if (pdefs->flags & DoBlue)
                        bmap[pix&pVis->bluemask)>>pVis->blueoffset] = pdefs-
>blue>>8;
            } else {
                /* Indirect colormap */
                if (pdefs->flag & DoRed)
                        rmap[pix] = pdefs->red>>8;
                if (pdefs->flags & DoGreen)
                        gmap[ix] = pdefs->green>>8;
                if (pdefs->flags & DoBlue)
                        bmap[pix] = pdefs->blue>>8;
            }
            pdefs++;
    }
    <<put entire rmap, gmap, bmap into the color LUT for clutId>>

    return (TRUE);
}
```

Note – The single-entry optimization in “Simulating a Direct Color LUT With an Indirect Color LUT” on page 114 can also be used in this situation, although it is not shown in Code Example 8-2.

Color LUT Pool Description

For multiple color LUT devices, each MPG `mpgInfo` structure uses a specific color LUT pool, called a *clut pool.* A clut pool contains one or more color LUTs. Windows with a particular `mpgInfo` have their colormaps installed into the color LUTs in this pool. The color LUTs in a pool are assigned on a first-come-first-served basis. Throughout its existence `mpgInfo` always refers to the same color LUT.

The `mpgInfo` structure is in the MPG library. It defines the plane groups used by a window, what they are used for, and the window management operations that are performed on them. For more information, see Chapter 5, “Multiple Plane Group Interface”.

A color LUT is identified with a *clut ID* that is a small positive number. The value is only interpreted by the device handler and is opaque to CMAP.

In the call to `cmapScreenInit`, the device handler must supply a description of the device’s clut pools, the pool each color LUT resides in, and the pools used by the device’s default `mpgInfo`s. The default `mpgInfo`s are the ones that the device handler specifies in the `mpgVisInfo` structure passed to `mpgScreenInit`. The device handler provides this description by passing in an array of `CmapClutPoolDesc` structures, one for each clut pool. The number of clut pools is passed as an argument to `cmapScreenInit`.

There are limitations on how `mpgInfo`s use clut pools. These are described below.

CmapClutPoolDesc Structure

Code Example 8-3 shows the `CmapClutPoolDesc` structure that describes the color LUTs assigned to a particular pool and the MPG infos that use them.

Code Example 8-3 `CmapClutPoolDesc` Structure

```
typedef struct {

    /* number of cluts in pool */
    unsigned int   numCluts;

    /* array of clut IDs in pool */
    CARD32         pClutIds[CMAP_POOL_MAX_CLUTS];

    /* number of MPG infos using pool */
    unsigned int   numPgs;

    /* array of MPG info dids */
    CARD32         pPgs[CMAP_MAX_PGS];

    /*
    ** maximum number of flavors for MPG infos
    ** using this pool
    */
    unsigned int   maxFlavors;

} CmapClutPoolDesc;
```

For each clut pool, `numCluts` specifies the number of cluts in the pool. `pClutIds` is an array containing clut IDs for each clut in the pool. `numPgs` is the number of `mpgInfo`s using the pool. `pPgs` is an array containing drawing IDs (DIDs) for each `mpgInfo` using the pool. The DID is the *internal ID* (iid) of the *drawing plane group* of that `mpgInfo` (this is the last plane group inserted into the `mpgInfo` with `op` `MPG_DRAW`). `numPgs` is the number of `mpgInfo` DIDs in the `pPgs` array. An `mpgInfo` DID can appear in no more than one clut pool description.

Note – Currently, `numPgs` must always be equal to `1`. See “Multi-Depth Color LUT Pool Sharing” on page 121” for more details on this constraint.

The maximum number of *flavors* (`maxFlavors`) for the pool must also be specified. See “Flavors” on page 122 for more detailed information.

Note – The CMAP interface refers to an `mpgInfo` with the abbreviations “Pg” or “PG.” These do *not* refer to individual plane groups. These abbreviations refer to combinations of plane groups and correspond to mpg Info structures.

Note – Currently, `CMAP_POOL_MAX_CLUTS` is `32` and `CMAP_MAX_PGS` is `32`.

Relationship to MPG

This section describes the relationship between windows, visuals, `mpgInfo`s, and clut pools in greater detail. See also Chapter 5, “Multiple Plane Group Interface” for additional information.

When `mpgScreenInit` is called, the device handler supplies an `mpgVisInfo` table that specifies, for each visual ID in the table, the default `mpgInfo` that is to be assigned to windows created with that visual. When `cmapScreenInit` is called, CMAP uses this table to map visual IDs to clut pools. It uses this mapping to determine the clut into which a window’s colormap should be installed. This depends on the window’s visual.

Window contents are stored in device memory buffers called *plane groups*. Multiple plane groups can be associated with a window. The plane group in which the image color data is stored is called the *drawing plane group*. Besides the drawing plane group, the window might require other plane groups to control rendering and to properly display the window contents. For example, it might require a *window id* plane group to control visibility or a *Z buffer* plane group to control 3D rendering. All the plane groups associated with a window are described in its `mpgInfo`.

When an X window is created, the X client selects a visual for the window. This visual is a type descriptor describing how the window should be displayed. It contains information such as class and colormap entries. At the same time the client selects a visual for the window, a depth is also selected. Both the depth and visual remain constant for a window throughout its existence. The device handler must assign each visual a unique visual ID.

The `mpgVisInfo` table passed to `mpgScreenInit` contains, for each visual, the default `mpgInfo` for that visual. This means that when a window is created, this table is used to find the `mpgInfo` for the window’s visual. This `mpgInfo` is assigned to the window and controls display of the window

contents and render to the window. In the `mpgVisInfo` table, more than one visual ID can point to the same `mpgInfo`. For example, this can happen if the visuals differ only in the type of colormap they use for display—an 8-bit PseudoColor visual and an 8-bit StaticColor visual can share the same `mpgInfo`.

Note – Currently, the number of visuals that can refer to the same `mpgInfo` is limited to 6.

The `mpgVisInfo` table is shown in Figure 8-1.

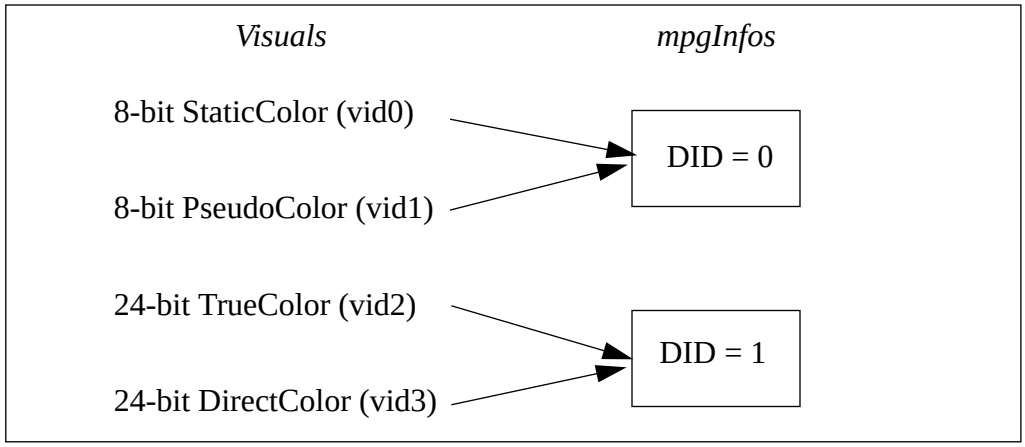


Figure 8-1 Relationship Between Visuals and `mpgInfo`s in the `mpgVisInfo` Table

After a window has been created, the X client may do something to it that requires a different `mpgInfo`. For example, the window might become multibuffered, grabbed through DGA, or a Z buffer attached. It might be necessary to move the window contents to a different drawing plane group. It

might also be necessary to add plane groups to the combination used by the window. MPG provides a routine, `mpgChangeInfo`, to allow a DDX handler to change the `mpgInfo` of a window. This is shown in Figure 8-2.

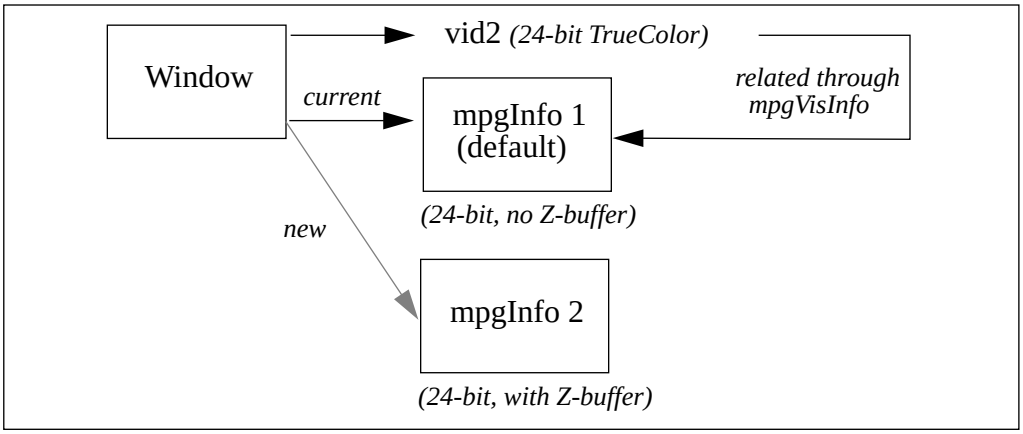


Figure 8-2 Changing the `mpgInfo` of a Window.

Because the visual and depth of a window never change, the new `mpgInfo` must have the same depth as the original `mpgInfo`. In addition, the new `mpgInfo` must always use the same clut pool as the original `mpgInfo`. For this reason, it is only necessary to specify to `cmapScreenInit` the clut pools used by the default `mpgInfo`s.

The first entry in the `pPgs` array of a clut pool description (`pPgs[0]`) defines the default `mpgInfo` that uses that clut pool. Other variants of this default `mpgInfo`, attached to windows using `mpgChangeInfo`, also use that same clut pool. This is shown in Figure 8-3.

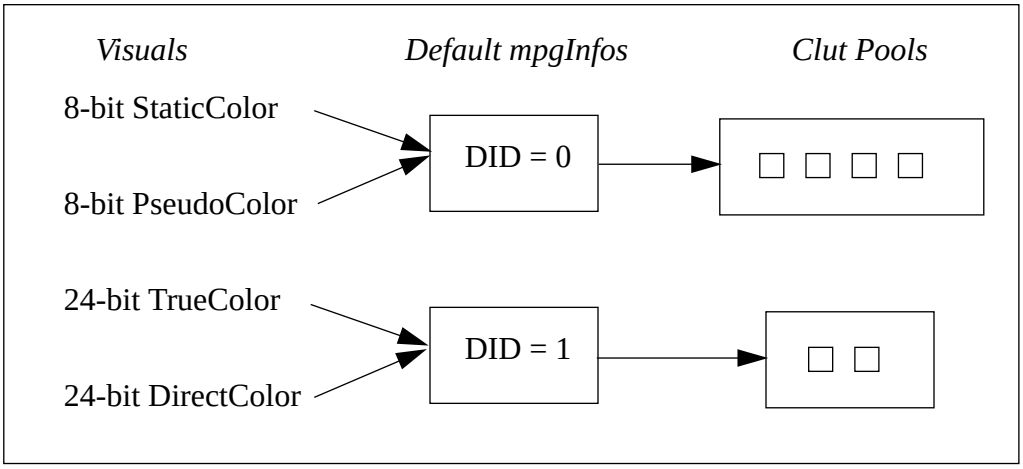


Figure 8-3 Relationship Between Visuals, Default `mpgInfo`s, and Color LUT Pools.

Multi-Depth Color LUT Pool Sharing

The `CmapClutPoolDesc` structure has an array of `mpgInfo` DIDs instead of just a single DID so that future configurations with multiple depths can share the same color LUT pool. These are called *multi-depth* configurations.

Note – Multi-depth configurations are not supported in the current release. Consequently, the `numPgs` of a clut pool description must always be 1. This restriction might be relaxed in a future release.

In a multi-depth configuration, a set of color LUTs is used by `mpgInfo`s of different depths. In such a configuration, the `pPgs` array contains more than one `mpgInfo` DID. It contains one for each default `mpgInfo` that used the clut pool. The different `mpgInfo`s in the array could be referred to by visuals of different depths. This is shown in Figure 8-4.

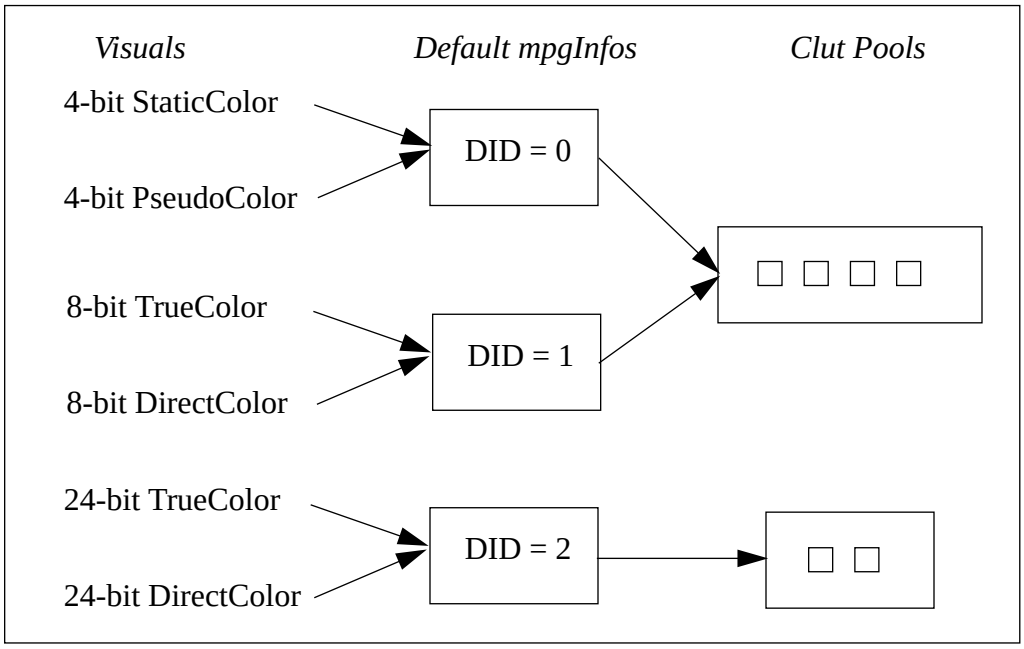


Figure 8-4 `mpgVisInfo` Table and Color LUT Pool Description for Multi-Depth (*not supported*)

Note – Sharing clut pools between default `mpgInfo`s of different depths is *not* supported in the current release. Also, sharing clut pools between default `mpgInfo`s of the same depth, but which differ in some other characteristic, is *not* supported either.

Flavors

CMAP needs to know the flavors of the `mpgInfo`s using its clut pools. At any one time, a window has an `mpgInfo`. On WID devices, a window's WID depends on this `mpgInfo`. The visible shape of the window is filled with this WID. The hardware uses the WID to control display of and rendering into the window. The type of the WID is called its *flavor*. CMAP uses the flavor of a WID to promote the sharing of WIDs between similar windows.

When CMAP is first initialized and the clut pools are described, the device handler needs to know the maximum number of flavors used by the set of all `mpgInfo`s using each clut pool. On non-WID devices, `maxFlavors` is always 0 for each clut pool description.

A flavor is a distinct combination of hardware WID attributes. It is identified by a small positive number. This number is opaque to CMAP and its value is not interpreted by CMAP. Because the number uniquely identifies a flavor, the term “flavor” is often applied to the number itself, although it really means the combination of WID attributes it represents.

For a particular hardware WID, the flavor of a WID depends on the hardware characteristics. The hardware WID is the bit pattern that the video display hardware uses to display a particular pixel on the screen. The bit pattern can also be used to control rendering to that pixel. Each pixel on the screen has an associated WID. On *Direct WID* devices, the controlling bit pattern is derived from the WID value itself. On *Indirect WID* devices, the WID value is used as an index into a table to find the controlling bit pattern. The controlling bit pattern of a WID is called its *attributes*. The attributes bit pattern is subdivided into a number of *fields*, each of which controls a particular characteristic, such as depth, double-buffer selection, or color LUT selection.

Note – Direct WID devices are *not* supported.

Since the purpose of flavors is to promote sharing of WIDs among similar windows, any WID attribute field that is specific to an individual window, and not sharable with other windows, is not a part of the flavor. For example, the double-buffer selection field of a WID is not part of the flavor because buffer changes to one window should not affect other windows. In a similar fashion, the color LUT selection field of a WID is not part of the flavor because even if two windows share the same visual (and the same `mpgInfo`) they might not share the same colormap. These types of WID attribute fields are referred to as *unique* fields. This means that each window that requires a WID in which a unique field changes, requires a unique WID. It cannot share the WID of another window.

Another example of a unique field is hardware clipping. It is unique because we don’t want hardware-clipped rendering into one window to spill out into another window. On some hardware, a WID field controlling the selection of a *fast clear set* might be a unique field. (A fast clear set is a hardware construct for rapidly setting the entire shape of a window to a specified pixel value).

Only sharable WID attribute fields are a part of the flavor. Examples include depth and Z-buffer-enable fields. These fields are called *flavor fields*.

The attribute fields of a WID vary from device to device. Follow this list of rules to determine the flavor fields for a device:

1. Start with the list of WID attribute fields that the hardware supports.
2. Eliminate the fields that are constant for all WIDs.
3. Eliminate those fields that, if enabled, prevent the WID from being shared by other windows. Examples: hardware clip, fast clear set.
4. Eliminate those fields that will be dynamically manipulated for an individual window. Examples: double buffer display select.
5. Eliminate those fields whose values are dependent on the values of other fields.
6. Eliminate the color LUT select field.

The remaining fields are the flavor fields. To derive the set of flavor IDs, assign unique small positive integers to all possible combinations of the flavor attributes.

The following is an example of four possible flavors that might be used by a device:

- Flavor 0: 8-bit, no Z buffer
- Flavor 1: 8-bit, Z buffer
- Flavor 2: 24-bit, no Z buffer
- Flavor 3: 24-bit, Z buffer

The `maxFlavors` of a clut pool is the sum of the flavors of the `mpgInfo`s that can use the pool. Continuing the **above** example, if clut pool 0 can be used by both an `mpgInfo` with an 8-bit Z buffer flavor and one with an 8-bit non-Z buffer flavor, the `maxFlavors` of this pool is 2.

When multiple windows using the same `mpgInfo` share the same colormap, only one WID is necessary to display the window contents. This is the WID for that `mpgInfo`. However, if the windows have different colormaps, then one WID per colormap is necessary. This is because CMAP installs each colormap into its own color LUT.

For example, there are three 24-bit Z buffered windows, each with its own colormap. These colormaps are installed into color LUTs 0, 1, and 2. These windows require three distinct WIDs, each differing only in the color LUT selection field. But the flavor attributes of these WIDs are all set to 24-bit and Z-buffered.

If a fourth window is created that shares the same colormap as the first window, it can share the first window's WID; it does not need a new WID. CMAP is designed to notice these opportunities for sharing.

For MHC WID devices, CMAP keeps track of the WIDs of windows using the colormaps it is managing. Whenever it needs to allocate a new WID for a window, it first checks to see if an appropriate sharable WID is already available. An appropriate WID is defined as a WID having the same color LUT as the window's colormap and flavor attributes the same as the desired flavor.

Initialization Example - Multiple Color LUT

Code Example 8-4 hows how to initialize colormap management for a device with two `mpgInfo`s. The first `mpgInfo` has one dedicated color LUT and the second one has four dedicated color LUTs.

Code Example 8-4 Initialize CMAP For a Device With Two Plane Groups

```
CmapClutPoolDesc   myclutDescArray[] = {

    /* Pool for 8-bit mpgInfo */
    {
        /* clut ids */
        1, { 0 },

        /* used by which mpgInfo */
        1, { 0 },

         /* max flavors */
        3
    },

    /* Pool for 24-bit mpgInfo */
    {
        /* clut ids */
        3, { 1, 2, 3 },
```

Code Example 8-4 Initialize CMAP For a Device With Two Plane Groups *(Continued)*

```
        /* used by which mpgInfo */
        1, { 1 },

        /* max flavors */
        1
    }
};

cmapScreenInit(pMyScreen, pMyDevFuncs, TRUE, 2,
        &myclutDescArray, cmapWidIndirect);
```

`pMyDevFuncs` is a pointer to a structure with the device-dependent colormap access functions.

Initialization Example - Single Color LUT

To initialize colormap management for a single color LUT, the following call should be used:

```
cmapScreenInit(pMyScreen, pMyDevFuncs, FALSE, 0, NULL, CmapWidNone);
```

`pMyDevFuncs` is a pointer to a structure with the device-dependent colormap access functions.

WID Types

When initialized for multiple color LUT management, CMAP needs to know whether the device uses WIDs. If the device uses WIDS, it needs to know whether the device is an indirect or direct WID device. Use the `widType` argument to `cmapScreenInit` to indicate this with one of the following values:

```
typedef enum {
    CmapWidUnknown,
    CmapWidNone,
    CmapWidIndirect,
    CmapWidDirect,
} CmapWidType;
```

Note – The value of the `widType` argument to `cmapScreenInit` is ignored in single-color LUT mode. `CmapWidUnknown` is for use by the system; do not use it in your DDX handler.

Note – Direct WID devices are not supported in this release.

Utility Routines

The following utility routines are provided for cleaning up after colormap management is no longer needed, accessing arguments to `cmapScreenInit`, and making the storage method of these data opaque to the calling function.

cmapCloseScreen

```
Bool
cmapCloseScreen (int index,ScreenPtr pScreen)
```

Purpose This function cleans up state initialized by `cmapScreenInit`. This function is responsible for restoring the color lut, the hardware wid, and other device dependent hardware states to correctly display the black and white colors of the glass tty console.

Called by The device-dependent `CloseScreen`.

cmapGetDevFuncs

```
CmapDevFuncs*
cmapGetDevFuncs (ScreenPtr pScreen)
```

Returns The device-dependent colormap access functions passed to `cmapScreenInit`.

cmapGetMultiple

```
Bool
cmapGetMultiple (ScreenPtr pScreen)
```

Returns `TRUE` if the given screen has been initialized with multiple color lookup table management.

cmapGetClutPoolDescs

```
void
cmapGetClutPoolDescs (ScreenPtr pScreen, int *pNumClutPools,
                      CmapClutPoolDesc **pClutPoolDescs)
```

Results The output arguments are untouched in the single-color LUT management case.

Returns In the multiple-color LUT management case, this procedure returns the number and array of pool descriptions given to `cmapScreenInit`.

cmapGetWidType

```
CmapWidType
cmapGetWidType (ScreenPtr pScreen)
```

Returns `widType` argument passed to `cmapScreenInit`, in multiple-color LUT mode.

`CmapWidUnknown`, in single-color LUT mode.

Colormap Private Data

CMAP uses the `devPriv` member of `ColormapRec` for its own purposes. If you want to attach device-dependent data to a colormap, it must coordinate with CMAP.

CMAP attaches its own private data structure to all colormaps. The colormap `devPriv` member points to this structure. CMAP reserves in its structure a data member called `devPriv`. Set `devPriv` to point to your own data.

To access `devPriv`, call `cmapGetCmapPriv`.

cmapGetCmapPriv

```
CmapPrivPtr
cmapGetCmapPriv (ColormapPtr pCmap)
```

Results If `devPriv` is `NULL`, a `CmapPrivRec` is created and `devPriv` is pointed to it.

Returns The `devPriv` member of a colormap. This function returns a pointer to a structure of the following format:

```
typedef {
    pointer     cmapOpaque1;
    pointer     cmapOpaque2;
    int         cmapOpaque3;
    pointer     devPriv;
} CmapPrivRec, *CmapPrivPtr;
```

You can read and write to `CmapPrivRec.devPriv` as needed by your DDX handler. The `cmapopaqueX` members are opaque; do not read or write to them. So, if `pCmapPriv` is the pointer returned by `cmapGetCmapPriv`, read or write to the `pCmapPriv->devPriv` data member to attach device-dependent data to the colormap.

Controlling MHC's WIDs

Most MHC devices are also WID devices. This section applies only to MHC devices that have WIDs.

An example of an MHC device that does not have WIDs, is a device with an 8-bit plane group and a 24-bit plane group whose visibility is selected by a 1-bit control plane. The value 0 in the control plane selects display of the 8-bit plane group and 1 selects the 24-bit plane group. Each plane group has a single, dedicated color LUT. This is an MHC device because it has two color LUTs; one each for the 8-bit and 24-bit plane groups. However, visibility is controlled by a control plane, not WIDs. If visibility was selected using a WID, then the device would be a WID device.

Devices that support more than one color LUT per plane group are usually WID devices. This sections applies to these devices also.

MHC devices with WIDs need to initialize the WID package. See Chapter 7, “Window ID Interface” for more information on WIDs.

CMAP uses a set of hardware WIDs to display colormaps in windows. CMAP is flexible about the number of WIDs it requires. It can be told to use more or less WIDs. If it uses less, color flashing might increase. The flashing condition persists until CMAP is told to use more WIDs, or until one of the colormaps causing the flashing is destroyed. See “Overloading Control Routines” on page 132 for information on how to tell CMAP the number of WIDs to use.

Overloading WIDs

CMAP uses WIDs to display different hardware color LUTs in different windows. Since, even on advanced display devices, WIDs are a relatively scarce resource, there might be times when you need a WID, but cannot get one.

The CMAP package is designed to be flexible about the number of WIDs it uses. In normal operation, it tries to use as many WIDs as it needs. However, if it tries to allocate a WID for a colormap and cannot, it shares the WID of another colormap that has a similar WID. This colormap is called an *overload partner*. When a colormap shares a WID with an overload partner, it uses the color LUT of the partner. Visually, the colormap flashes against the partner colormap. If all WIDs are used, this kind of flashing can occur even if there are free hardware color LUTs because there must be a free WID and a free hardware color LUT for a window to have its own LUT. This WID sharing technique is called *overloading*.

Depending on the type of device, CMAP might not be the only consumer of WIDs; the handler itself might need to use WIDs. For example, if it assigns special WIDs to hardware clipped windows or hardware double-buffered windows. In some situations, when the handler needs a WID it absolutely must acquire it; it cannot share the WID with some other window. In this case, the handler uses a unique WID.

You need to handle WID allocation failure if your handler uses WIDs. Rather than failing the operation requiring the WID, the handler is permitted to *steal* a WID from CMAP. It does this by forcing CMAP into an overloading situation.

In most cases, this approach is preferable: overloading CMAP means that there is more colormap flashing, but failing means that the application window needing the WID cannot be created at all.

It is recommended, therefore, that when you try to allocate a unique WID, and the allocation fails, call `cmapMhcForceOverload`. (The only exception to this is from the device-dependent `widAllocate` function.) This routine forces CMAP to give up a WID by overloading two colormaps onto each other. However, this routine does not always result in a free WID—there might not be any more free WIDs. When `cmapMhcForceOverload` fails (returns `0`), the handler has no other option but to return failure.

When forcing an overload condition, be sure to also call `cmapMhcReleaseOverload` whenever it frees a WID. This allows CMAP to remove any overloading conditions that exist and go back to less flashing. Always do this from the device-dependent WID free function, `freeFunc`. See Chapter 7, “Window ID Interface” for more information.

Note – The use of `cmapMhcReleaseOverload` and `cmapMhcForceOverload` from the WID free function is *not* symmetric. Even when the free function calls `cmapMhcReleaseOverload`, its counterpart allocation function should never call `cmapMhcForceOverload`. The `cmapMhcForceOverload` call is made elsewhere in the device-independent layers of the system.

Overloading Control Routines

cmapMhcForceOverload

```
int
cmapMhcForceOverload (ScreenPtr pScreen, VisualID visual)
```

Purpose	Forces CMAP to give up a WID.
Called by	A device handler that needs a unique WID for another purpose, such as double buffering.
Arguments	`visual` indicates the visual type of the WID.
Returns	`1` if it gives up a WID; `0` otherwise.

This code seeks to free a WID of *any* flavor for the visual. It starts at the least recently installed colormap in the visual's color LUT pool and progresses toward more recently installed ones. For each colormap, it attempts to find a viable overload partner colormap of the same flavor. To find the overload partner, it starts at the least recently installed colormap and progresses toward the most recently installed. It prefers partners that are not already overloaded, but accepts partners already overloaded. If it finds a partner that is already overloaded, the colormap becomes *over-overloaded*.

Note – This heuristic attempts to minimize the effect on windows with *hot* (most recently installed) colormaps by confining flashing effects on less recently used colormaps, even if it has to over-overload to do it.

Note – Call this routine only if the device handler needs a unique WID and cannot get one. Do not call this function when creating a sharable WID for a window. Instead, let `cmapMhcWindowAttachId` handle it.

cmapMhcReleaseOverload

```
void
cmapMhcReleaseOverload (ScreenPtr pScreen, VisualID visual)
```

Purpose This routine tries to take back any overloaded colormaps. This requires a WID, so this routine is called when the caller has reason to expect that a WID is available. This is the case when the caller has just freed a WID.

The installed list of that visual's color LUT pool is searched for a colormap that is overloaded. The search progresses from the most recently installed colormap toward less recently installed ones until one is found that is overloaded or the end of the list is reached. When it finds one, it allocates a new WID and assigns it to all windows using that colormap. The overload condition is then removed.

Arguments `visual` indicates the visual type of the WID that is needed.

8

Changing a Window's WID

When the DDX handler for a non-MHC device creates a window, or changes a window's WID, it uses the WID routines of the MPG package to make the change. For example, when a window is first created the `CreateWindow` routine of the device's screen is called. This routine calls `widAllocate` to allocate a WID and then `widSetWindowWid` to attach the WID to the window.

If the device is MHC, it must let CMAP change the WID. To promote WID sharing, the CMAP package needs to keep track of both WIDs and colormaps used by windows. Specifically, CMAP must be notified when the DDX handler does one of the following operations:

- Creates a window
- Destroys a window
- Changes a window's colormap
- Changes a window from software clipping to hardware clipping
- Changes a window from single buffered to hardware double buffered

In either the MHC or non-MHC case, the DDX handler has ultimate responsibility for deciding when WIDs get allocated and when WID attributes are changed. MHC DDX handlers must use CMAP for these operations.

cmapMhcWindowAttachWid

```
int
cmapMhcWindowAttachWid (WindowPtr pWin, Bool unique, CARD32 flavor)
```

A device that uses WIDs must wrap the `pScreen->CreateWindow` routine to create the window by assigning the window a WID.

When the wrapping routine is called, it first calls the wrapped `CreateWindow`. Next, it calls the following routine that ensures that the window is assigned an appropriate WID. This routine checks if there is another window with an appropriate WID, and uses that; if not, it allocates a new WID. It can force an overload to get this WID.

This routine chooses an appropriate WID for the given window. The choice of WID depends on:

- The window's colormap

- The specified flavor
- The specified uniqueness

Arguments — If `unique` is `FALSE`, CMAP tries to use an existing sharable WID of the given flavor. If it cannot find an existing one, a new WID is allocated.

Results — If the window already has a WID, it is freed.

Returns — `1` is returned on success and `0` on failure.

cmapMhcWindowDetachWid

Prior to destroying a window on an MHC device, CMAP must be notified. To do this, the device handler wraps `pScreen->DestroyWindow`. It calls the following routine and then destroys the window. When the window is destroyed the reference count of the attached WID decreases. If this was the only reference to this particular WID, the WID is freed.

```
int
cmapMhcWindowDetachWid (WindowPtr pWin)
```

Changing A Window's Colormap

The device handler should wrap `pScreen->ChangeWindowAttributes`. This way, the device handler detects if a `CWColormap` change is occurring. If it does not, then call the wrapped `ChangeWindowAttributes` normally.

If the colormap is being changed, then it calls `cmapMhcWindowDetachWid` on the window first, the wrapped `ChangeWindowAttributes` next. then the `cmapMhcWindowAttachWid` last.

Note – If the call to `cmapMhcWindowAttachWid` fails, the device handler returns an error.

If a CMAP routine returns failure status during the the device handler's wrapped `ChangeWindowAttributes` or during the call to `cmapMhcWindowAttachWid`, then this indicates the MHC could not allocate a WID. If this is the case, the device handler needs to back out of the change it

was trying to make. To do this, the device handler should attach the old colormap to the window using the wrapped `ChangeWindowAttributes`. Next, it should call `cmapMhcWindowAttachWid` using the *flavor* and *unique* values of the old WID (that is, the WID that used to be attached to the window).

Note – This call to `cmapMhcWindowAttachWid` should never fail since the old WID was returned to the free pool of WIDs and should still be there.

Finally, the device handler's wrapped `ChangeWindowAttributes` should return a `BadAlloc` failure status.

cmapMhcChangeFlavor

Whenever a window is modified in a way that changes its flavor, CMAP must be notified. A new WID needs to be assigned to the window, one with the new flavor. It is CMAP that makes this reassignment.

Call the following routine whenever the device handler is about to make a change that affects a WID's flavor. The routine is given the desired flavor and it attempts to either share a WID of the same flavor or else allocate a new one. In either case, it finds a WID and assigns it to the window.

```
int
cmapMhcChangeFlavor (WindowPtr pWin, CARD32 newFlavor)
```

This function tells CMAP that you want a WID of a different flavor attached to the window. CMAP selects a new WID for the window, using either an existing sharable WID or a new WID.

Note – Call this function only for windows with sharable WIDs.

This function returns `1` on success and `0` on failure. A failure return indicates that a WID of the desired flavor could not be acquired for the window. In this case, the previous WID of the window is left untouched.

Example

Code Example 8-5 shows you how to change the flavor of a window in pseudo-code. Attaching a Z buffer to a window is used as a hypothetical example. This code might be called from the DGA GPI routine `DgaZbufSetup` in response to a call to the `libdga XDgaZbufGrab` API routine. See Chapter 10, “Direct Graphics Access Drawable Client Interface” for more information.

Note – This is only a hypothetical example to illustrate the changing of a WID flavor attribute. MPG provides a superior service for attaching a Zbuffer to a window. For most devices, the MPG service is preferred because it sets up the Z buffer contents to be moved when the window moves. See Chapter 5, “Multiple Plane Group Interface” for more information. The actual possibilities for changeable flavor attributes are device-dependent.

Note – Depth is a WID flavor attribute, but dynamically changing the depth of a window is not permitted under the X model.

Code Example 8-5 Changing the Flavor of a Window Pseudo-Code

```
#define DDZBufFlavor<< device-dependent >>

DDAttachZBuffer (WindowPtr pWin)
{
    WidPtr         pWid;
    unsigned long  value;

    pWid = mpgWindowGetWid(pWin);
    value = widGetValue(pWid);

    if (widGetUnique(pWid)) {
        if (device has indirect WIDs) {
            widAttrs = get WID LUT entry ‘value’
            <change widAttrs to specify Z buffer attached>
            WID LUT entry ‘value’ = widAttrs
    }   } else {
            /* device has direct WIDs */
            <change ‘value’ to specify hardware clipping>
            widSetValue(pWid, value);
        }
    } else {
        if (!mhcChangeFlavor(pWin, DDZBufFlavor))
```

Code Example 8-5 Changing the Flavor of a Window Pseudo-Code

```
            return failure;
    }

    <Do other device-dependent operations to attach Z buffer>
    /* For Direct WId devices, whenever you change a WID
     * attribute, you must reprepare the WID plan group of the
     * window. To do this, set the window's WID to same WID and
     * specify repreparation. You do not need to do this for
     * Indirect WID devices.
     */
    mpgWindowSetWid(pWin, pWid, 1);
    return Success;

}
```

Allocating Unique WIDs

There are times when one or more non-sharable WIDs are needed for a window—double buffering and *XGL stenciling*. These techniques require unique WIDs. Use the following function to allocate unique WIDs; it forces an overload if the WID allocation fails.

```
int
cmapMhcAllocWids (WindowPtr pWin, int number)
```

This function allocates the specified number of WIDs for the window. The window's current WID is dereferenced and the WID object representing the new WIDs is attached. The WIDs allocated are contiguous to a power-of-two boundary determined by rounding up `number` to the next power of two. The WIDs are unique.

The value of `number` must be >= 1.

This function returns `1` on success and `0` on failure. If `0`, `pWin`'s original WID is left untouched.

Example

Code Example 8-6 shows you how to allocate multiple unique WIDs. This is an example of a DGA-based graphics library that wants to clip rendering to a sub-region of the window. In the first part of the example, two consecutive unique WIDs are allocated by the device handler and returned via the DGA mechanism.

Code Example 8-6 Allocating Multiple Unique WIDs in Pseudo-Code

```
DDGetClippingWids (WindowPtr pWin)
{
    WidPtr         pWid;
    unsigned long  value;

    if (!cmapMhcAllocWids(pWin, 2))
        return failure;

    pWid = widGetWindow(pWin);
    value = widGetValue(pWid);

    <place value and value+1 in the DGA shared information page>

    return Success;
}
```

Initialize this routine as the DGA GPI routine, `WidSetup`. This routine is invoked via a call to the `libdga` API routine, `XDgaGrabWids`. For more information on these routines, see Chapter 10, “Direct Graphics Access Drawable Client Interface.”

To complete the example, the graphics library calls `XDgaGrabWids`, getting back the two WID values. The library then does the following:

- Enables the hardware clipping attribute of the WID 1. (This can be done because WID 1 is unique.)
- Prepares the WID plane group throughout the entire drawable region of the window to WID 2.
- Prepares the WID plane group in the interior of the clipping sub-region to WID 1.
- Sets up the hardware to render, clipped to WID 1.
- Renders the graphics.

This will result in the graphics being clipped to the sub-region, as desired.

Note – Currently, this example is applicable only to indirect WID devices. Multiple hardware WIDs per WID object are not supported on direct WID devices. If the same feature is desired on a direct WID device, write the routine to allocate two separate WID objects rather than using `cmapMhcAllocWids`. In this case, if either of the WID allocations fails, call `cmapMhcForceOverload`, and retry the failing WID allocation. Once allocated, the hardware WID values can be derived from the WID objects by calling `widGetValue` on each one. Finally, store pointers to these WID objects in the handler's `devPrivates` area of the window so they can later be freed when the window is destroyed. This may change in future releases.

Multibuffering Extension to X Interface

9

This chapter describes the MBX (the Multibuffering Extension to X) interface for DDX handlers. This implementation of MBX permits hardware multibuffering on devices with special hardware (HW MBX). Internal changes were made to the X11R5 MBX sample implementation and a device porting interface was added.

Devices *not* capable of supporting hardware multibuffering can still use the MBX extension without any porting effort. Devices capable of hardware multibuffering need to register a device-dependent function vector with the server during Screen initialization.

See the *Solaris X Window System Developer's Guide*, which is part of the SDK, for the MBX extension specification.

Multibuffering

Clients can grab MBX image buffers and render directly to them.

Multibuffered Windows and Multibuffer Sets

A window is *multibuffered* if the MBX API routine `XmbufCreateBuffers` has been called on the window. `XmbufCreateBuffers` creates a specified number of *multibuffers* associated with the window. At any one time, the contents of one of these multibuffers is displayed within the window. Together, the

window and its associated multibuffers form a *multibuffer set*. The window of a multibuffer set is called the *main window*. The main window and its multibuffers are called *members* of the multibuffer set.

Multibuffer Flip Modes

Two methods exist for an undisplayed buffer to become the displayed buffer, or *buffer flipping*. The first is the copy flip method, or MBCOPY_FLIP, where the framebuffer creates *n* buffers in memory. When a client requests that buffer, `i` becomes the displayed buffer and the pixel contents of `i` are copied to the pixel store of the window drawable. This copying is transparent to MBX clients. This means that if the client renders to multibuffer `i` (the current display buffer) again, the rendering should be immediately visible. But, since copying is being used, the rendering instead goes into a nonviewable pixel store. Buffer aliasing solves this problem in copy flip mode: whenever a multibuffer is made the current display buffer, its XID is aliased to refer to the pixel store of the window drawable. If the client makes any multibuffer the current display buffer, subsequent rendering to that multibuffer will be immediately visible because it is drawn to the pixel store of the window drawable, which is always viewable.

The second method of buffer flipping is video flip, or MBVIDEO_FLIP, and can only be accomplished on an MPG device. This method requires that the framebuffer be capable of switching the video to be displayed out of any of the multibuffers created for a window. When the DDX handler is notified to display buffer `i` it is responsible for switching the hardware so that it displays from buffer `i`. Additionally, it must notify MPG that the window has migrated to a new plane group so that rendering to the window or displayed buffer will be immediately viewable.

HW MBX Functions

MbxScreenInit

```
#define _MULTIBUF_SERVER /* do not want Xlib structures */
#include “multibufst.h”
#include “multibufstruct.h”
.
.
.
int
mbxScreenInit(pScreen, pMbxdevfuncs, major, minor)
ScreenPtr  pScreen;
void       *pMbxdevfuncs;
int        major;
int        minor;
```

Purpose This function initializes HW MBX from your DDX handler’s Screen initialization function, if the Screen supports hardware multibuffering.

Arguments `major` is the major version number of the server DDK (`1`).

`minor` is the minor version number of the server DDK (`0`).

Note – The MBX initialization function in last release, `MultibufferDevFuncsInit`, is supported in this release. See “New Features and Changes” on page xxiii for information on backward compatibility.

MbxDevFuncs

This function vector, as well as many other MBX data structures and constants referenced throughout this chapter, is defined in the `multibufst.h` (in `/usr/openwin/include/X11/extensions`) and the `multibufstruct.h` (in `SUNWowddk/extensions/server/multibufstruct.h`) header files.

```
typedef struct _MbxDevFuncs {
int         (*TryMpg)(WindowPtr, int, int, int);
PixmapPtr   (*CreateMultibuffer2)(WindowPtr, int, int, int, int, int);
void        (*DestroyMultibuffer)(WindowPtr, PixmapPtr, int, int);
PixmapPtr   (*ResizeMultibuffer)(WindowPtr, int, int, int, int);
void        (*RepositionMultibuffer)(WindowPtr, PixmapPtr, int, int);
int         (*DisplayMultibuffer)(WindowPtr,int);
int         (*SetMultibufferInvisible)(WindowPtr, PixmapPtr);
int         (*SetMultibufferVisible)(WindowPtr, PixmapPtr);
void        (*LastUpdateTime)(WindowPtr, u_long, u_long);
} MbxDevFuncs;
```

This function vector does not have to be completely filled in by every device. Only functions applicable to the device need to be filled in; other entries can be NULL. Functions, if supplied, are called by the device-independent layer of MBX when it needs to perform a device-dependent operation.

TryMpg

```
int
(*TryMpg)(WindowPtr pWin, int num_buf, int updateAction,
        int updateHint)
```

Purpose

This function is called when a client initiates multibuffering on a window, `pWin`, through MBX. The server attempts to create the requested number of buffers, `num_buf`, to associate with the window. This call requests the handler to indicate whether the device provides MPG multibuffering. If a device supports MPG, then the plane group used by the window for multibuffering might be different than the current plane group. In this case, this routine re-prepares the necessary plane groups. Refer to Chapter 5, “Multiple Plane

Group Interface" for more details. Non-MPG devices need not supply this routine and should have a NULL entry in the `mbufdevfunc` vector.

Results

If the number of buffers requested for multibuffering, `num_buf`, is greater than the available number that hardware can support, or if any other device- dependent criteria for turning on hardware multibuffering fails, hardware multibuffering is *not* enabled.

If all conditions are satisfied, the device maintains private information about the multibuffering state if necessary.

Returns

This function returns `1` if hardware multibuffering is enabled and `0` otherwise.

Arguments

The last two arguments to this function, `updateAction` and `updateHint`, are supplied because some devices might not handle all cases and combinations of update action or update hint.

`updateHint` indicates how often the client will request a different buffer to be displayed. This hint allows smart server implementations to choose the most efficient means to support a multibuffered window based on the current need of the application (dumb implementations may choose to ignore this hint). Possible hints are:

`MultibufferUpdateHintFrequent` means an animation or movie loop is being attempted and the fastest, most efficient means for multibuffering should be employed.

`MultibufferUpdateHintIntermittent` means the displayed image will be changed every so often. This is common for images displayed at a rate slower than a second. For example, a clock that is updated only once a minute.

`MultibufferUpdateHintStatic` means the displayed image buffer will not be changed any time soon. Typically set by an application whenever there is a pause in the animation.

`updateAction` indicates what should happen to a previously displayed buffer when a different buffer becomes displayed. Possible actions are:

`MultibufferUpdateActionUndefined` means the contents of the buffer last displayed will become undefined after the update. This is the most efficient action since it allows the implementation to trash the contents of the buffer if it needs to.

`MultibufferUpdateActionBackground` means the contents of the buffer last displayed will be set to the background of the window after the update. The background action allows devices to use a fast clear capability during an update.

`MultibufferUpdateActionUntouched` means the contents of the buffer last displayed will be untouched after the update. Used primarily when cycling through images that have already been drawn.

`MultibufferUpdateActionCopied` means the contents of the buffer last displayed will become the same as those that are being displayed after the update. This is useful when incrementally adding to an image.

CreateMultibuffer2

```
PixmapPtr
(*CreateMultibuffer2)(WindowPtr pWin, int num_buf,
       int cur_buf, int updateAction, int updateHint,
       int mode);
```

Purpose

When the device does not support hardware buffers, the buffers are implemented as software pixmaps. However, on devices supporting hardware buffers, this function has to create a pixmap that points to device memory and return a pointer to the pixmap.

Returns If the requested number of buffers is greater than what the hardware can support, this routine returns `NULL`. If the buffer was created, this routine returns a pointer to the pixmap.

Arguments `num_buf` indicates the total number of buffers being requested and this routine is called to create each of the requested image buffers.

`cur_buf` indicates the number of the buffer being created. `num_buf` is provided in case you want to perform a sanity check on `cur_buf`.

The values of `updateHint` and `updateAction` are supplied, because you might not want to support hardware buffers for certain values or combinations of `updateAction` and `updateHint`. See page 145 and page 146 for complete definitions of these arguments.

`mode` is an integer pointer that the DDX handler should fill in with either MBCOPY_FLIP or MBVIDEO_FLIP, depending on the type of buffer created. All buffers for any window must be the same mode.

Note – The MBX creation function in last release, `CreateMultibuffer`, is supported in this release. See "New Features and Changes" on page xxiii for information on backward compatibility.

DestroyMultibuffer

```
void
(*DestroyMultibuffer)(Windowptr pWin, PixmapPtr pPix,
       int num_buf, int cur_buf);
```

Purpose This function is called when the server is destroying the multibuffers of a window so that the DDX handler can clean up the resources used by the buffers. This function is called once for each buffer in a window's multibuffer set that is being destroyed.

Arguments

`pPix` is the `PixmapPtr` that was returned by `CreateMultibuffer2`.

`num_buf` is the total number of buffers allocated to this window. This number should not change throughout the existence of a multibuffer. It can change if a multibuffered window is unbuffered, then buffered again with a differing number of buffers. In particular, this value should not change during the buffer destruction process (rather than being updated after the deletion of each buffer/pixmap).

`cur_buf` is the buffer number of the buffer currently being deleted.

ResizeMultibuffer

```
PixmapPtr
(*ResizeMultibuffer)(WindowPtr pWin, int num_buf,
           int cur_buf, int updateAction, int updateHint);
```

Purpose

If a multibuffer window `pWin`, is resized, and if the buffers are in hardware, they need to be resized as well. Often, this means destroying the previously allocated hardware buffer and recreating a new one with the new size. This function, if available, is called to resize each of the buffers associated with the window `pWin`.

Results

The new dimensions of the buffer are the same as the dimensions of the window, `pWin`. All the conditions specified for `CreateMultibuffer` also apply to this function.

If the device maintains private data about the hardware buffers, it is updated as well. The contents from the buffer before it was resized are copied into the newly resized buffer.

Returns

If the hardware pixmap-buffer associated with the `cur_buf` is successfully resized, a pointer to this pixmap-buffer is returned. Otherwise a NULL pointer is returned.

Arguments `num_buf` indicates the total number of buffers associated with this window and `cur_buf` indicates the number of the buffer currently being resized.

See page 145 and page 146 for complete definitions of `updateAction` and `updateHint`.

RepositionMultibuffer

```
void
(*RepositionMultibuffer)(WindowPtr pWin,
       PixmapPtr pBuffer, int new_x, int new_y)
```

Purpose If a multibuffer window `pWin`, is repositioned, and if the buffers are in hardware, they each need to be repositioned. The hardware might need to be updated with the new origin of the buffer, as well as any private information that the device maintains about this buffer. This function, if available, is called to reposition each of the hardware buffers.

Arguments `new_x` and `new_y` indicate the new coordinates of the window.

Depending on the hardware, the contents in the hardware buffers might need to be copied to the new location.

DisplayMultibuffer

```
int
(*DisplayMultibuffer)(WindowPtr pWin, int buf_num)
```

Purpose When the client program issues a request to display a certain buffer on a multibuffered window, `pWin`, this function, if available, is called. If the multibuffer set of pWin is of type MBVIDEO_FLIP, mpgChangeInfo() must be called to migrate the window to the plane group of the new display buffer.

Returns This function then initiates flipping the buffer to display the hardware buffer associated with this buffer number and return `1` upon success and `0` on failure. If the hardware buffer flip fails for some reason, the contents of the buffer are copied to the window using `CopyArea`.

Arguments `buf_num` indicates the number of the buffer to be displayed.

`DisplayMultibuffer` can be an asynchronous function. It can post the buffer flip to the device and return immediately. If this is the case, it is the responsibility of the device handlers' rendering code to block until the buffer flip has been completed before proceeding to render.

Some devices do not display a new buffer by doing a flip in hardware. Instead their hardware is specialized to perform accelerated copying from the hardware buffer to the window. These devices have a NULL entry in the device function vector for this function.

SetupMultibufferInvisible

```
int
(*SetMultibufferInvisible)(WindowPtr pWin,
       PixmapPtr pPrevBuf);
```

Purpose This function, if available, is called to indicate that `pPrevBuf` is no longer the visible buffer of the multibuffered window, `pWin`.

In the device-independent part of MBX, the resource id of the window is aliased to the resource id of the visible buffer. When the client requests a new buffer to be displayed, the resource id of the window needs to be aliased to the new buffer. If the currently visible buffer is in hardware, the hardware might need to be updated to know that this buffer is no longer the visible buffer.

Returns This function returns `1` upon success and `0` on failure.

SetMultibufferVisible

```
int
(*SetMultibufferVisible)(WindowPtr pWin, PixmapPtr pCurBuf);
```

Purpose This function, if available, is called to indicate that the buffer, `pCurBuf` is the currently visible buffer on the multibuffered window, `pWin`.

After marking the currently visible buffer as invisible, MBX then aliases the resource id of the window to the resource id of the buffer (pixmap) about to be displayed. If this buffer is in hardware, the hardware might need to be updated to indicate that this buffer is now visible.

Returns This function returns `1` upon success and `0` on failure.

LastUpdateTime

```
int
(*LastUpdateTime)(WindowPtr pWin, u_long months,
       u_long milliseconds)
```

Purpose This function, if available, is called to find out when the last display update was completed. You are required to supply the last update time in your device handler.

Results This function assumes that `DisplayMultibuffer` returns after the buffer is flipped.

On devices that allow `DisplayMultibuffer` to be asynchronous, this assumption is no longer valid.

9

Direct Graphics Access Drawable Client Interface 10

The direct graphics access (DGA) drawable interface, like the rest of the DGA client interface, is not an application developer interface. To use it, a developer must know the specifics of the hardware interface for each device supported. Many graphics devices are supported under Solaris, and often the hardware interfaces are not documented in books available in your local bookstore. DGA is an interface targeted for IHVs (Independent Hardware Vendors) porting Solaris graphics libraries to a particular graphics device. Developers porting the XGL, XIL, and Direct Xlib libraries may want to take advantage of the DGA drawable interface in the device handlers for those libraries.

The DGA drawable interface is compatible to the DGA window grabber interface in Version 3.3. All of the existing DGA client interface routines are still supported. See “New Features and Changes” on page xxiii for information about backward compatibility.

Overview

The DGA drawable interface is the basic mechanism for sharing screen access between the window server and one or more X11 client processes. This allows a DGA client to access the frame buffer for improved performance while the window server is still in charge of managing screen real estate for all clients in order to maintain the integrity of the screen. This is accomplished via efficient locking primitives and shared memory information which is accessed via a set of routines and macros. Not only does it apply to windows residing on a screen, but to other types of drawables that can be created on a screen, such as pixmaps and MBX buffers.

10

The goal of DGA is to provide clients with *direct access* to the graphics hardware while retaining coherence with the window system. DGA allows the window server to pass device-specific information to Solaris VISUAL foundation library clients such as XGL, XIL, and Direct Xlib. The device-specific information is passed to the foundation library device handler so that the handler knows how to drive the hardware.

The coordination between the server and the client is provided by means of the DGA drawable interface. This interface performs two primary functions; first, it allows the server to pass the target drawable's size and clip shape to the client; and second, it allows the client to lock the drawable, so that it does not change while graphics are being rendered. It also enables the client to detect changes to the drawable, such as the addition of backing store, which the client must maintain. A secondary function of the drawable interface is a mechanism that allows the device-dependent portions of the server to share device-dependent information with the client.

Drawable Types

The OpenWindows server provides clients with several different types of resources on which graphics can be drawn. These resources are called *drawables*. Drawables are always associated with a particular X screen. There are two basic types of drawables: *viewable* and *nonviewable*. The pixel contents of viewable drawables can be directly seen by the user. They reside in special device memory from which a video signal can be output to the display screen. The contents of nonviewable drawables cannot be directly seen by the user. For the user to be able to view the drawable contents, the pixels of a nonviewable drawable must be copied to a viewable drawable.

Windows are always viewable drawables. Pixmaps are always nonviewable. MBX multibuffers may be either viewable or nonviewable depending on the type of memory where their pixels reside.

DGA Drawables

A graphics client that intends to do direct rendering into one or more drawables first makes arrangements with the window system to *grab* the drawable. This enables direct access to the drawable. Only window and pixmap drawables are grabbed. MBX multibuffer drawables are implicitly grabbed by grabbing their associated window.

Once a drawable is grabbed, the client must lock the drawable prior to rendering to it. The client must provide arguments to the lock routines specifying the drawable it is going to render to. For each drawable locked, the lock routines take a `Dga_drawable` and a buffer index. A `Dga_drawable` is a opaque handle returned by grabbing a drawable. It is some times also called a DGA *client structure* for the drawable. Depending on the values of the `Dga_drawable` and buffer index, the client can specify any of the following to be locked: a multibuffer of a window, the window itself, or a pixmap. For details see “Drawable Locking and Change Detection” on page 162.

It should be emphasized that multibuffers share the same Dga_drawable as their main window; it is only through the buffer index that windows and multibuffers are differentiated. The buffer index for windows and pixmaps is always -1. The buffer index for a multibuffer is always a small natural number.

The drawable’s client structure contains a pointer to the shared memory information about the drawable. This information is shared with the window server. It acts as a communication pathway between the window server and the client. DGA clients cannot access the contents of the `Dga_drawable` structure or the shared memory information directly; access it through this DGA interface. When this initialization transaction is complete, the client can begin rendering into the drawable.

The window server updates its information in response to changes in the drawable’s attributes. These changes are usually initiated by the user, by popping up a menu or resizing a window, for example. Some of these changes can be initiated by a client program through a programmatic interface, such as the MBX (multibuffering extension) API or the XGL double buffering API. The client uses the routines provided in the drawable interface to maintain consistency with these changes.

Mutual Exclusion

At a given time, only a single process may access the shared drawable information. Mutual exclusion is enforced by lock and release primitives in the client and window server code streams. Denial of access permission is transparent to the requesting process; it will be blocked when it tries to lock down the shared data structure and will not continue until it has acquired the right to own the shared data structure. Once a process acquires the shared data structure, it retains uninterrupted use of it. When a process decides to give up ownership, another process may acquire ownership. For this reason, the DGA

locking primitives should not be held outside of rendering code or for extended periods of time. At present, DGA does not support multi-threaded graphics access to a single drawable from within a single client process.

The drawable interface enforces fairness in that, a process which is denied access is given ownership rights as soon as they become available. Release of ownership is voluntary and the owning process can retain ownership for an indefinite period of time. This exposes a potentially vulnerable area in the mutual exclusion technique, since the owning process may loop, sleep, terminate or perform time-consuming operation while in possession of access rights. This situation is ameliorated by a time-out mechanism that limits a client process's ownership time to a maximum value (currently three seconds). The window server process is not so limited and may retain possession of the lock indefinitely.

Sites

A drawable can reside in different types of memory called *drawable site types*. System memory and device off-screen memory are examples of drawable site types. Within a site type, a drawable has a location. This location is defined either by an address or, for some types of multibuffers, a render buffer state. Together, the site type and location within the site define the drawable's *site*.

In between locks, a drawable's site may change for several reasons:

1. The display buffer may have changed, causing aliasing to another site. (See "Multibuffer Flip Modes" on page 142 for more information).

2. The cache state of the drawable may have changed.

Because any type of drawable can potentially change site between locks, the client should either:

- always check for a site change when the drawable is locked and `DGA_DRAW_MODIF` returns nonzero, or
- register a site change notification function

There are two ways of detecting site changes:

1. MODIF Testing

   A site change causes `DGA_DRAW_MODIF` to return nonzero. As part of the state interrogation that follows this, the client can call `dga_draw_sitechg` to see if the site has changed since the last lock.

2. Notification

   Another way to detect site changes is to register a site change notification function. This function is automatically called by the drawable locking routines when a site change is detected.

The client may use either of these two approaches.

When a drawable is first grabbed, its site is considered changed so the client can synchronize with the initial site.

Backing Store

When a window has backing store, DGA clients must update the backing store as illustrated in Figure 10-1.

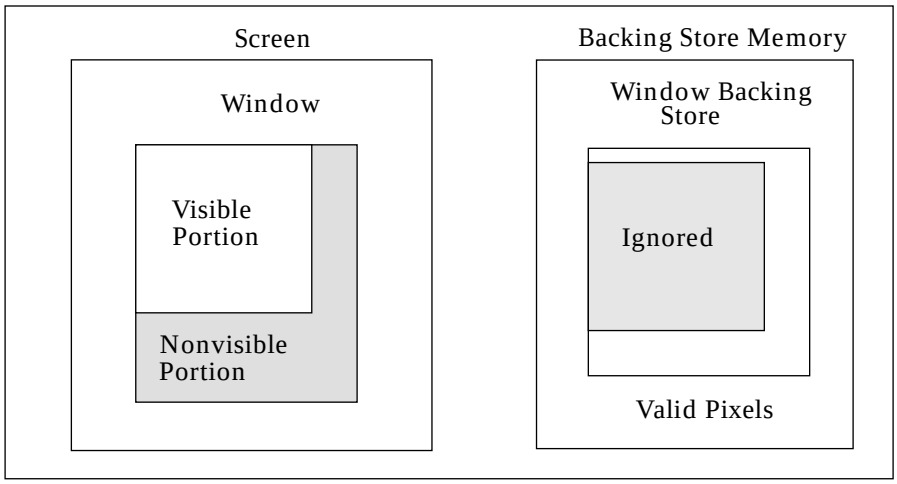


Figure 10-1 Screen and Backing Store Memory Relationship

The backing store always contains the contents of the nonvisible portion of the window. Not only is the DGA client supposed to render to the visible portion of the window, but it is also expected to keep the valid pixel area of the backing store up-to-date. The valid portion of the backing store always has the same shape as the nonvisible portion of the window. This shape is equal to the window's boundary shape minus the visible shape.

The backing store of a window is *not* a drawable itself. It can be rendered to and can be cached like a pixmap, but it cannot be separately grabbed. It has no XID of its own and no presence in the system independent of its owning window. Backing store can only be accessed by grabbing the window that owns it.

During each lock critical section, the amount of rendering the client must perform depends on the degree to which the window is obscured. `dga_draw_visibility` can be called to determine which of the following cases holds:

1. If the drawable is entirely unobscured (`DGA_VIS_UNOBSCURED`), the client can restrict rendering to just the visible shape of the drawable. This shape is returned by `dga_draw_clipinfo`.
2. If the window is partially obscured (`DGA_VIS_PARTIALLY_OBSCURED`), the client should render to both the visible and the retained portions.
3. If the drawable is completely obscured (`DGA_VIS_FULLY_OBSCURED`) then the client should render to the entire backing store area.

The client must complete rendering updates to both the drawable and backing store within a single lock critical section.

If the client needs to read pixels from the drawable, it should use the clip state of the drawable to determine whether it should read the pixels from the visible portion of the drawable, the backing store, or both. This is done in a similar fashion to rendering (described above).

By default, the shared information file for backing store is placed by the server in `/tmp` but because these files can tend to be rather large the server also supports placing the files in a path as defined by the `-sharedretainedpath` server command-line argument.

Compiling and Linking

To use this interface, the `/usr/openwin/include/dga/dga.h` file should be included in a library device handler's source file (it contains the definitions of many of the defined symbols and data structures referred to in this document).

The library device handler should be linked with the `/usr/openwin/lib/libdga.so` library.

Note – Routines with all uppercase names, such as DGA_DRAW_LOCK, are C macros—you cannot manipulate them as true C routines.

DGA Drawable Functions

Most DGA drawable routines can only be called when the drawable is locked. Otherwise, conflicts could occur with either the server or another client accessing the drawable. An inquiry routine called while the drawable is not locked may return invalid information. An action routine called while the drawable is not locked may not have the desired effect.

In the following routine specifications, if a routine must be called within a lock critical section, it is marked with the tag ”(Lock Only)”. The results of calling such a routine outside a lock critical section are undefined.

All other routines may be called either inside or outside of a lock critical section.

Initialization and Cleanup

The following routines initialize DGA, initiate and terminate direct access to a drawable, and cleanup DGA.

DGA_INIT

```
void
DGA_INIT()
```

Purpose This macro performs the initialization required to use any of the DGA interfaces: this drawable interface, the window compatibility interface, the colormap grabber, and the miscellaneous grabbers.

Called by All client programs before making any other DGA function calls. This macro can be called multiple times by a client program so that, multiple libraries using DGA can be used by the application program without difficulty.

XDgaGrabDrawable

```
Dga_drawable
XDgaGrabDrawable (dpy, drawid)
Display        *dpy;
Drawable       drawid;
```

Purpose Initiates direct access to a window or pixmap drawable. `drawid` is the XID of the window or pixmap. If the grab succeeds, a handle to the DGA client structure for the drawable is returned. If the grab fails or is refused by the server, 0 is returned.

Returns The returned `Dga_drawable` is used to form the handle to be passed to subsequent DGA inquiry routines on that drawable.

For a window, use buffer index equal to -1. Likewise for a pixmap.

If the client wants direct access to a multibuffer, it should first query MBX to determine the main window of the multibuffer set. It should then call `XDgaGrabDrawable` to grab this window. When locking the multibuffer or inquiring state for the multibuffer, the index of the multibuffer (received from MBX) should be passed to DGA along with the `Dga_drawable` of the main window.

Results This routine allocates several resources in the calling process's address space for the drawable, including a mapping of the shared memory information. This function opens a file descriptor for the correct graphics device file, using information found in the shared memory area. Only one file descriptor per graphics device will be opened.

Note – If the drawable is a member of a multibuffer set (a multibuffered window or one of its multibuffers) the grab will succeed only if the number of multibuffers is less than or equal to 16.

Note – One file descriptor is consumed when the client grabs a window by calling `DgaGrabDrawable`. If `dga_draw_rtngrab` is also called, an additional file descriptor is consumed. In addition, a single additional file descriptor is used whenever there are one or more pixmaps grabbed, or windows grabbed with one or more multibuffers. Since multibuffers that are not viewable can be assigned to windows subsequent to the grab, this file descriptor may not be actually consumed by `DgaGrabDrawable` itself, but rather, may be allocated during a later lock of the drawable. Finally, for each file descriptor used by the client, a file descriptor is consumed in the server.

XDgaUnGrabDrawable

```
int
XDgaUnGrabDrawable (dgadraw)
Dga_drawable   dgadraw;
```

Purpose

This function terminates direct access to a drawable. If this was the last direct use of the drawable by the client, DGA resources for the drawable in the client's address space are freed. These were the resources allocated by a previous call to `XDgaGrabDrawable`. All resources and memory mappings that were created are freed or made inaccessible as a result of this operation. If this was the last direct use of the drawable on the screen, the window server DGA resources for this drawable are also freed.

Results

If `dgadraw` refers to a multibuffered window, all multibuffers associated with this window are also ungrabbed.

If the drawable is locked at the time of this call, it is first unlocked.

If resources for backing store have been allocated for the drawable, these resources are freed. The shared memory mappings for the backing store in the calling process's address space are unmapped, the backing store shared info file is closed, and the server is notified to free all its resources associated with the direct access to backing store.

Returns Nonzero on success
0 on failure

Drawable Locking and Change Detection

The following functions provide the ability to gain exclusive access to a drawable while client operations are being performed. Routines are also provided to detect changes that have occurred to the drawable since the client last locked it.

DGA_DRAW_LOCK

```
void
DGA_DRAW_LOCK(dgadraw, bufIndex)
Dga_drawable   dgadraw;
short          bufIndex;
```

Purpose This macro locks the drawable info shared memory data structure. The client must lock the drawable info shared memory area before it uses any information in it. This restrains the window server from applying any modifications to the attributes of the drawable a client is rendering into it. It also prevents collisions with other clients. The lock should be held while any rendering is performed or information from the shared memory is being accessed. The lock is lightweight enough to be placed around a small number of primitives without sacrificing performance. Thus calls to the locking primitives should be kept in the graphics library and not exposed in the library API.

Results The current lock subject is the drawable (window, pixmap, or multibuffer) to which subsequent DGA inquiry routines executed within the lock apply.

If `dgadraw` is a multibuffered window, not only is the window locked but all the multibuffers in the current multibuffer set are locked as well.

Locks nest correctly. If DGA_DRAW_LOCK has been called multiple times without an intervening unlock, DGA_DRAW_UNLOCK must be called the same number of times before the drawable is unlocked.

Arguments

If the drawable to be locked is a non-multibuffered window or pixmap, bufIndex should be -1. The *current lock subject* (used within the lock critical section by other DGA routines) will be the window or the pixmap.

If the drawable to be locked is a multibuffer, dgadraw should be the Dga_drawable of the main window of this multibuffer. bufIndex should be its buffer index. The current lock subject will refer to this multibuffer.

DGA_DRAW_UNLOCK (Lock Only)

```
void
DGA_DRAW_UNLOCK(dgadraw)
Dga_drawable dgadraw;
```

Purpose

This macro permits external modification of the information in the shared memory data structure. Locke nest properly. This routine should be used only when a drawable has been first locked with DGA_DRAW_LOCK. If DGA_DRAW_LOCK has been called multiple times without an intervening unlock, DGA_DRAW_UNLOCK must be called the same number of times before the drawable is unlocked.

DGA_DRAW_LOCK_SRC_AND_DST

```
void
DGA_DRAW_LOCK_SRC_AND_DST(dgasrc, bufIndexSrc, dgadst,
       bufIndexDst)
Dga_drawable   dgasrc;
short          bufIndexSrc;
Dga_drawable   dgadst;
short          bufIndexDst;
```

Purpose

This macro atomically locks two drawables at the same time. It should be used when the client will be accessing two drawables in a rendering operation. An example of such an operation is a copy from the source drawable to the destination drawable. `dgasrc` must not be the same as `dgadst`. Furthermore, it is required that at least one of `dgasrc` or `dgadst` be a pixmap drawable. No failure status is returned if either of these conditions fails. For this macro, there are two current lock subjects, one for each `Dga_drawable`.

Results

The current lock subject is the drawable (window, pixmap, or multibuffer) to which subsequent DGA inquiry routines executed within the lock apply.

If either of the drawables is a multibuffered window, not only is the window locked, but all the multibuffers in the current multibuffer set are locked as well.

Locks nest correctly. If `DGA_DRAW_LOCK_SRC_AND_DST` has been called multiple times without an intervening unlock, `DGA_DRAW_UNLOCK_SRC_AND_DST` must be called the same number of times before the drawables are unlocked.

Arguments

If the drawable to be locked is a window or pixmap, the buffer index should be `-1`. The current lock subject of that drawable (used within the lock critical section by other DGA routines) will be the window or the pixmap.

If either of the drawables to be locked is a multibuffer, the `Dga_drawable` passed in should be that of the main window for the multibuffer. `bufIndex` should be its buffer index. The current lock subject will refer to this multibuffer.

When using this macro, make sure you call `DGA_DRAW_MODIF` for both `dgasrc` and `dgadst`, to synchronize with any changes that have occurred to either drawable.

DGA_DRAW_UNLOCK_SRC_AND_DST (Lock Only)

```
void
DGA_DRAW_UNLOCK_SRC_AND_DST(dgasrc, dgadst)
Dga_drawable dgasrc;
Dga_drawable dgadst;
```

Purpose This macro permits external modification of the drawable. This routine should be used only when the drawable was locked with `DGA_DRAW_LOCK_SRC_AND_DST`. Locks nest correctly. If `DGA_DRAW_LOCK_SRC_AND_DST` has been called multiple times without an intervening unlock, `DGA_DRAW_UNLOCK_SRC_AND_DST` must be called the same number of times before the drawables are unlocked.

Results If either the source or destination drawable is a multibuffer, the lock count for the entire multibuffer set is decremented, and if zero, all members of the multibuffer set are unlocked.

DGA_DRAW_MODIF (Lock Only)

```
int
DGA_DRAW_MODIF(dgadraw)
Dga_drawable dgadraw;
```

Purpose This macro checks to see if the current lock subject has been altered since the calling client locked it.

Called by The client must call this macro after locking, prior to rendering.

Returns Nonzero is returned if some state information has changed with which the client needs to synchronized.

If no change has occurred, or the client has been notified of all changes through notification call back routines, this routine returns zero.

If this macro returns nonzero and the client has *not* registered with DGA to receive change notifications, the client should call the following routines to detect changes to the drawable: `dga_draw_curshandle`, `dga_draw_sitechg`, `dga_draw_rtnchg` and `dga_draw_clipchg`. These routines should always be called in this order. (If the client has registered with DGA to receive a particular type of change notification by specifying a notification callback, do not call these routines.)

General Utility Functions

These routines allow the client to inquire various drawable attributes.

dga_draw_display

```
Display *
dga_draw_display(dgadraw)
Dga_drawable dgadraw;
```

Returns The display of a drawable that has been grabbed with `XGrabDrawable`.

dga_draw_id

```
Drawable
dga_draw_id(dgadraw)
Dga_drawable dgadraw;
```

Returns The XID of a drawable that has been grabbed with `XGrabDrawable`.

Note – This routine only returns the XID of a window or a pixmap. To determine the XID of a multibuffer, use this routine to inquire the XID of the main window and then use the MBX API routine, `XmbufGetWindowAttributes`, to inquire the multibuffer set information.

dga_draw_type

```
int
dga_draw_type(dgadraw)
Dga_drawable dgadraw;
```

Returns The type of the drawable client structure. The returned value is one of: `DGA_DRAW_WINDOW`, `DGA_DRAW_PIXMAP`, or `DGA_DRAW_OVERLAY`.

dga_draw_devname

```
char *
dga_draw_devname(dgadraw)
Dga_drawable dgadraw;
```

Returns A pointer to a null-terminated string representing the device name of the screen with which the grabbed drawable is associated.

dga_draw_devfd

```
int
dga_draw_devfd(dgadraw)
Dga_drawable dgadraw;
```

Returns The client's file descriptor for the screen with which the grabbed drawable is associated.

dga_draw_depth

```
int
dga_draw_depth(dgadraw)
Dga_drawable dgadraw;
```

Returns This routine returns the depth of the grabbed drawable.

dga_draw_set_client_infop

```
void
dga_draw_set_client_infop(dgadraw, client_info_ptr)
Dga_drawable dgadraw;
void *client_info_ptr;
```

Purpose This routine allows the client to set a pointer to client-specific data associated with `dgadraw`. This pointer could point to information that is local to the client alone.

dga_draw_get_client_infop

```
void *
dga_draw_get_client_infop(dgadraw)
Dga_drawable dgadraw;
```

Returns The client-specific data pointer associated with `dgadraw`. If this pointer was not set by the client, then this routine returns `NULL`.

dga_draw_devinfo (Lock Only)

```
void *
dga_draw_devinfo(dgadraw)
Dga_drawable dgadraw;
```

Returns This function returns a pointer to the device-specific information area the DGA shared information area for the current lock subject. The structure should be accessed by the client to inquire device-dependent information which is be shared between server and client. DGA routines do not interpret the device-dependent data but the client graphics library device-dependent code may need to. The size of this area is 132 bytes. The returned pointer is 4-byte aligned.

If the lock subject is cached, the device-dependent information can specify its location in the cache.

The format of this data area is completely device-dependent. The return pointer is NULL if the lock subject is not cached. An example of this structure could be:

```
struct {
short basex, basey;    /* drawable's position in dev. memory */
u_char mode;           /* a device specific mode */
u_char pad[2];
} Cache_Dev_Info;
```

Note – This routine returns a pointer to the `device_info` data member of the `dga_draw_dbinfo` structure. A pointer to this structure is returned by the buffer interface routine `dga_win_dbinfop`. This routine is still supported for compatibility with existing clients.

Drawable Sites

The routines in this section allow a client to detect site changes. Write the client to detect site changes for *all* types of drawables—all types of drawables can potentially undergo site changes.

- Pixmaps and nonviewable multibuffers can undergo site changes because they can become cached in device memory and alternately uncached.
- A site change can occur to a viewable multibuffer if the multibuffers in the multibuffer set for the main window are destroyed and then recreated. The multibuffer of the same buffer index in the multibuffer set may have a different address or viewability.
- Windows can also undergo site changes. But since a window may become multibuffered anytime after it is grabbed, and window aliasing of a multibuffer window can produce a site change, any window may potentially undergo a site change.

dga_draw_sitechg (Lock Only)

```
int
dga_draw_sitechg(dgadraw, reason)
Dga_drawable   dgadraw;
int            *reason;
```

Returns Nonzero if the current lock subject has undergone a change in site since the last time it was locked by this client. `dga_draw_site` can be called to inquire the site in which the drawable currently resides. The site can change for two reasons: either the site itself changed or the location within the site changed.

This routine should be called if `DGA_DRAW_MODIF` returns nonzero and the client has not registered a site change notification function.

Zero is returned if the last site and location within the site noted by the client still applies.

This routine returns valid results only the first time it is called after locking the drawable.

If nonzero is returned, `reason` indicates why the site change occurred. These are the possible values for this return argument:

DGA_SITECHG_INITIAL — A site change is always reported the first time a drawable is locked.

DGA_SITECHG_ZOMBIE — The site change occurred because the current lock subject is a zombie drawable (i.e. it's underlying X11 resource has been destroyed).

DGA_SITECHG_ALIAS — The site change is due to a change in the display buffer of the current lock subject from the previous lock subject. (This is only applicable to drawables that are members of an active multibuffer set).

DGA_SITECHG_CACHE — The site change is due to a change to the cache state of the current lock subject from the previous lock subject.

DGA_SITECHG_MB — The site change happened because the multibuffer set was changed (activated, deactivated, or replaced).

dga_draw_sitesetnotify

```
int
dga_draw_sitesetnotify(dgadraw, site_notify_func, client_data)
Dga_drawable       dgadraw;
DgaSiteNotifyFunc  site_notify_func;
void               *client_data;
```

Purpose Registers a function to be called by one of the drawable locking routines whenever a site change has occurred since the last lock of the drawable.

Arguments `client_data` is a client-specific data pointer that is given to the notification function as an argument.

`DgaSiteNotifyFunc` is defined as:

```
typedef void (*DgaSiteNotifyFunc)
                  (Dga_drawable, short, void *, int);
```

Description The calling sequence for a typical notification function is:

```
void
site_notify_func(dgadraw, bufIndex, client_data, reason)
Dga_drawable   dgadraw;
short          bufIndex;
void           *client_data;
int            reason;
```

The notification function should call `dga_draw_site` to determine the current site of the drawable.

`site_notify_func` will be called whenever a site change occurs to either the window or, if multibuffered, to any of its associated multibuffers. When the change has occurred to a window, `bufIndex` will be `-1`, otherwise it will be the index of the changed multibuffer.

When a site notification function is registered for a drawable, the client will receive notification of drawable site changes only through this function. `dga_draw_sitechg` will never return nonzero.

The site notification function is always called within the lock critical section. Therefore, care should be taken to not perform lengthy and time-consuming operations within it, such as system calls. Otherwise, the DGA lock time-out might expire, causing the lock to be prematurely broken.

dga_draw_sitegetnotify

```
void
dga_draw_sitegetnotify(dgadraw, site_notify_func, client_data)
Dga_drawable      dgadraw;
DgaSiteNotifyFunc *site_notify_func;
void              **client_data;
```

Returns The site notification function and client data for the drawable which was given to `dga_draw_sitesetnotify`. NULL is returned for both if this routine has not been called.

dga_draw_site (Lock Only)

```
int
dga_draw_site(dgadraw)
Dga_drawable   dgadraw;
```

Returns The site in which the current lock subject resides. Possible return values are:

```
DGA_SITE_SYSTEM
DGA_SITE_DEVICE
DGA_SITE_NULL
```

DGA_SITE_SYSTEM indicates the current lock subject resides in system memory (i.e. memory that is mapped into the client address space). In this case, the routines dga_draw_address, dga_draw_linebytes dga_draw_bitsperpixel return, respectively, the address of the origin pixel of the drawable, the inter-scanline stride (i.e. the number of bytes per scanline), and the number of bits per pixel.

DGA_SITE_DEVICE indicates the drawable resides in device memory. In this case, the return values of the routines dga_draw_address, dga_draw_linebytes, and dga_draw_bitsperpixel are invalid. Information about the exact location of the drawable within the site can be queried with dga_draw_devinfo. The data returned by this routine is device dependent and is not interpreted by DGA.

DGA_SITE_NULL means the underlying X11 resource for the drawable has been destroyed since the last time the drawable was locked. Refer to the section “Zombie Drawables” on page 204 for more details.

The site of a viewable drawable is always DGA_SITE_SYSTEM, unless it has been destroyed, in which case the site is DGA_SITE_NULL. The site of a nonviewable depends on whether or not it is cached.

dga_draw_address (Lock Only)

```
void *
dga_draw_address (dgadraw)
Dga_drawable   dgadraw;
```

Returns A pointer to the origin pixel of the current lock subject (x = 0, y = 0). A valid result is only returned when the site of the drawable is DGA_SITE_SYSTEM.

dga_draw_linebytes (Lock Only)

```
int
dga_draw_linebytes(dgadraw)
Dga_drawable   dgadraw;
```

Returns The value of the inter-scanline stride of the current lock subject. A valid result is only returned when the site of the drawable is `DGA_SITE_SYSTEM`.

dga_draw_bitsperpixel (Lock Only)

```
int
dga_draw_bitsperpixel(dgadraw)
Dga_drawable   dgadraw;
```

Returns The bits per pixel of the current lock subject. A valid result is only returned when the site of the drawable is `DGA_SITE_SYSTEM`.

Clipping State

The following functions enable clients to detect whether the clipping information of a drawable has changed and to synchronize with the new information.

dga_draw_clipchg (Lock Only)

```
int
dga_draw_clipchg(dgadraw)
Dga_drawable dgadraw;
```

Purpose If `DGA_DRAW_MODIF` returns nonzero, this routine should be called to determine if the clipping state for the current lock subject changed. Zero is returned if there were no such changes, otherwise nonzero is returned.

If a clipping change has occurred, the `dga_draw_bbox`, `dga_draw_visibility`, `dga_draw_empty` and `dga_draw_clipinfo` routines can be called to inquire the new clipping information.

Returns Valid information only the first time it is called after the drawable is locked.

dga_draw_bbox (Lock Only)

```
void
dga_draw_bbox(dgadraw, xp, yp, widthp, heightp)
Dga_drawable dgadraw;
int *xp, *yp, *widthp, *heightp;
```

Returns The screen coordinates of the upper left origin of the current lock subject and the width and height in the locations pointed to by the `xp`, `yp`, `widthp`, and `heightp` arguments. These values represent the shape of the bounding box of the drawable. If `dga_draw_visibility` returns `DGA_VIS_UNOBSCURED` and `dga_draw_singlerect` returns nonzero, the bounding box can be used to clip rendering rather than using the clip shape returned by `dga_draw_clipinfo`.

If the current lock subject is a window or multibuffer, the returned rectangle shape does not include any clipping of the window by other overlapping windows or multibuffers. For viewable drawables, the bounding box corresponds to the minimum and maximum x and y coordinates of the drawable. If the drawable is nonviewable, the x and y coordinates of the origin are (0, 0).

dga_draw_visibility (Lock Only)

```
int
dga_draw_visibility(dgadraw)
Dga_drawable dgadraw;
```

Returns Whether the drawable is fully obscured, partially obscured, or fully unobscured. Possible return values are:*)*

```
DGA_VIS_UNOBSCURED
DGA_VIS_PARTIALLY_OBSCURED
DGA_VIS_FULLY_OBSCURED
```

`DGA_VIS_UNOBSCURED` means the drawable is not obscured by any other drawable (i.e. children, siblings, or ancestors). `DGA_VIS_PARTIALLY_OBSCURED` means a proper subset of the drawable pixels are obscured by some other drawable. `DGA_VIS_FULLY_OBSCURED` means the entire drawable is obscured.

This routine is useful for deciding how much of the backing store of a window should be rendered. See section “Backing Store” on page 157 for more details.

dga_draw_empty (Lock Only)

```
int
dga_draw_empty(dgadraw)
Dga_drawable dgadraw;
```

Returns Nonzero if the current clipping shape of the current lock subject is empty, zero otherwise.

dga_draw_clipinfo (Lock Only)

```
short *
dga_draw_clipinfo(dgadraw)
Dga_drawable dgadraw;
```

Purpose

This routine is used to get the address of the clipping shape of the current lock subject. The clip shape is represented by a sequence of signed shorts which describes a sequence of rectangles. The data consists of a sequence of one or more (`ymin`, `ymax`) pairs, each of which is followed by a sequence of one or more (`xmin`, `xmax`) pairs. (`xmin`, `xmax`) sequences are terminated by a single value of `DGA_X_EOL`. (`ymin`, `ymax`) sequences are terminated by a single value of `DGA_Y_EOL`. `DGA_X_EOL` and `DGA_Y_EOL` are defined constants. This is best described with the following sample code:

```
short int x0, y0, x1, y1, *ptr;
ptr = dga_draw_clipinfo(dgadraw);
while((y0=*ptr++)!= DGA_Y_EOL) {
    y1 = *ptr++;
    while((x0=*ptr++)!= DGA_X_EOL) {
        x1 = *ptr++;
        printf("rectangle from (%d,%d)to (%d,%d)\n",x0,y0,x1,y1);
    }
}
```

Note that for each (`min`, `max`) pair, the `min` coordinate pixels are included in the clipping shape, but the `max` coordinate pixels are not (they are one pixel unit outside the clipping shape). The client should not modify the contents of the data area pointed to by the return value.

If the drawable is a window, this clip shape does not include the clipping shape of the children of the window.

If the drawable is a pixmap, the clip shape is always a single rectangle.

A NULL pointer is returned if the X resource referred to if the current lock subject no longer exists. In this case, all rendering to this drawable will be entirely clipped.

Note – It is recommended that `dga_draw_visibility` be used rather than the following two routines. However, these two routines are still provided for the convenience of programmers used to the older DGA window grabbing interface. These two routines are deprecated and will be removed in a future version of this interface.

dga_draw_singlerect (Lock Only)

```
int
dga_draw_singlerect(dgadraw)
Dga_drawable dgadraw;
```

Returns Nonzero if the current clipping shape of the current lock subject is a single rectangle, zero otherwise. Note that a clipping shape consisting of a single rectangle is not necessarily the same as the bounding box of the drawable.

dga_draw_obscured (Lock Only)

```
int
dga_draw_obscured(dgadraw)
Dga_drawable dgadraw;
```

Returns Nonzero if the current clipping shape of the drawable is the same as the full shape of the window without regard to overlapping windows, zero otherwise. At present, this routine returns valid information only for rectangular windows.

10

Dealing with Cursor Conflicts

The cursor image may conflict with rendering when the DGA client is about to perform. In these cases, the client must detect the conflict and take down the cursor image. Only then should the client render. The window system restores the cursor image after the client unlocks the drawable.

Some devices always render the cursor image in a plane group dedicated for that purpose. These devices never display viewable drawables in this plane group. On these types of devices, there will never be any cursor conflicts. These devices are called *dedicated cursor* devices.

Some devices always render the cursor image in a plane group in which viewable drawables also reside. In this case, each time a viewable drawable is locked, the DGA client must detect a cursor conflict and then deal with the conflict. These devices are called *software cursor* devices.

Always truncate the cursor on a hardware cursor device. This forces the cursor into hardware any time a window is grabbed.

Whether a DGA client must handle potential cursor conflicts depends, therefore, on the type of device. No cursor conflict handling is needed for dedicated cursor devices or hardware cursor register devices that always truncate large cursor images. On the other hand, conflict handling is required for software cursor devices or hardware cursor register devices that don't truncate.

Except on devices for which there will never be conflicts, DGA clients are required to call `dga_draw_curshandle` if, after a drawable is locked, `DGA_DRAW_MODIF` returns nonzero. This is the case for window and multibuffer drawables only. This is not required for pixmap drawables. If the cursor image currently intersects the pixels of the drawable, the cursor will be taken down.

dga_draw_curshandle (Lock Only)

```
void
dga_draw_curshandle(dgadraw, take_down_func, client_data)
Dga_drawable           dgadraw;
DgaCursTakeDownFunc    take_down_func;
void                   *client_data;
```

Purpose If the device is can have cursor conflicts, this routine should be called if, after locking a drawable, `DGA_DRAW_MODIF` returns nonzero. If there is a cursor conflict, this routine will take down the cursor.

Arguments `take_down_func` is a pointer to a client-supplied function which can take down the cursor by restoring the pixels that the cursor was rendered over. `client_data` is a pointer to arbitrary client data which will be passed to the client-supplied function. The calling sequence for a typical take-down function is defined by the following type:

```
typedef void (*DgaCursTakeDownFunc)(
                  void *,            /* client_data */
                  Dga_drawable,      /* dgadraw */
                  int, int,          /* x, y */
                  Dga_curs_memimage */* memimgp */
               );
```

where the `Dga_cur_memimage` structure is defined as follows:

```
typedef struct dga_curs_memimage {
    u_int width;
    u_int height;
    u_int depth;
    u_int linebytes;
    void *memptr;
} Dga_curs_memimage;
```

`take_down_func` should restore (`width*height`) pixels of depth `depth` stored at the locations pointed to by `memptr` to the screen starting at (`x`, `y`) relative to the window origin. Successive scanlines of the stored pixels are separated by `linebytes` bytes. The current possible depths are 1, 8, 32. Depth 1 pixels are packed 8 pixels per byte. Depth 8 pixels are packed 1 pixel per byte. Depth 32 pixels are packed 1 pixel per 4 bytes.

The cursor take-down function is always called within the lock critical section. Therefore, care should be taken to not perform lengthy and time-consuming operations within it, such as system calls. Otherwise, the DGA lock time-out might expire, causing the lock to be prematurely broken.

Note – `take_down_func` will only be called if the cursor needs to be taken down because it is currently up and intersects the pixels of the drawable. The overlap test is currently based on the bounding box of the drawable, not on the actual exposed shape.

Note – It is very important that `dga_draw_curshandle` be called after every window or multibuffer lock for which `DGA_DRAW_MODIF` returns nonzero. If the drawable is locked without checking `DGA_DRAW_MODIF` and calling `dga_draw_curshandle`, future locks of the drawable may not notice the cursor conflict.

Backing Store Routines

The following routines are provided for direct access to the backing store of a drawable. Currently, only windows have backing store.

dga_draw_rtngrab

```
int
dga_draw_rtngrab(dgadraw)
Dga_drawable   dgadraw;
```

Purpose This routine provides direct access to the backing store of a window. A window may have backing store either due to some client setting the `XWindowAttributes.backing_store` attribute of the window to `WhenMapped` or `Always`, or due to the window being occluded by a save-under window.

Returns Nonzero if direct access to the backing store of a window is permitted. In this case, the necessary client/server information sharing channel is established.

Zero is returned if the server denies access to backing store for the drawable or the routine otherwise fails.

The window does not need to actually have backing store at the time of the call. The backing store may be provided by the server at a later time. It is the responsibility of the client to always check for the presence of backing store. See section “dga_draw_rtnchg (Lock Only)” on page 184 for more on this.

Note – If a DGA client does not call this routine, or if it does call it, but the routine fails, the server assumes that the client is not updating the contents of the backing store when it renders. If this is the case, the server considers the backing store inconsistent when the drawable is unlocked. This may cause an exposure event to be sent for the drawable.

Note – Grabbing the backing store of a drawable consumes one file descriptor in the client and one file descriptor in the server.

dga_draw_rtnungrab

```
int
dga_draw_rtnungrab(dgadraw)
Dga_drawable   dgadraw;
```

Purpose This routine terminates direct access to backing store for the given window and frees any associated resources.

dga_draw_rtnchg (Lock Only)

```
int
dga_draw_rtnchg(dgadraw)
Dga_drawable   dgadraw;
```

Purpose This routine should be called if, after the window drawable is locked, `DGA_DRAW_MODIF` returns nonzero.

Returns Nonzero if the state of the drawable backing store has changed since the last time the drawable was locked. If nonzero is returned, `dga_draw_rtnactive` should be called to determine whether backing store is currently present. This is because the window server may attach or detach backing store at any time. If backing store is present, the client is required to update the contents of the backing store appropriately.

This routine returns valid information only the first time it is called after locking the drawable. To use this routine, `dga_draw_rtngrab` must have first been called on the drawable and the grab must have succeeded.

Another reason this type of change can happen is if the current lock subject of the window is actually a multibuffer. Since multibuffers don't have backing store in the current release, it might result in a reported retained change if the window itself has backing store. Another reason might be that a window with backing store was previously aliased but is no longer.

For initialization purposes, this routine will always return nonzero the first time it is called.

dga_draw_rtnactive (Lock Only)

```
int
dga_draw_rtnactive(dgadraw)
Dga_drawable   dgadraw;
```

Purpose This routine should be called each time `dga_draw_rtnchg` indicates a change occurred to the state of a window drawable's backing store; the server may have granted or taken away backing store from the windows.

Returns Nonzero if backing store is currently available to the drawable; otherwise zero.

A return value of zero indicates that backing store is not (or no longer) available for the drawable. In this case, the client does not need to update the backing store contents. Otherwise, the client should call the routines described below in order to update the backing store.

dga_draw_rtncached (Lock Only)

```
int
dga_draw_rtncached(dgadraw)
Dga_drawable   dgadraw;
```

Returns A nonzero value if the backing store is cached in hardware as opposed to being in system memory. If the return value is set to `DGA_RTN_NEW_DEV`, then it means that the server has re-cached the backing store from system memory to the hardware device associated with the drawable. If this is the case, then the name and type of the device may be obtained by calling `dga_draw_rtndevtype` (see page 187).

If the return value is set to DGA_RTN_SAME_DEV, then the backing store remains cached in the same device as previously recorded. If the backing store is not cached, DGA_RTN_NOT_CACHED is returned. DGA_RTN_NEW_DEV, DGA_RTN_SAME_DEV and DGA_RTN_NOT_CACHED are predefined constants.

dga_draw_rtndevinfop (Lock Only)

```
void *
dga_draw_rtndevinfop(dgadraw)
Dga_drawable   dgadraw;
```

Returns

A pointer to the device-specific shared backing store information when the backing store is cached. The pointer is invalid if the backing store is not cached. This structure contains device-specific information. This device-specific information is required because devices that support cached backing store may implement it differently. The pointer points to a memory area which is 8 bytes long and 4-byte aligned. An example of this structure could be:

```
struct {
short basex, basey; /* backing store's position on frame buffer */
u_char mode;        /* a device specific mode */
u_char pad[2];
} Shared_Retained_Dev_Info;
```

dga_draw_rtndevtype (Lock Only)

```
void
dga_draw_rtndevtype(dgadraw, type, name)
Dga_drawable   dgadraw;
u_char         *type;
char           **name;
```

Purpose This routine is used to obtain the shared backing store's hardware cache device type and name.

Arguments `type` is device dependent.

`name` should point to an array of characters. The returned name will be a maximum of 32 characters long, including a NULL terminator.

dga_draw_rtndimensions (Lock Only)

```
void
dga_draw_rtndimensions(dgadraw, width, height, linebytes)
Dga_drawable   dgadraw;
short          *width;
short          *height;
u_int          *linebytes;
```

Purpose This routine is used to obtain the dimensions of the shared backing store.

Arguments `linebytes` is valid only for non-cached backing store.

dga_draw_rtnpixels (Lock Only)

```
void *
dga_draw_rtnpixels(dgadraw)
Dga_drawable dgadraw;
```

Returns A pointer to the backing store's pixel memory. This pointer is valid only for non-cached backing store. The format of the shared memory backing store is the same as the pixmap format of the corresponding depth for the window's screen.

Colormap Grabber Interface

The following routines are the client's interface to the colormap grabber functions.

XDgaGrabColormap

```
Dga_token XDgaGrabColormap(dpy, cmap)
Display *dpy;
Colormap cmap;
```

Results This function grabs an existing X11 (virtual) colormap and creates server-side resources for sharing updates to it with the client. The window server is sent a protocol request to create a shared colormap information file.

Returns A token, which is used by the client to access the shared information.

An error code if the window system refuses the registration request. The request also fails if the DGA client and the server are not running on the same machine.

dga_cm_grab

```
Dga_cmap dga_cm_grab(devfd, token)
int devfd;
Dga_token token;
```

Purpose This function is similar to `dga_win_grab`, in that it maps a shared memory data structure and returns a pointer to a client-side structure.

Arguments `devfd` is the file descriptor of the graphics device on which the grabbed window is resident.

If the device is not known or not yet opened, the caller can pass in -1, and `dga_cm_grab` opens the correct device file, using information found in the shared memory area.

`token` must be obtained by a previous call to `XDgaGrabColormap`.

Returns A `Dga_cmap` handle if successful; NULL for failure.

The `Dga_cmap` structure contains client-specific information and a pointer to the shared memory. Thus, several clients can grab the same colormap.

dga_cm_ungrab

```
void dga_cm_ungrab(dgacmap,cflag)
Dga_cmap dgacmap;
int cflag;
```

Purpose This function releases resources allocated by a previous call to `dga_cm_grab`. All resources and memory mappings created by `dga_cm_grab` are freed or made inaccessible as a result of this operation. Call `XDgaUnGrabColormap` after invoking this function to free window server resources. If the `cflag` argument is a nonzero value, the graphic device file is also closed.

XDgaUnGrabColormap

```
int XDgaUnGrabColormap(dpy, cmap)
Display *dpy;
Colormap cmap;
```

Purpose This function releases server resources associated with a shared colormap by sending the window server a protocol-extension request.

dga_cm_devfd

```
int dga_cm_devfd(dgacmap)
Dga_cmap dgacmap;
```

Returns The client's file descriptor for the frame buffer with which the grabbed colormap is associated.

dga_cm_devinfop

```
void *dga_cm_devinfop(dgacmap)
Dga_cmap dgacmap;
```

Returns A pointer to a shared-memory area containing device-dependent colormap information. The pointer is guaranteed to be 4-byte aligned and points to an area of 132 bytes. On devices with multiple hardware colormaps, information regarding the identity of the hardware colormap associated with the grabbed X colormap could be stored here. This device-specific information is required because each device that supports multiple hardware colormaps implements it differently. Any device information that needs to be sent between the server device code and the client device code is stored here. Device-dependent server code stores information here that the client can read.

dga_cm_set_client_infop

```
void dga_cm_set_client_infop(dgacmap, client_info_ptr)
Dga_cmap dgacmap;
void *client_info_ptr;
```

Purpose This routine allows the client to set a pointer to client-specific data associated with `dgacmap`. This pointer could point to information that is local to the client alone.

dga_cm_get_client_infop

```
void *dga_cm_get_client_infop(dgacmap)
Dga_cmap dgacmap;
```

Returns The client-specific data pointer associated with `dgacmap`. If this pointer was not set by the client, then this routine returns NULL.

dga_cm_write

```
void dga_cm_write(dgacmap, index, count, red, green, blue,
       putfunc)
Dga_cmap dgacmap;
int index, count;
u_char *red, *green, *blue;
void (*putfunc());
```

Purpose This function requests that the colormap information in the `red`, `green`, and `blue` arrays in user data space be placed in the grabbed colormap referenced by the `dgacmap` argument, starting at `index`, for `count` entries.

Arguments `putfunc` is a client-supplied and device-dependent callback function that updates the hardware colormap when necessary.

The calling sequence for the callback routine is:

```
void putfunc(dgacmap, index, count, red, green, blue);
```

The purpose of calling the device-dependent routine indirectly through `dga_cm_write` is to ensure proper coordination with the server.

If the colormap is currently installed, then the new values are loaded into the appropriate hardware colormap via the client-supplied callback routine. If the X11 colormap is not currently installed, no hardware update is performed. The `putfunc` function is called only if the colormap is installed in hardware.

dga_cm_read

```
void dga_cm_read(dgacmap, index, count, red, green, blue)
Dga_cmap dgacmap;
int index, count;
u_char *red, *green, *blue;
```

Purpose

This function reads colormap information into the `red`, `green`, and `blue` arrays in user data space. The `dgacmap` argument describes which colormap to read from. The data is read, starting at `index`, for `count` entries. The information is read from the shared-memory representation of the X11 virtual colormap.

Multibuffering Grabber Interface

The following functions do not manipulate graphics device registers or device state. The developers of the graphics library device-dependent code that use these routines are responsible for all manipulations of a particular graphics device, and for providing callback routines that are called from within these functions. The callback routines can get to device-dependent information stored in shared memory.

Some of the following functions let the client communicate with the server with shared memory, which buffer it is using for pixel reads and writes, and for display. The server uses this information to select the buffer for Xlib rendering, so that applications that mix server rendering with DGA rendering in the same window behave properly.

dga_draw_db_grab

```
int dga_draw_db_grab(dgadraw, nbuffers, vrtfunc, vrtcounterp)
Dga_drawable dgadraw;
int nbuffers;
int (*vrtfunc)(Dga_drawable);
u_int *vrtcounterp;
```

Purpose This function requests the window system to provide multibuffering services for the grabbed drawable named in the `dgadraw` argument. The call requests `nbuffers` to be allocated to the client. This drawable must have been grabbed previously via `XDgaGrabDrawable`. The call to `XDgaGrabDrawable` yields a handle, `dgadraw`, which is used in this call. The window server initializes the portion of the shared memory drawable information area that relates to multibuffering.

Returns Zero if the window system refuses the registration request; a nonzero value upon success. Each graphics device supports a small number of buffers. If more buffers are specified than the device can support, this call fails.

Arguments `vrtfunc` is a client-supplied function that blocks the client process until (at least) the beginning of the next vertical retrace period. This function is called when operations dependent on the vertical retrace period are performed on `dgadraw`. If this pointer is NULL, the functions `dga_draw_db_display` (page 197), `dga_draw_db_interval_wait` (page 198) and

`dga_draw_db_interval_check` (page 198) will not perform accurate timing of the minimal interval between buffer swaps. The calling sequence of this function is:

```
void *vrtfunc(dgadraw);
```

Often, `vrtfunc` is implemented as an ioctl to the associated device driver which, in turn, blocks until (at least) the beginning of the next vertical retrace period. The `vrtfunc` function might require the file descriptor of the device or information stored in the client-private data area and can obtain the information with `dga_draw_devfd` and `dga_draw_get_client_infop`.

`vrtcounterp` is a pointer to a client-supplied free-running vertical retrace counter. Often, this counter is implemented as a read-only device register that can be mapped into the client's address space or a memory location mapped into the client's space which is incremented by the associated device driver at vertical retrace interrupt. The DGA functions only read this counter. If this pointer is NULL, the functions `dga_draw_db_display`, `dga_draw_db_interval_wait` and `dga_draw_db_interval_check` will not perform accurate timing of the minimal interval between buffer swaps.

dga_draw_db_ungrab

```
int dga_draw_db_ungrab(dgadraw)
Dga_drawable dgadraw;
```

Returns

A previously multibuffered window to single-buffer mode. It also frees server and client multibuffering resources associated with `dgadraw`.

If any of these steps fail, zero is returned. A nonzero value is returned upon success.

dga_draw_db_write

```
void dga_draw_db_write(dgadraw, buffer, writefunc, data)
Dga_drawable dgadraw;
int buffer;
int (*writefunc)(void*, Dga_drawable, int);
void *data;
```

Purpose This function selects which buffer is written to when the client program draws to a multibuffered window.

Arguments The device-dependent callback routine for setting the write buffer, `writefunc`, is supplied by the client program.

Permissible values for the `buffer` argument are small integers (from 0 to nbuffers-1). No function will be called if `writefunc` is NULL. The calling sequence of the callback routine is:

```
int writefunc(data, dgadraw, buffer);
```

The purpose of calling the device-dependent routine indirectly through `dga_draw_db_write` is to ensure proper coordination with the server. The application program uses `data` to pass private information to (and from) `writefunc` (but the `data` argument is now redundant, since `dga_draw_get_client_infop` has been added to the interface).

dga_draw_db_read

```
void dga_draw_db_read(dgadraw, buffer, readfunc, data)
Dga_drawable dgadraw;
int buffer;
int (*readfunc)(void*, Dga_drawable, int);
void *data;
```

Purpose This function selects which buffer is read from when the client program reads pixel values from a multibuffered window.

Arguments The device-dependent callback routine for setting the read buffer, `readfunc`, is supplied by the client program.

Permissible values for the `buffer` argument are small integers. No function will be called if `readfunc` is NULL. The calling sequence of the callback routine is:

```
void readfunc(data, dgadraw, buffer);
```

The purpose of calling the device-dependent routine indirectly through `dga_draw_db_read` is to ensure proper coordination with the server. Use `data` to pass private information to (and from) `readfunc` (but the `data` argument is redundant since `dga_draw_get_client_infop` has been added to the interface).

dga_draw_db_display

```
void dga_draw_db_display(dgadraw, buffer, visfunc, data)
Dga_drawable dgadraw;
int buffer;
int (*visfunc)(void*, Dga_drawable, int);
void *data;
```

Purpose

This function causes `buffer` to become visible. A device-dependent callback routine for making this buffer visible is supplied by the caller in the form of the `visfunc` routine. This routine gets the values of `data`, `dgadraw` and `buffer` as arguments.

Arguments

No function is called if `visfunc` is NULL. Use `data` to pass private information to (and from) `visfunc` (but the `data` argument is now redundant, since `dga_draw_get_client_infop` has been added to the interface).

This routine performs the following steps:

```
call dga_draw_db_interval_check();
if (interval isn't exhausted)
    call dga_draw_db_interval_wait();
call (*visfunc(data, dgadraw, buffer));
```

This function first waits for the minimum display interval to elapse for the previous frame (if it has not already done so). Then, `visfunc` makes the named buffer visible. After calling `visfunc`, this function returns. `visfunc` need *not* block until the buffer is actually visible. It is up to the device-rendering routines to ensure that buffer flip has been completed before commencing rendering. Typically, the actual visibility of the new buffer will be delayed until the next vertical retrace. This means that rendering a subsequent frame to the old buffer might need to be delayed until the next retrace. A client can check to see if the operation is completed by calling the `dga_draw_db_display_done` routine (see page 200).

The purpose of calling `visfunc` indirectly through this routine is proper coordination with the server and maintenance of the buffer swap timing and vertical retrace synchronization.

dga_draw_db_interval

```
void dga_draw_db_interval(dgadraw, interval)
Dga_drawable dgadraw;
int interval;
```

Purpose This function establishes a timed delay between buffer swaps.

Arguments `interval` specifies in milliseconds the minimum delay between successive buffer swaps. The default interval is one refresh period. Assigning a negative value to `interval` results in the interval being set to the default interval. The exact duration of the default interval depends on the frequency characteristics of the monitor.

dga_draw_db_interval_wait

```
void dga_draw_db_interval_wait(dgadraw)
Dga_drawable dgadraw;
```

Results This function blocks the calling process until the minimum display interval time is exhausted.

dga_draw_db_interval_check

```
int dga_draw_db_interval_check(dgadraw)
Dga_drawable dgadraw;
```

Purpose This Boolean function indicates whether the minimum display time has elapsed since a buffer flip was requested.

Returns A nonzero value if the time has elapsed, zero if there is still time left.

dga_draw_db_write_inquire

```
int dga_draw_db_write_inquire(dgadraw)
Dga_drawable dgadraw;
```

Purpose This function is used to determine the state of multibuffering on a window, and indicates which buffer is selected for writing.

Returns The buffer number.

dga_draw_db_read_inquire

```
int dga_draw_db_read_inquire(dgadraw)
Dga_drawable dgadraw;
```

Purpose This function is used to determine the state of multibuffering on a window, and indicates which buffer is selected for reading.

Returns The buffer number.

dga_draw_db_display_inquire

```
int dga_draw_db_display_inquire(dgadraw)
Dga_drawable dgadraw;
```

Returns The buffer number of the visible buffer.

dga_draw_db_display_done

```
int dga_draw_db_display_done(dgadraw, flag, display_done_func)
Dga_drawable dgadraw;
int flag;
int (*display_done_func)(Dga_drawable);
```

Purpose This function checks to see if the new buffer is visible after a previous call to `dga_draw_db_display`. If the `flag` is set to zero, it performs a polling function. In this case, the function returns a nonzero value if the buffer has been switched, zero otherwise. If the flag is set to a nonzero value, the function blocks until the buffer has switched. In this case, a nonzero value is returned once the switch has occurred, -1 on error.

Arguments `display_done_func` is a non-blocking routine that returns 1 when the new buffer is visible, 0 when the new buffer is not yet visible and -1 on error. The calling sequence of this function is:

```
(*display_done_func)(dgadraw);
```

dga_draw_db_dbinfop

```
Dga_dbinfo *dga_draw_db_dbinfop(dgadraw)
Dga_drawable dgadraw;
```

Returns A pointer to the multibuffering area in the DGA shared memory. The structure can be accessed for device-dependent information that must be shared between server and client. DGA routines do not interpret device-dependent data, but your graphics library device-dependent code might. This

structure also contains information regarding the hardware window ids associated with multibuffered windows. The definition of this structure is in the file `dga.h`.

```
typedef struct dga_dbinfo {
    short number_buffers;
    short read_buffer;
    short write_buffer;
    short display_buffer;
    u_int reserved_1;          /* for the future */
    u_int reserved_2;          /* for the future */
    u_int reserved_3;          /* for the future */
    u_int reserved_4;          /* for the future */
    u_int reserved_5;          /* for the future */
    u_int wid;                 /* db window id */
    u_int reserved_6;          /* for the future */
    u_char device_info[132];
} dga_dbinfo;
```

Miscellaneous Grabbers

The following routines define the client interface to the window id grabber, fast clear plane grabber, stereo grabber and Z buffer grabber. These grabbers may require specialized hardware.

Note – Currently, miscellaneous grabbers only work for windows; they will fail for pixmaps.

XDgaDrawGrabWids

```
int
XDgaDrawGrabWids(dpy, drawid, nwids)
Display dpy;
Drawable drawid;
int nwids;
```

Purpose

Some graphics devices control video display characteristics and/or hardware clipping via a control plane group called a window ID (WID) plane group. Normally WIDs are allocated and managed by the server. In some cases, DGA clients can make use of multiple WIDs for a single window to optimize some operation. `XDgaDrawGrabWids` is called to allocate `nwids` consecutive WID's. The window must have previously been grabbed via `XDgaGrabDrawable`.

Returns

Zero for failure; nonzero for success.

If successful, the WID values can be obtained from the shared memory via the `dga_draw_widinfop` (see page 203) routine. WIDs are 16-bit unsigned integer values. The base WID will be aligned on a power-of-two boundary which is determined by rounding up `nwids` to the next power of two.

To release the allocated WIDs, call `XDgaDrawGrabWids` with an `nwids` argument of zero.

dga_draw_widinfop

```
Dga_widinfo *
dga_draw_widinfop(dgadraw)
Dga_drawable dgadraw;
```

Returns A pointer to the `dga_widinfo` structure in the shared memory area for `dgadraw`. The structure is defined as follows and can also be found in the file `dga.h`:

```
typedef struct dga_widinfo {
short w_number_wids; /* number contiguous block wids allocated */
short w_start_wid;  /* starting wid of the wid block */
short w_wid;        /* current drawing wid */
short reserved_1;   /* for the future */
} Dga_widinfo;
```

In this structure, `w_number_wids` is the total number of wids that have been allocated as a contiguous block. `w_start_wid` is the starting window id value of this block. `w_wid` is the value of the window id currently being used for the window associated with the handle `dagdraw`.

XDgaDrawGrabFCS

```
int
XDgaDrawGrabFCS(dpy, drawid, nfcs)
Display *dpy;
Drawable drawid
int nfcs;
```

Purpose Some graphics devices have a feature called fast clear sets which can effectively speed up the clearing of the back buffer in a multibuffering application. Fast clear sets are scarce resources dedicated to a particular window. `XDgaDrawGrabFCS` is used to request one or more fast clear sets. The window must have previously been grabbed via `XDgaGrabDrawable`. The function returns zero for failure,

nonzero for success. If successful, the FCS values can be obtained from the shared memory via the `dga_draw_db_dbinfop` routine, described in a previous section. The FCS information will be stored in the device-dependent section (`device_info`) of the `dga_draw_dbinfo` structure. To release the allocated FCSs call `XDgaDrawGrabFCS` with an `nfcs` argument of zero.

XDgaDrawGrabStereo

```
int
XDgaDrawGrabStereo(dpy, drawid, st_mode)
Display *dpy;
Drawable drawid
int st_mode;
```

Purpose Some graphics devices are capable of stereo display of images. This function is used to inform the server that a particular window will be using stereo display. The window must have previously been grabbed via `XDgaGraDrawable`.

Returns Zero for failure, nonzero for success.

Arguments `st_mode` is 1 to enable stereo, 0 to disable it.

Zombie Drawables

There is nothing to prevent an X11 drawable resource from being destroyed at any time by an X11 client. Even if the underlying drawable resource is destroyed, a DGA client may still hold a handle to the drawable in the form of a `Dga_drawable` client structure. A `Dga_drawable` window or pixmap whose underlying X11 resource has been destroyed is called a *zombie*. A multibuffer can also be a zombie if the buffer index specified by the client is outside the range of multibuffers in the current multibuffer set. This can be the case if the main window is no longer multibuffered or the buffer index is greater or equal to the current number of multibuffers.

The first time a client locks a zombie drawable after its underlying X11 resource has been destroyed, a site change is reported. The site will be reported as `DGA_SITE_NULL`. In addition, `dga_draw_clipinfo` always returns NULL for a zombie drawable.

Note – When an MBX application changes the number of multibuffers in a multibuffer set, it must first destroy all of the existing multibuffers and then create new ones. Because multibuffers in the DGA drawable interface are identified with a buffer index, it is possible for the index that identified a certain multibuffer in the old set, to now refer to a new one. To avoid this type of aliasing, client libraries should avoid rendering into multibuffers that have been destroyed. Presently, client libraries themselves need to make this determination with help from the application.

DGA Overlays

The DGA overlay interface allows direct access to windows in overlay planes. To render in overlay windows, the client must be able to manipulate the device's visibility planes. When overlay windows are in the same plane group as other windows, they are *in conflict*. Direct rendering to overlay windows in conflict is not allowed. A client may inquire the *overlay state* by calling `dga_draw_ovlstate` to determine whether the overlay windows supported on that device are in conflict with other windows.

Note – Currently, direct rendering to backing store associated with overlay windows is not supported. Future versions of the DGA interface will provide this feature.

Direct access to overlay windows follows the same locking rules as other windows. `dga_draw_type` (on page 167) returns `DGA_DRAW_OVERLAY` for a grabbed overlay window.

Note – The following new functions are specific to overlay windows and should only be called when the drawable holds the lock.

---

dga_draw_ovlstate

```
int
dga_draw_ovlstate(dgadraw)
Dga_drawable dgadraw;
```

Purpose

Use this function to determine whether to render directly to an overlay window. It returns the overlay state for `dgadraw`.

To render to an overlay, clients need to get additional device-specific information. Get this information from the device's `devinfo` pointer with `dga_draw_devinfo` (see page 169).

Returns

`DGA_OVLSTATE_SAFE`

If the return state is `DGA_OVLSTATE_SAFE`, render both opaque and transparent paint on the window using the device-specific information.

`DGA_OVLSTATE_MULTIWID`

If the return state is `DGA_OVLSTATE_MULTIWID`, render with opaque paint using the device-specific information. Most devices are unable to provide enough information for the client to successfully render transparent paint on its own. In this case, use X11 protocol requests to render transparent paint to the overlay.

`DGA_OVLSTATE_CONFLICT`

If the return state is `DGA_OVLSTATE_CONFLICT`, the client cannot render either opaque or transparent paint to the overlay.

dga_draw_ovlstatechg

```
int
dga_draw_ovlstatechg (dgadraw)
Dga_drawable dgadraw;
```

Purpose Indicates whether the overlay state has changed since the last time the drawable was locked. If the `DGA_DRAW_MODIF` macro indicates that an overlay has been altered, use this function to see if the overlay's state has changed.

Returns `1` if the overlay state has changed; `0` otherwise.

dga_draw_ovlstatesetnotify

```
void
dga_draw_ovlstatesetnotify(dgadraw, ovlstate_notify_func,
       client_data)
Dga_drawable dgadraw;
DgaOvlStateNotifyFunc ovlstate_notify_func;
void *client_data;
```

Purpose Allows the client to set a pointer to a user-specified overlay state change notification function associated with `dgadraw`. It is automatically called during lock and `MODIF` flag processing if the overlay window's conflict state has changed.

dga_draw_ovlstategetnotify

```
void
dga_draw_ovlstategetnotify(dgadraw, pOvlstate_notify_func,
    pClient_data)
Dga_drawable dgadraw;
DgaOvlStateNotifyFunc *ovlstate_notify_func;
void **client_data;
```

Purpose Returns the previously set overlay state change notification function.

Returns `NULL` if no function has been set.

Direct Graphics Access Drawable DDX Interface 11

This chapter describes routines the server provides for you to interface with DGA with your DDX handlers to make various types of changes to a drawable. This interface is called the direct graphics access (DGA) drawable DDX interface.

Note – It is strongly recommended that you upgrade your DDX handler to use the DGA drawable interface. If you do not upgrade your DDX handler to work with the drawable interface, see "New Features and Changes" on page xxiii for detailed information about functions that are still supported.

The DGA initialization function defined in the previous release, `DgaDevFuncsInit`, is still supported. This routine allows grabbing of windows *only*. `DgaDevFuncsInit` and the new initialization function, `dgaScreenInit` should never be used by a DDX handler at the same time.

Initializing Drawable Grabs

The latest version of the DGA applications programmer's interface (API) in the SDK supports direct access to window, pixmap, and multibuffer drawables. In the initialization sequence that supports arbitrary drawable types, not only is this same function vector given to DGA, but two new functions are also given. Providing these new functions is optional. If they are `NULL`, the DGA drawable interface (`dga_draw_`*xxxx* API routines) is only able to grab window drawables.

Call the following initialization routine from the `InitOutput` routine of your DDX handler.

dgaScreenInit

```
int
dgaScreenInit(pScreen, pDgadevfuncs, major, minor)
ScreenPtr  pScreen;
void       *pDgadevfuncs;
int        major;
int        minor;
```

Arguments `pDgadevfuncs` is a function vector of device-dependent functions cast to a void*:

```
typedef struct _DgaDevFuncsDraw {
int            (*DgaAvail)();
void           (*GrabDrawable)(DrawablePtr);
void           (*UngrabDrawable)(DrawablePtr);
int            (*CacheDrawInit)(DrawablePtr);
int            (*CacheDrawCleanup)(DrawablePtr);
int            (*DbSetup)(WindowPtr, WXINFO*, int, Bool);
int            (*WidSetup)(WindowPtr, int, WXINFO*);
int            (*FcsSetup)(WindowPtr, WXINFO*, int);
int            (*ZbufSetup)(WindowPtr, int, WXINFO*);
int            (*StereoSetup)(WindowPtr, int, WXINFO*);
int            (*ChokeFb)(ScreenPtr, Bool);
int            (*SyncDrawable)(DrawablePtr,GCPtr);
int            (*UnsyncDrawable)(DrawablePtr,GCPtr);
int            (*CmapSetup)(CmapPtr, Grabbedcmap*)
} DgaDevFuncsDraw;
```

The `pDgadevfuncs` argument may be NULL. If so, it means that client DGA is not available on the device.

The device handler is not required fill out all members of `devFuncs`; some functions may not be applicable to a device and these entries should be NULL in the vector.

The `major` and `minor` arguments are the major and minor version numbers for the DDK release as specified in “DDX Versioning” on page 9.

All of the types and structures listed above are defined in the include file `dga/dgawinstr.h`.

Device-Supplied Routines

Use the following routines during DGA initialization. Values can be `NULL`; however, functionality might be limited.

DgaAvail

```
int (*DgaAvail)()
```

Purpose This function advertises the flavor of DGA that a device supports. If this function is NULL, the device is considered to not support client DGA. All devices supporting client DGA must supply this routine.

Returns The definitions of the return codes are found in `dga/dgawinstr.h`.

If a device does not support DGA, this routine should return `DGA_AVAIL_NONE`.

If the device supports DGA and also has a cursor that is always rendered in hardware, it should return `DGA_AVAIL_CURS_HW`.

If the cursor is always rendered in software, this routine should return `DGA_AVAIL_CURS_SW`.

A device that has a limit to the size of cursor that can be drawn in hardware and intends to support larger cursors in software, this routine should return `DGA_AVAIL_CURS_HW_SW`.

For example, on the GX/GX+, the maximum size for a hardware cursor is 32x32. If a client loads in a cursor that is larger than this, the GX switches to software to render this cursor. So, GX/GX+ would return `DGA_AVAIL_CURS_HW_SW` from this routine.

GrabDrawable

```
void (*GrabDrawable)(DrawablePtr pDraw)
```

Purpose This function is called when a drawable is first grabbed to allow the device handler to initialize device-dependent information for the drawable. See *Server-Supplied Multibuffering Routines* for routines to update the device-dependent information area of a drawable. Also, this section describes routines that should be called if the drawable is cached.

Note – This function is only called the *first* time a client grabs the drawable. It is never called for subsequent attempts to grab the same drawable, either by the client to first grab or other clients. Likewise, `UngrabDrawable` is only called when the last grabbing client ungrabs.

Note – This function is called on the first grab, even if the drawable is a window that is being grabbed through the older version of the DGA interface, the Window Compatibility Interface. In this case the `WindowPtr` is cast to a `DrawablePtr`. .

UngrabDrawable

```
void (*UngrabDrawable)(DrawablePtr pDraw)
```

Purpose This function is called when a drawable is ungrabbed. It should undo anything that `GrabDrawable` has done. For example, the device-specific shared information may need to be updated.

Note – This function is called on the first grab, even if the drawable is a window that is being grabbed through the older version of the DGA interface, the Window Compatibility Interface. In this case the `WindowPtr` is cast to a `DrawablePtr`. See “Window Grabber Supported Functions” on page xxiv.

CachedDrawInit

```
int (*CachedDrawInit)(DrawablePtr pDraw)
```

Purpose This function allows the device handler to do any device-specific setup needed for the drawable when it is cached. Examples include: location within the cache and the format of the data within the cache.

This routine is called for drawables that may be cached in special device memory. Drawable types that can cached include: pixmaps, nonviewable multibuffers, and the backing store of a window.

Note – Drawable refers to backing store in this context, even though a backing store is technically not a *drawable* because it doesn’t have an XID.

The type of drawable may be determined by inspecting `pDraw->type`. If this is `DRAWABLE_WINDOW`, the type of drawable that is being referred to is the drawable’s backing

store. The server-internal structure for this backing store (which, incidentally, happens to be of type `PixmapPtr`) can be derived using the expression:

```
((miBSWindowPtr)((WindowPtr)pDraw)->backStorage)->pBackingPixmap
```

If the type is `DRAWABLE_PIXMAP`, then the routine `DgaMbIsMultibuffer` should be called to determine if the drawable is a pixmap or a multibuffer.

Results

If the drawable is cached, this routine should do the following:

1. Call `DgaCacheDescribeDev` on the `pScreen` of the drawable with `devCode` and `devname`.

2. Call `DgaCacheStateChange` with a value of `TRUE`.

3. Call `DgaDevInfoGet` and `DgaDevInfoChange` to update any device-dependent information which is necessary for the cached drawable.

After this routine has been called, whenever the device handler changes the cache state of the drawable, it should call these routines.

Returns

If this routine returns 0, DGA assumes that the drawable is of type `DGA_DRAW_SYSTEM` and it copies the contents of the pixmap to the shared page.

This routine should return 1 if the drawable is not of type `DGA_DRAW_SYSTEM`, or the device handler has already copied the pixmap to the shared page.

CachedDrawCleanup

```
int
(*CachedDrawCleanup)(DrawablePtr pDraw)
```

Purpose

This function is called when a nonviewable drawable or backing store is ungrabbed. It should undo anything done by `CachedDrawInit`. For example, it would call `DgaCacheStateChange` to mark the drawable as uncached. `DgaDevInfoGet` and `DgaDevInfoChange` might need to be called to clean up information in the device-dependent shared area.

The type of drawable might be determined by inspecting `pDraw->type`. If this is `DRAWABLE_WINDOW`, the type of drawable being referred to is the drawable's backing store. The server-internal structure for this backing store can be derived using the expression:

```
((miBSWindowPtr)((WindowPtr)pDraw)->backStorage)->pBackingPixmap
```

If the type is `DRAWABLE_PIXMAP`, then the routine `DgaMbIsMultibuffer` should be called to determine if the drawable is a pixmap or a multibuffer.

Returns

`1` on success; `0` on failure. If `0` is returned, DGA assumes the drawable (or backing store) is uncached and directs its data pointer at the shared page. At this time, the contents of the drawable (or backing store) are copied to the shared page.

DbSetup

```
int
(*DbSetup)(WindowPtr pWin, WXINFO *infop, int num_buf,
        Bool flag)
```

Purpose

This function is called when an application requests direct access to do multibuffering. Typically, this function would update some device-specific structures / hardware states, as well as information on the shared info page.

Arguments

The `WXINFO` structure has a field, `wx_dbuf`, which is a structure containing information relevant to multibuffering. The definitions of these structures are found in `dga/dgawinstr.h`.

This function must update the following structures:

`infop->wx_dbuf.num_buffers` should be set equal to the total number of buffers that the device supports in hardware. If the number of buffers available from the device is less than the requested number, `num_buf`, this function should return failure (0).

MPG Devices with hardware window ids can allocate a new window id for the multibuffered window. If so, this function is responsible for repreparing the window with the new (hardware) window id. If a new and unique WID is allocated for this window, the `infop->wx_dbuf.WID` field should be updated with this new value and the `infop->wx_dbuf.UNIQUE` flag should be set to 1 to indicate that this is a unique window id. See Chapter 5, “Multiple Plane Group Interface” for more information.

The `wx_dbuf` structure contains a device-specific field, `wx_dbuf->device`, that can be used by the device to communicate information between the server and the client. In the `wx_dbuf` structure, this is declared as:

```
union { char pad[128];} device
```

Each device can cast this to its own structure and communicate information to the client.

`infop->w_refresh_period` should be set equal to the refresh period of the monitor in milliseconds. This information is required by client-side DGA code. If this value is not supplied (set to zero), the client-side code defaults to a 66Hz monitor.

Returns 1 on success; 0 on failure.

WidSetup

```
int
(*WidSetup)(WindowPtr pWin, int num_wids, WXINFO *infop)
```

Purpose This function is called when an application requests a block of window ids to be grabbed. The allocation of window ids is device specific and should be handled by this routine.

Results On MPG devices, the window might need to be reprepared after new window ids are allocated. This routine should take care of the repreparation as well.

Arguments This routine should update information in the DGA shared page pertaining to window ids:

`infop->w_number_wids` should be set equal to the number of contiguous wids, `num_wids` that have been allocated. If the device was not able to allocate the requested number of contiguous wids, this function should return 0 for failure.

`infop->w_start_wid` should be set equal to the value of the first WID in the newly allocated block. The base WID should be aligned on a power-of-two boundary.

`infop->w_wid` should be set equal to the current WID of the window. This is often equal to `infop->w_start_wid`.

If the window has been allocated a new window id, this function is responsible for repreparing the window with this WID value. See Chapter 5, “Multiple Plane Group Interface” for details on how to do this.

Returns `1` on success; `0` on failure.

FcsSetup

```
int
(*FcsSetup)(WindowPtr pWin, int num_fcs, WXINFO *infop)
```

Purpose This function is called when an application requests a number of fast clear planes, `num_fcs`, to be grabbed for a window, `pWin`. The allocation of fcs planes is device-specific and should be handled by this routine.

On MPG devices, allocation of FCS planes may require repreparation of the window. This function is responsible for repreparation. See Chapter 5, “Multiple Plane Group Interface” for more details about accessing the MPG information.

Arguments This routine should update the information in the DGA shared page pertaining to fast clear planes. Information about a window’s fast clear planes is stored in the device-specific portion of the `wx_dbuf` structure found in the `WXINFO` structure `infop->wx_dbuf.device`. This structure can be cast to a device-defined structure and the fcs information could be stored here.

Returns 1 on success; 0 on failure.

ZbufSetup

```
int
(*ZbufSetup)(WindowPtr pWin, int zbuf_type, WXINFO *infop)
```

Purpose This function is called when an application requests direct access to the Zbuffer for a window, `pWin`. This is a device-specific operation and should be handled by this routine.

Arguments This routine should update the device-specific information in the DGA shared page pertaining to Zbuffer. A device may support various types of Z buffers and the second argument, `zbuf_type`, indicates which type of Zbuffer is being requested. Each device may support different types of Z buffers.

Information about a window's `Zbuffer` is stored in the device-specific portion of the `wx_dbuf` structure found in the `WXINFO` structure `infop->wx_dbuf.device`.

This array can be cast to a device-defined structure and the `Zbuffer` information could be stored here. On MPG devices, allocation of `Zbuffer` may require repreparation of the window. This function is responsible for repreparation. Please see Chapter 5, "Multiple Plane Group Interface" for more details about accessing the MPG information.

Returns `1` on success; `0` on failure.

StereoSetup

```
int
(*StereoSetup)(WindowPtr pWin, int st_mode, WXINFO *infop)
```

Purpose This function is called when an application requests that a stereo mode be associated or disassociated with this window, `pWin`.

Arguments If the second argument, `st_mode` is a nonzero value, a stereo mode is associated with the window and if it is equal to zero, stereo mode is turned off. This is device-specific and should be handled by this routine.

This routine should update the device-dependent information in the DGA shared page pertaining to stereo.

Information about a window's stereo state is stored in the device-specific portion of the `wx_dbuf` structure found in the `WXINFO` structure `infop->wx_dbuf.device`.

This array can be cast to a device-defined structure and the stereo information could be stored here.

Returns `1` on success; `0` on failure.

ChokeFb

```
int
(*ChokeFb)(ScreenPtr pScreenr, Bool flag)
```

Purpose When all windows on a screen are locked down, frame buffers having asynchronous accelerators need to choke the accelerator. This prevents the accelerator from rendering into a locked window. Since this is a device-specific operation, this function has to implement the choking and unchoking.

Arguments If the second argument, `flag`, is `1`, this function should choke the accelerator; if `flag` is `0`, it should unchoke the accelerator. Typically, this is done via an `ioctl`. For example, the GT uses the `FBIOGRABHW` `ioctl` to choke its accelerator.

Returns 1 on success; 0 on failure.

SyncDrawable

```
int
(*SyncDrawable)(DrawablePtr pDraw, GCPtr pGC)
```

Purpose When DGA is used to switch buffers, all X rendering functions need to be directed at the currently displayed buffer. This function is called before calling the X rendering function but only if the window is multibuffered.

This routine can also be used to update device-private structures with the current buffer state.

Results This function might need to call `dgaMbGetBufferInfo` to get the current buffer configuration.

UnsyncDrawable

```
int
(*UnsyncDrawable)(DrawablePtr pDraw, GCPtr pGC)
```

Purpose This function should undo anything that was done in `SyncDrawable`.

This routine can also be used to update device private structures with the current buffer state.

Results This function may need to call `dgaMbGetBufferInfo` to get the current buffer configuration.

CmapSetup

```
int
(*CmapSetup)(CmapPtr pCmap, Grabbedcmap cginfop)
```

Purpose This function is called when a colormap is being grabbed. The include file that provides definition of the `Grabbedcmap` structure is `dga/dgacmapstr.h`. This function is typically used by devices supporting multiple hardware colormaps or other specialized colormap hardware.

Arguments In this routine, the DDX handlers can set up `cginfo->devinfop` to point to a private data area. The maximum size of this private area is `DGA_CM_DEV_INFO_SZ`, defined in `dga/dgacmapstr.h`. This field is declared as an `u_char` array.

Each DDX handler can cast this to a device-private structure. Typically, this device-dependent structure contains information about the hardware colormap associated with the grabbed X colormap.

On the client side, the client program can gain access to this data by using the appropriate `libdga` function call, `dga_cm_get_devinfo`. See Chapter 9, “Multibuffering Extension to X Interface” for more information.

Devices that do not have specialized colormap hardware, like multiple hardware color look up tables, do not need to fill out this element in the function vector, `DgaDevFuncsDraw`.

Returns The return value is ignored.

Server-Supplied Multibuffering Routines

If your DDX handler defines a non-NULL MBX `TryMpg` function, you are required to use the following routines to inform DGA of multibuffer set attributes of a multibuffered window. If your DDX handler does not define `TryMpg`, you do not need to make these calls.

To use these routines, include the `dgambufstr.h` header file.

dgaMbCrtSetInfo

```
int
dgaMbCrtSetInfo (pWin, flipMode, accessMode, siteTypeConst
                 bufViewabilityMask)
WindowPtr      pWin;
int            flipMode;
int            accessMode;
Bool           siteTypeConst;
unsigned long  bufViewabilityMask
```

Purpose This function informs DGA of the attributes of the multibuffer set of a multibuffered window. Nonzero is returned if the information was successfully associated with the window, zero otherwise.

Called by The MBX `TryMpg` routine. If the device driver does not call this routine, the following defaults will apply:

`flipMode`	DGA_MBFLIP_COPY
`accessMode`	DGA_MBACCESS_MULTIADDR
`siteTypeConst`	FALSE
`bufViewableMask`	`0` (all nonviewable)

Arguments `flipMode` specifies the method used to display multibuffers. It may be one of:

DGA_MBFLIP_VIDEO — use this if multibuffers are displayed by copying their contents into a viewable drawable

DGA_MBFLIP_COPY — use this if they are displayed by directly outputting a video single from the multibuffer

`accessMode` specifies how a foundation library client can access the multibuffers. It may be one of:

DGA_MBACCESS_SINGLEADDR — specifies single address access mode. In this mode, clients use a single address and a render buffer state in the device to specify the rendering destination

DGA_MBACCESS_MULTIADDR — specifies multiple address mode. In this mode, clients use a unique address for each buffer to specify the rendering destination

`siteTypeConst` is `TRUE` if the sites of the multibuffers in the multibuffer set will never change during the lifetime of the set, and `FALSE` otherwise.

`bufViewableMask` is a bit mask in which the bits specify the viewability of all multibuffers in the multibuffer set. The viewability of multibuffer `i` is specified by (`1L<<i`) in the mask. `1` means the multibuffer is *viewable* (video can be sent directly out of it). `0` means the multibuffer is nonviewable (the multibuffer must be copied to a viewable drawable to be seen).

dgaMbSetBufViewability

```
int
dgaMbSetBufViewability (pWin, bufIndex, viewable)
WindowPtr  pWin;
short      bufIndex;
Bool       viewable;
```

Purpose This function is used to specify the viewability of an individual buffer.

Called by The MBX `ResizeMultibuffer` routine if resizing causes a change in the viewability of a multibuffer.

Arguments `bufIndex` is the index of the multibuffer in the multibuffer set (counted from 0).

If `viewable` is `TRUE`, the multibuffer is viewable, otherwise it is nonviewable.

Returns Nonzero if the information was successfully associated with the window; zero otherwise.

dgaMbSetDisplayBuf

```
void
dgaMbSetDisplayBuf (pWin, bufIndex)
WindowPtr   pWin;
short       bufIndex;
```

Purpose This function specifies the current display buffer of a multibuffered window. This routine must be called only after a creation sequence has been successfully completed on the window. The initial display buffer is `0`.

Called by `DisplayMultibuffer`, if the DDX handler defines a non-NULL `DisplayMultibuffer`.

dgaMbIsMultibuffer

```
Bool
dgaMbIsMultibuffer (pPix, ppWin)
PixmapPtr   pPix;
WindowPtr   *ppWin
```

Returns `TRUE` if the given pixmap is actually a pixmap that was created through the MBX extension. In other words, returns `TRUE` if it is a multibuffer. Otherwise returns `FALSE`. Regardless of the drawable type, the drawable must have been previously grabbed. Otherwise returns `FALSE`.

If `TRUE` is returned, a pointer to the main window of the multibuffer is also returned.

Note – The DGA cache notification routines (see “Caching Routines” on page 226) use this routine to distinguish multibuffers from pixmaps.

Note – This should be implemented by adding a field to the `DgaPixmapRec`. This field has three states: pixmap, multibuffer, or don’t know. If pixmap or multibuffer, return FALSE or TRUE respectively. If don’t know, do a

`LookupIdByType` on `pPix->drawable.id` with type `MultibufferResType`. If this succeeds, it's a multibuffer. If not, it's a pixmap. Record the result in the `DgaPixmapRec` and return it.

dgaMbGetBufferInfo

```
void
dgaMbGetBufferInfo (pDraw, num_buffers, read_buffer,
write_buffer, display_buffer)
DrawablePtr    pDraw;
short          *num_buffers;
short          *read_buffer;
short          *write_buffer;
short          *display_buffer;
```

Returns Information about the current buffer set.

Called by The DGA routines, `SyncDrawable` and `UnsyncDrawable`.

Caching Routines

The following routines allow a DDX handler to keep DGA informed of caching changes on a device.

dgaCacheDescribeDev

```
void
DgaCacheDescribeDev (pScreen, devCode, devName)
ScreenPtr  pScreen;
int        devCode;
char       *devName;
```

Results The contents of `devName` are copied into an internal structure.

dgaCacheStateChange

```
void
DgaCacheStateChange (pDraw, state)
DrawablePtr    pDraw;
Bool           state;
```

Purpose Informs DGA that a change has occurred to the cache state of a drawable. `DgaCacheDescribeDev` must have been called prior to calling this routine.

Arguments If `state` is `TRUE`, the drawable is currently cached. If it is `FALSE`, the drawable is not cached.

dgaSharedDataInfo

```
void
DgaSharedDataInfo (pDraw, addr, linebytes)
DrawablePtr    pDraw;
pointer        *addr;
int            *linebytes
```

Purpose When a nonviewable drawable or backing store is not cached, the data pointer of the drawable should be directed toward the pixel store that exists in the shared page and the contents of the drawable should be copied into the shared page. This is automatically performed by DGA if the DGA routines `CacheDrawInit` or `CacheDrawCleanup` return 0. However, the DDX handler itself may want to copy the drawable contents into the shared page (for performance). To do this, the DDX handler must know where to put the data. It must also know the scanline stride (*linebytes*). This routine supplies the necessary information necessary. This routine should only be called when the drawable has been grabbed.

11

Device Information Routines

In each shared information page of a drawable, DGA provides an area in which a DDX handler can place device-specific information. When anything in this area changes, the DDX handler must inform DGA so that it can signal the change to the client.

dgaDevInfoGet

```
pointer
DgaDevInfoGet (pDraw)
DrawablePtr    pDraw;
```

Purpose — The device-dependent area can be used by DDX handlers to transmit device-dependent information to the DDX handlers of the client foundation libraries. The format of this area is completely opaque to DGA; no interpretation is given.

Called by — This routine might need to be called from a DDX handler's DGA `GrabDrawable` routine to initialize device-dependent information for a drawable. It might also need to be called for a cached nonviewable drawable if the DDX handler changes the location of the cache.

Results — If the device alters any information in this area, it should call `DgaDevInfoChange` to inform DGA.

Returns — A pointer to the device-dependent area in the shared information of the given drawable. Returns NULL if the drawable has not yet been grabbed.

dgaDevInfoChange

```
void
DgaDevInfoChange (pDraw)
DrawablePtr    pDraw;
```

Purpose This routine informs DGA that a change has occurred to the device-dependent area of the drawable. A pointer to this area is returned by calling `DgaCacheDevInfo`. This routine must be called after any DDX handler changes to this area.

DGA and Colormaps

The colormap grabber is discussed in “Colormap Grabber Interface” on page 188. It allows DGA foundation libraries to directly load color lookup tables, bypassing the X protocol. This functionality is not required for Solaris to operate properly. The implementation of DGA libraries handles the case where colormap grabs fail and fall back to Xlib to load the lookup tables. The performance loss is minimal.

The implementation of the colormap grabber uses interfaces which are private to the CMAP pacakage and DGA. By default, the colormap grabber is disabled for each screen. It is enabled when the handler for a given screen calls `cmapScreenInit()` to initialize the CMAP package for that screen.

If the DDX handler implementor chooses to disable the colormap grabber on a device that is using the CMAP package, the handler should call the function `dgaDisableCmapGrabs(ScreenPtr)` after the call to `cmapScreenInit()`.

Note – Ideally, the DGA implementation should check the return value from the screen’s `CmapSetup` function to disable and enable grabs, but unfortunately, it does not. This cannot be changed without breaking binary compatibility.

11

Input Devices 12

This chapter describes how to add an extension input device to the OpenWindows server and access it with the MIT XInput Extension. This extension is an MIT standard that is distributed with X11 Release 5 (X11R5). The OpenWindows server loads input devices dynamically and accesses them through the Input Extension. Dynamic loading reduces the size of the core X server and allows you to develop device drivers independently.

Note – The client interface for accessing input devices in OpenWindows is the Input Extension as defined in X11R5. The design presented here does not change that interface in any way. All client protocol requests in this chapter are as defined in the Input Extension.

The Input Extension includes the following three documents that are prerequisite to this chapter. These documents are on line in the `doc/extensions/xinput` directory. The associated filename is in parentheses.

- *X11 Input Extension Protocol Specification*, Patrick and Sachs, MIT X Consortium. (`protocol`.ms)
- *X11 Input Extension Library Specification*, Patrick and Sachs, MIT X Consortium. (`lib`rary.ms)
- *X11 Input Extension Porting Document*, Sachs, MIT X Consortium. (`porting.ms`)

12

Extension Input Device Overview

Figure 12-1 on page 232 shows a block diagram of the device input portion of the OpenWindows server. The diagram also indicates which components must be developed by Independent Hardware Vendors (IHVs) and Independent Software Vendors (ISVs) to add an extension input device to OpenWindows.

The server implements most of the Input Extension capabilities: decoding protocol requests, managing input devices, and distributing events to interested clients. No changes to the server are required to add a new input device.

The *device handler* reads device events, converts device events to X events, and adds the events to the servers global event queue. Each new input device must have a device handler developed for it.

The device's STREAMS modules convert raw data from the physical input device into event packets that are read by the device handler. A STREAMS module is not required for each input device, but when needed it is developed by the IHV and ISV.

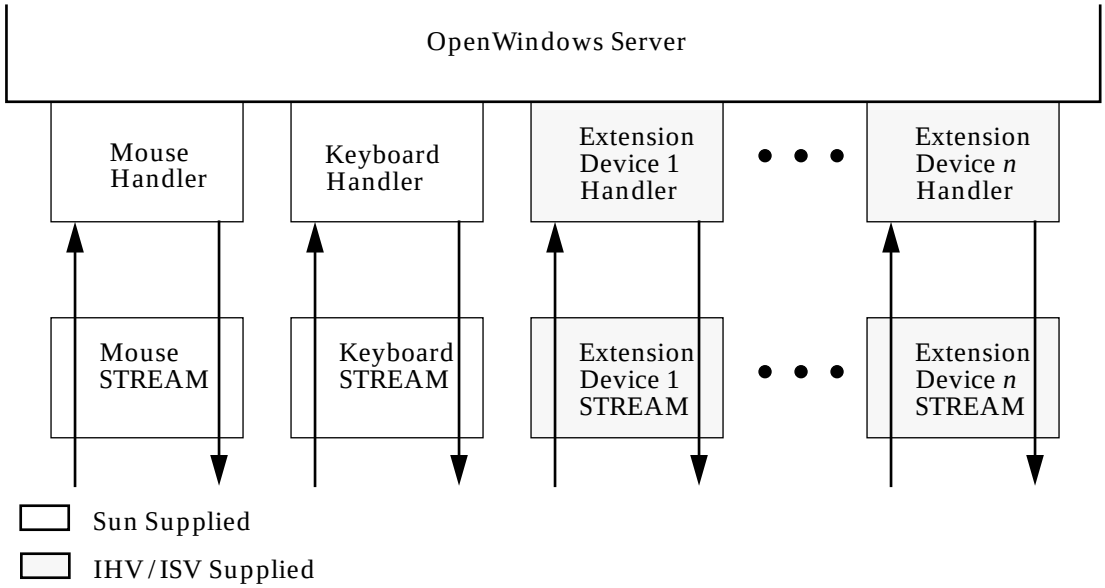


Figure 12-1 Extension Input Device Block Diagram

Handling of Extension Input Devices

This section provides a high level discussion of how extension input devices are implemented in the OpenWindows environment.

Extension Device Initialization

After server start-up, the core keyboard and core pointer are the only devices that are initialized and generating events. Additional devices can be requested by a client with the `XListInputDevices` request. Each time a client issues this request, the server executes the following tasks:

1. Reads the configuration file

   The server parses the `OWconfig` configuration file, searching for input devices. Each time the `OWconfig` file is read due to an `XListInputDevices` request, devices listed in the `XDISPLAY` class as `coreKeyboard` and `corePointer` *and* at server start-up were not the core pointer and keyboard, are treated as extension devices.

   For more information on the OWconfig file, see Appendix A, “The OWconfig File.”

2. Loads input device

   All devices in the `OWconfig` file that have not been initialized are loaded. Thus, for the first request after start-up the core keyboard and core pointer have already been initialized; only new devices are loaded.

   Later, upon receipt of another `XListInputDevices` request, the server again searches the `OWconfig` file for any devices that have been added since the last request. If it finds new devices, they are loaded.

3. Initializes the device

   After a device is loaded, its `DeviceControlProc` function is called with a value of `DEVICE_INIT`, causing the device to register all of its features with the server. `DeviceControlProc` is defined on page 260.

   The server can now return a reply to the `XListInputDevices` request issued by the client. The `XListInputDevices` request does not turn on the device so the server does not accept input from them yet.

If during initialization the `DeviceControlProc` routine returns a failure, the server assumes the hardware is not present and unloads the device.

Extension Device Open

After receiving the reply to the `XListInputDevices`, the client can open an extension device and start receiving input from it with the.`XOpenDevice` request. When the server receives the first `XOpenDevice` request for a particular device, it tells the device to start generating events by calling the `DeviceControlProc` function with a value of `DEVICE_ON`.

The server keeps a list of clients that currently have the device open. If the device is already opened by a client when an `XOpenDevice` request is received, the requesting client is added to the client list.

Server start-up is now complete. When input is pending on the device, the server reads the data and puts it into the event stream. The client can now issue any of the standard Input Extension protocol requests to receive events, initiate grabs, and control features of the device.

Reading Input Data

During initialization, devices register a read procedure with the server and set the device STREAM to generate SIGPOLLs when data reaches the STREAM head. The input data flow begins when a SIGPOLL signal is received by the server. The server then loops through the following steps as illustrated in Figure 12-2 on page 235, until no more events are available on any of the input devices:

1. For each device that is turned on, call the `DeviceReadProc` function for that device. `DeviceReadProc` is defined on page 262.
2. Check to see if there are any events from all of the sources just read.
   - If there are no more events, break out of the loop and return.
   - If there are more events, continue to step 3.
3. Find the oldest event.
4. Give the oldest event to the `DeviceEnqueueProc` for that device. `DeviceEnqueueProc` is defined on page 261.

The `DeviceEnqueueProc` procedure takes an event, processes any device-dependent information on the event, converts it to an `xEvent`, and places it on the global event queue via the `mieqEnqueue` procedure.

5. Loop back to Step 1.

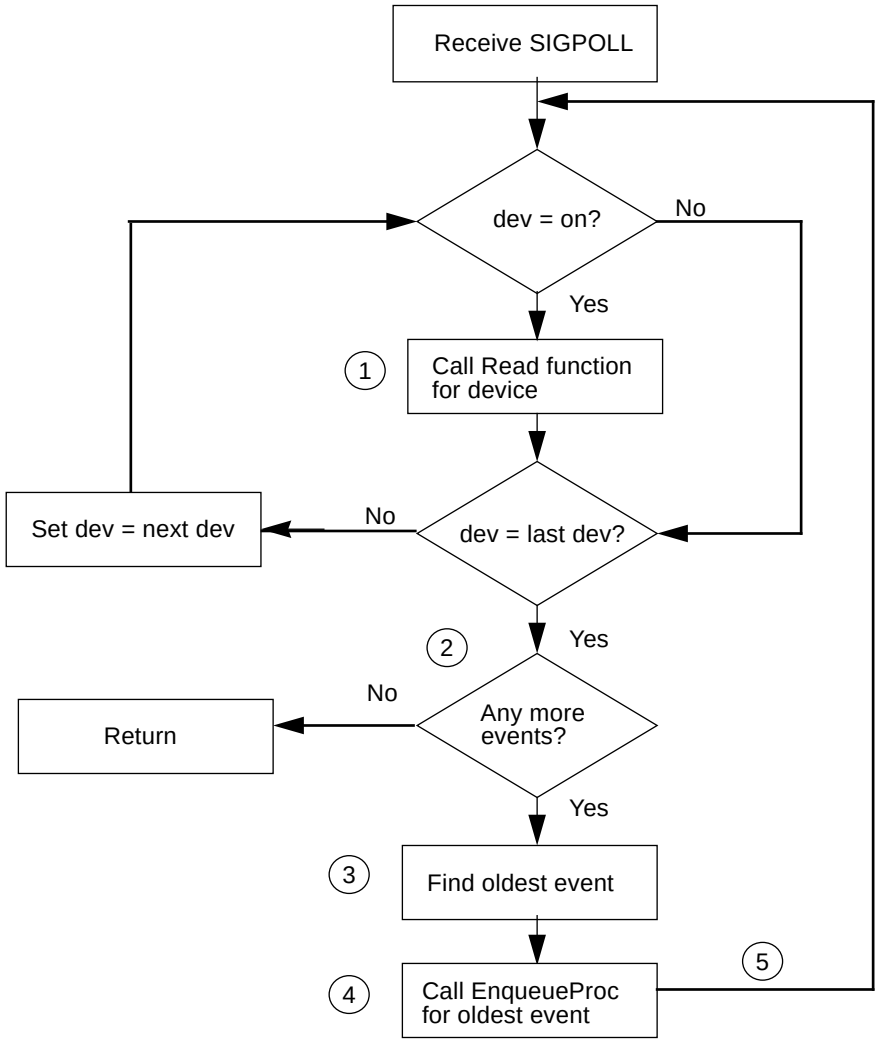


Figure 12-2 Data Flow When Reading Devices

Extension Device Close

When a client is finished with a device, it issues an `XCloseDevice` request to the server. The client that issued the `XCloseDevice` request does not receive any more events from the device. What happens next depends on how many clients have the device open:

- If other clients have the device open, the server continues to read the device until no clients have the device open. The client that issued the `XCloseDevice` request does not receive any more events from the device because the event mask for that client is cleared by the Input Extension as part of the `XCloseDevice` procedure.
- If the client is the only client with the device open, the server calls the `DeviceControlProc` with a value of `DEVICE_OFF` instructing the device to stop generating events.

Restart and Shutdown

Restarting and shutting down the server involve the same actions. All open devices are closed and unloaded. During the close process the input device is notified of the shutdown. The device must free any memory that has been allocated and close the device's file descriptor.

When the server is about to exit or restart, the server calls the `DeviceControlProc` function with a value of `DEVICE_CLOSE`. This call instructs the device to free all of its resources because the server is about to exit.

Adding An Extension Input Device

Each device added to the server must have the following components:

- A device handler shared object
- An entry in the local `OWconfig` file

And is recommended to have:

- A STREAMS module

Writing the Device Handler

All device handlers must have `DeviceControlProc`, `DeviceGetEvents`, and `DeviceEnqueueProc` procedures, as well as device-dependent procedures. This section describes each of these procedures. A sample tablet handler is provided in `server/ddx/solaris/reference/sunTablet` to aid in the understanding of this chapter.

Device Control Procedure

The `DeviceControlProc` function allows the server to control an extension device without having to know the capabilities of each particular device. There are four actions that the `DeviceControlProc` must handle:

- `DEVICE_INIT`
- `DEVICE_ON`
- `DEVICE_OFF`
- `DEVICE_CLOSE`

DEVICE_INIT

When the `DeviceControlProc` is called with action `DEVICE_INIT`, the procedure completes the following tasks:

1. The device is opened and initialized.
2. Any private device structures are allocated and initialized.
3. An atom for the device must be generated and assigned to the device. The device's state is initialized to `off` by setting the `device->on` flag to `FALSE`.
4. The device registers its `DeviceGetEvents` and `DeviceEnqueueProc` by calling `RegisterFdIo`.
5. All device-dependent structures must be initialized and device-dependent procedures registered. If the device can become the core pointer or the core keyboard, pointer or keyboard interest must be registered. The initialization and registry functions are listed in "Public Server Functions" on page 241.

DEVICE_ON

When the `DeviceControlProc` is called with action `DEVICE_ON`, the procedure completes the following tasks:

1. Call AddEnabledDevice to let the server know the device has been turned on.
2. Set the devices on state to TRUE.
3. Cause the device to generate SIGPOLLs with the I_SETSIG ioctl.

DEVICE_OFF

When the DeviceControlProc is called with action DEVICE_OFF, the procedure completes the following tasks:

1. Call RemoveEnabledDevice to let the server know the device has been turned off.
2. Set the device's on state to FALSE.

DEVICE_CLOSE

When the DeviceControlProc is called with action DEVICE_CLOSE, the procedure completes the following tasks:

1. If the device's on state is TRUE, call RemoveEnabledDevice and set on state to FALSE.
2. Perform any device specific clean-up.
3. Close the device.
4. Free any private device structures.

Device Get Events Procedure

The DeviceGetEvents procedure must read the device, put the events into an XI_event structure, and return a pointer to the event or events. If the DeviceGetEvents procedure allocates memory for the XI_event structure it must be freed in the DeviceEnqueueProc. The example tablet device handler keeps a static array of XI_event structures and passes a pointer to this array each time.

Device Enqueue Procedure

The `DeviceEnqueueProc` is required to be in all device handlers. The `DeviceEnqueueProc` takes one `XI_eventPtr` and enqueues one or more events on the global event queue. The `DeviceEnqueueProc` is passed a `XI_event` structure which has an opaque pointer to the event. The `DeviceEnqueueProc` must typecast this pointer to match the format that the `DeviceGetEvents` procedure put into the structure. The server does not do any processing on the event before it is passed to the `DeviceEnqueueProc`. As noted above, if the `DeviceGetEvents` procedure allocates memory for the `XI_event` structure it must be freed here.

As stated in *X11 Input Extension Protocol Specification*, `DeviceKeyPress`, `DeviceKeyRelease`, `DeviceButtonPress`, `DeviceButtonRelease`, `ProximityIn`, `ProximityOut`, and `DeviceStateNotify` events can be followed by zero or more `DeviceValuator` events. Devices that have valuators and are reporting absolute motion must follow each of the above events with one or more `DeviceValuator` events to specify the current state of the valuators. Devices that don't have valuators or have valuators but are reporting relative motion send zero `DeviceValuator` events following the events listed above. A `DeviceMotionNotify` event is always followed by one or more `DeviceValuator` events regardless of the mode of the device (relative or absolute). See the Input Extension Protocol Specification for more details.

Devices that have registered themselves as potential core pointer devices must be able to control the cursor from this procedure. The device must not control the cursor until after the server has notified the device that it is the core pointer. Cursor control is accomplished calling either `miPointerDeltaCursor` or `miPointerAbsoluteCursor` depending on whether the device is reporting relative or absolute motion. The device must not enqueue MotionNotify events when it is the core pointer; this is done by the miPointer procedures. It is the responsibility of the device handler to enqueue ButtonPress and ButtonRelease events if the device supports buttons.

Devices that have registered themselves as potential core keyboards enqueue `DeviceKeyPress` and `DeviceKeyRelease` events unless the device handler has been notified that it is the core keyboard. Once it becomes the core keyboard it must enqueue KeyPress and KeyRelease events until such time the device is notified it is no longer the core keyboard.

Device-Dependent Procedures

Devices also have to support additional procedures based on the types of input classes a given device supports, such as KEY, BUTTON, and VALUATOR. These procedures are explained in “Device Shared Library Functions” on page 260.

Adding An OWconfig File Entry

Appendix A, “The OWconfig File” describes the `OWconfig` file and the name value attribute pairs that describe each input device. Appendix B, “Packaging and Installation Hints” discusses how a new input device is packaged for installation by users. “DDX Versioning” on page 9 specifies shared object naming and versioning conventions. Read these sections before attempting to add an input device.

Debugging the Device Handler

Since the input device handlers are shared objects, breakpoints cannot be set in the handler until after the server has loaded the shared object. All extension input device handlers are loaded when the first client issues an `XListInutDevices.`

Breakpoints can be set in an input device handler by following these steps:

1. Add a line to the `OWconfig` file for the input device to be debugged. Make sure the new line is directly below the mouse and keyboard lines.
2. From a remote machine, debug the server (`dbx Xsun` or `debugger Xsun`).
3. Set a breakpoint in `AddInputDevice.`
4. Run the server. The `AddInputDevice` breakpoint hits twice during server initialization; just continue each time.
5. Start a client that opens the extension input device. This causes the breakpoint to hit again. At this point the input device handler is loaded and you can set breakpoints inside the handler.

Writing The STREAMS Module

A STREAMS module is not required for every input device. For example, the device handler could read, interpret, and format the raw data from the `ttya` port. This design is least attractive from a performance perspective and it is strongly recommended that the interpreting and formatting of data be handled in a STREAMS module. This method is attractive if you have a limited amount of time to get an input device working, are unfamiliar with STREAMS module development, and are not concerned about performance.

A STREAMS module outputs vuid (virtual user input device) type events. See Appendix C, "Virtual User Input Device Interface" for more information on vuid events.

Note – The `DeviceReadProc` function returns the `XI_eventPtr` structure that is a timestamp and an opaque pointer to the devices event. This timestamp could be generated in `DeviceReadProc`. However it is strongly recommended that the device's STREAMS module timestamp the event and `DeviceReadProc` use this timestamp for the `XI_eventPtr`.

Input Library Functions

This section describes new functions in two categories:

- Public server functions
- Device-shared library functions

Public Server Functions

The functions in this section are callable from the device shared library.

InitPointerDeviceStruct

```
Bool InitPointerDeviceStruct(DevicePtr device, CARD8 *map,
       int numButtons, DeviceGetMotionProc GetMotionProc,
       DevicePtrCtrlProc PtrCtrlProc, int numMotionEvents)
```

Purpose This function is provided to allocate and initialize `ButtonClassRec`, `ValuatorClassRec`, and `PtrFeedbackClassRec`.

Used by the initial core pointer device. A call to `InitPointerDeviceStruct` is equivalent to `calling InitButtonClassDeviceStruct` (page 244), `InitValuatorClassDeviceStruct` (page 244), and `InitPtrFeedbackClassDeviceStruct` page 247).

Called by `DeviceControlProc` of the core pointer device during the `DEVICE_INIT` action.

Results Allocates and initializes `ButtonClassRec`, `ValuatorClassRec`, and `PtrFeedbackClassRec`.

Returns `TRUE` on success
`FALSE` on failure

InitKeyboardDeviceStruct

```
Bool InitKeyboardDeviceStruct(DevicePtr device,
       KeySymsPtr pKeySyms, CARD8 pModifiers[],
       DeviceBellProc BellProc, DeviceKbdCtrlProc KbdCtrlProc)
```

Purpose This function is provided to allocate and initialize `KeyClassRec`, `FocusClassRec`, and `KbdFeedbackClassRec.`

Used by the initial core keyboard device. A call to `InitKeyboardDeviceStruct` is equivalent to calling `InitKeyClassDeviceStruct` (page 243), `InitFocusClassDeviceStruct` (page 246), and `InitKbdFeedbackClassDeviceStruct` (page 247).

Called by DeviceControlProc of the core keyboard device during the DEVICE_INIT action.

Results Allocates and initializes KeyClassRec, FocusClassRec, and KbdFeedbackClassRec.

Returns TRUE on success
FALSE on failure

InitKeyClassDeviceStruct

```
Bool InitKeyClassDeviceStruct(DeviceIntPtr dev,
       KeySymsPtr pKeySyms, CARD8 pModifiers[])
```

Purpose This function is provided to allocate and initialize a KeyClassRec, and is called for extension devices that have keys. It is passed a pointer to the device, and pointers to arrays of keysyms and modifiers reported by the device.

InitKeyboardDeviceStruct calls this routine for the core X keyboard. It must be called explicitly for extension devices that have keys.

Called by DeviceControlProc during the DEVICE_INIT action.

Results Allocates and initializes KeyClassRec.

Returns TRUE on success
FALSE on failure

InitButtonClassDeviceStruct

```
Bool InitButtonClassDeviceStruct(DeviceIntPtr dev,
       int numButtons, CARD8 *map)
```

Purpose This function is provided to allocate and initialize a `ButtonClassRec`, and is called for extension devices that have buttons. It is passed a pointer to the device, the number of buttons supported, and a map of the reported button codes.

`InitPointerDeviceStruct` calls this routine for the core X pointer. It must be called explicitly for extension devices that have buttons.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results Allocates and initializes `ButtonClassRec`.

Returns `TRUE` on success
`FALSE` on failure

InitValuatorClassDeviceStruct

```
Bool InitValuatorClassDeviceStruct(DeviceIntPtr dev,
       int numAxes, DeviceGetMotionProc GetMotionProc,
       int numMotionEvents, int mode)
```

Purpose This function is provided to allocate and initialize a `ValuatorClassRec`, and is called for extension devices that have valuators. It is passed the number of axes of motion reported by the device, the address of the motion history procedure for the device, the size of the motion history buffer, and the mode (`Absolute` or `Relative`) of the device.

`InitPointerDeviceStruct` calls this routine for the core X pointer. It must be called explicitly for extension devices that report motion.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results Allocates and initializes `ValuatorClassRec`.

Returns `TRUE` on success
`FALSE` on failure

InitValuatorAxisStruct

```
Bool InitValuatorAxisStruct(DeviceIntPtr dev, int axnum,
       int minval, int maxval, int resolution)
```

Purpose This function is provided to initialize an `XAxisInfoRec`, and is called for core and extension devices that have valuators. The space for the `XAxisInfoRec` is allocated by the `InitValuatorClassDeviceStruct` function, but is not initialized.

`InitValuatorAxisStruct` is called once for each axis of motion reported by the device. Each invocation is passed the axis number (starting with 0), the minimum value for the axis, the maximum value for that axis, and the resolution of the device in counts per meter. If the device reports relative motion, 0 is reported as the minimum and maximum values.

This routine is not called by `InitPointerDeviceStruct` for the core X pointer. It must be explicitly called for core and extension devices that report motion.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results Initializes `XAxisInfoRec`.

Returns `TRUE` on success
`FALSE` on failure

InitFocusClassDeviceStruct

```
Bool InitFocusClassDeviceStruct(DeviceIntPtr dev)
```

Purpose This function is provided to allocate and initialize a `FocusClassRec`, and is called for extension devices that can be focused. It is passed a pointer to the device.

`InitKeyboardDeviceStruct` calls this routine for the core X keyboard. It must be called explicitly for extension devices that can be focused. Whether or not a particular device can be focused is implementation-dependent.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results Allocates and initializes `FocusClassRec.`

Returns `TRUE` on success
`FALSE` on failure

InitProximityClassDeviceStruct

```
Bool InitProximityClassDeviceStruct(DeviceIntPtr dev)
```

Purpose This function is provided to allocate and initialize a `ProximityClassRec`, and is called for extension absolute pointing devices that report proximity. It is passed a pointer to the device.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results Allocates and initializes a `ProximityClassRec.`

Returns `TRUE` on success
`FALSE` on failure

InitKbdFeedbackClassDeviceStruct

```
Bool InitKbdFeedbackClassDeviceStruct(DeviceIntPtr dev,
       DeviceBellProc BellProc, DeviceKbdCtrlProc KbdCtrlProc)
```

Purpose This function is provided to allocate and initialize a `KbdFeedbackClassRec`, and is called for extension devices that support some or all of the feedbacks that the core keyboard supports. It is passed a pointer to the device, a pointer to the procedure that sounds the bell, and a pointer to the device control procedure.

`InitKeyboardDeviceStruct` calls this routine for the core X keyboard. It must be called explicitly for extension devices that have the same feedbacks as a keyboard. Some feedbacks, such as LEDs and bell, can be supported either with a `KbdFeedbackClass` or with `BellFeedbackClass` or `LedFeedbackClass` feedbacks.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results Allocates and initializes `KbdFeedbackClassRec`.

Returns `TRUE` on success
`FALSE` on failure

InitPtrFeedbackClassDeviceStruct

```
Bool InitPtrFeedbackClassDeviceStruct(DeviceIntPtr dev,
       DevicePtrCtrlProc PtrCtrlProc)
```

Purpose This function is provided to allocate and initialize a `PtrFeedbackClassRec`, and is called for extension devices that allow the setting of acceleration and threshold. It is passed a pointer to the device, and a pointer to the device control procedure.

`InitPointerDeviceStruct`() calls this routine for the core X pointer. It must be called explicitly for the extension devices that support the setting of acceleration and threshold.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results Allocates and initializes `PtrFeedbackClassRec`.

Returns `TRUE` on success
`FALSE` on failure

InitLedFeedbackClassDeviceStruct

```
Bool InitLedFeedbackClassDeviceStruct(DeviceIntPtr dev,
       DeviceLedCtrlProc LedCtrlProc)
```

Purpose This function is provided to allocate and initialize a `LedFeedbackClassRec`, and is called for extension devices that have LEDs. It is passed a pointer to the device, and a pointer to the device control procedure.

Up to 32 LEDs per feedback can be supported, and a device can have multiple feedbacks of the same type.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results Allocates and initializes `LedFeedbackClassRec`.

Returns `TRUE` on success
`FALSE` on failure

InitBellFeedbackClassDeviceStruct

```
Bool InitBellFeedbackClassDeviceStruct(DeviceIntPtr dev,
       DeviceBellCtrlProc BellCtrlProc)
```

Purpose This function is provided to allocate and initialize a `BellFeedbackClassRec`, and is called for extension devices that have a bell. It is passed a pointer to the device, and a pointer to the device control procedure.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results Allocates and initializes `BellFeedbackClassRec`.

Returns `TRUE` on success
`FALSE` on failure

InitStringFeedbackClassDeviceStruct

```
Bool InitStringFeedbackClassDeviceStruct(DeviceIntPtr dev,
       DeviceStringCtrlProc StringCtrlProc, int max_symbols,
       int num_symbols_supported, KeySym *symbols)
```

Purpose This function is provided to allocate and initialize a `StringFeedbackClassRec`, and is called for extension devices that have a display upon which a string can be displayed. It is passed a pointer to the device and a pointer to the device control procedure.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results Allocates and initializes `StringFeedbackClassRec`.

Returns `TRUE` on success
`FALSE` on failure

InitIntegerFeedbackClassDeviceStruct

```
Bool InitIntegerFeedbackClassDeviceStruct(DeviceIntPtr dev,
       DeviceIntegerCtrlProc IntegerCtrlProc)
```

Purpose This function is provided to allocate and initialize an `IntegerFeedbackClassRec`, and is called for extension devices that have a display upon which an integer can be displayed. It is passed a pointer to the device and a pointer to the device control procedure.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results Allocates and initializes `IntegerFeedbackClassRec`.

Returns `TRUE` on success
`FALSE` on failure

RegisterFdIo

```
int RegisterFdIo(DevicePtr devptr, int fd,
       DeviceReadProc readProc, DeviceEnqueueProc enqueueProc)
```

Purpose This function is provided to register the device's file descriptor, read function, and enqueue function.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results Registers the device's file descriptor, `read` function, and `enqueue` function with the server. The device's `read` function is called when there is input pending on the given file descriptor.

Returns `Success` on success
`!Success` on failure

RegisterModifierCheckProc

```
int RegisterModifierCheckProc(DevicePtr devptr,
       DeviceModifierCheckProc ModifierCheckProc)
```

Purpose This function is provided to register a function to be called when a keycode needs to be checked for validity by the device. This is only valid for devices that support keys. See “DeviceModifierCheckProc” on page 262.

Called by `DeviceControlProc` during the DEVICE_INIT action.

Results If the device supports keys and this function is not specified, the server assumes that the keycode is valid. If the function is specified, the server calls the function to check validity.

Returns `Success` on success
`!Success` on failure

RegisterSetDeviceModeProc

```
int RegisterSetDeviceModeProc(DevicePtr devptr,
       DeviceSetModeProc SetDeviceModeProc)
```

Purpose This function is provided to register a function to be called when a client requests a change in the mode of a device. This refers to the device reporting absolute or relative positions. See “DeviceSetModeProc” on page 263.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results If this function is not specified, the server assumes that the mode of this device cannot be changed. If the function is present, the server calls it to notify the device that the client requests a mode change.

Returns `Success` on success
`!Success` on failure

RegisterSetDeviceValuatorsProc

```
int RegisterSetDeviceValuatorsProc(DevicePtr devptr,
       DeviceSetDeviceValuatorsProc SetDeviceValuatorsProc)
```

Purpose This function is provided to register a function to be called when a client requests a change in the valuators of a device. See “DeviceSetDeviceValuatorsProc” on page 263.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results If this function is not specified, the server assumes that the valuators of this device cannot be changed. If the function is present, the server calls it to notify the device that the client requests a change to the valuators.

Returns `Success` on success
`!Success` on failure

RegisterChangeDeviceControlProc

```
int RegisterChangeDeviceControlProc(DevicePtr devptr,
       DeviceChangeDeviceControlProc ChangeDeviceControlProc)
```

Purpose This function is provided to register a function to be called when a client requests a change in the control of a device. This can refer to any control on the device, but is currently limited to just the resolution of the device. See “DeviceChangeDeviceControlProc” on page 264.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results If this function is not specified, the server assumes that the control of this device cannot be changed. If the function is present, the server calls it to notify the device that the client wishes to change the control.

Returns `Success` on success
`!Success` on failure

RegisterXKeyboardInterest

```
int RegisterXKeyboardInterest(DevicePtr devptr, Bool focusable,
       DeviceChangeCoreKeyboardProc ChangeCoreKeyboardProc)
```

Purpose This function is provided to register interest with the server to indicate that the given device can become the core keyboard if a client so requests. The `focusable` argument specifies whether the device is focusable when it is not the core keyboard. See “DeviceChangeCoreKeyboardProc” on page 264.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results The device is registered as a possible core keyboard with the focusability that is specified. If the device is not registered as a possible core keyboard, the server assumes that the device cannot become the core keyboard.

Returns `Success` on success
`!Success` on failure

RegisterXPointerInterest

```
int RegisterXPointerInterest(DevicePtr devptr, Bool focusable,
       DevicePointerAxisChangeProc PointerAxisChangeProc)
```

Purpose This function is provided to register interest with the server to indicate that the given device can become the core pointer if a client so requests. The `focusable` argument specifies whether the device is focusable when it is not the core pointer.

The `DevicePointerAxisChangeProc` is called when the client requests this device to become the core pointer. See “DevicePointerAxisChangeProc” on page 265.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results The device is registered as a possible core pointer with the focusability that is specified. If the device is not registered as a possible core pointer, the server assumes that the device cannot become the core pointer.

Returns `Success` on success
`!Success` on failure

mieqUpdateKbdPtr

```
void mieqUpdateKbdPtr(DevicePtr pKbd, DevicePtr pPtr)
```

Purpose This function is provided to update the core keyboard or pointer device.

Called by The device shared libraries calls `mieqUpdateKbdPtr` from the `DeviceChangeCoreKeyboardProc` or `DevicePointerAxisChangeProc` each time the core keyboard or pointer device changes. `mieqUpdateKbdPtr` is called by the device that is becoming the core keyboard or pointer with it's `DevicePtr` in the appropriate argument. Set the other argument to NULL.

Results The `mi` event code treats the new device as the core keyboard or pointer.The old keyboard or pointer are treated as extension devices by the `mi` event code.

Returns None

mieqEnqueue

```
void mieqEnqueue(xEvent *e)
```

Purpose This function is provided to place the `xEvent` on the server's global event queue.

Called by Many different locations in the server, but for the current design this routine is being called only from the `DeviceEnqueueProc` in the device shared library.

Results The `xEvent` is placed on the global event queue.The event is copied from the caller, so the memory can be reused by the `DeviceEnqueueProc`.

Returns None

miPointerPosition

```
void miPointerPosition(int *x, int *y)
```

Purpose This function is provided to obtain the current location of the cursor. It is passed two pointers that are filled in with the current location of the cursor.

Called by The device shared libraries when they need to know the current location of the cursor.

Results The `*x` and `*y` pointers are set to the current `x` and `y` position of the cursor.

Returns None

miPointerDeltaCursor

```
void miPointerDeltaCursor(int dx, int dy, unsigned long time)
```

Purpose This function is provided to move the cursor as a result of device events. It is passed the delta x and y that the cursor is to move relative to its current position as well as the time of the motion event.

Called by `DeviceEnqueueProc` of the current core pointer in the device shared library.

Results The cursor is moved `dx,dy` from its previous position.

Returns None

miPointerAbsoluteCursor

```
void miPointerAbsoluteCursor(int x, int y, unsigned long time)
```

Purpose This function is provided to move the cursor as a result of device events. It is passed an absolute x and y position to which the cursor moves, as well as the time of the motion event.

Called by `DeviceEnqueuProc` of the current core pointer in the device shared library.

Results The cursor is moved to `x,y`.

Returns None

RegisterHandlers

```
int RegisterHandlers(DeviceWakeupHandler wakeupHandler,
       DeviceBlockHandler blockHandler, int *index)
```

Purpose This function is provided to register wakeup handlers or block handlers or both for the device. The server calls `wakeupHandler` immediately after it comes out of its `select` call due to client input or input device activity. The server calls `blockHandler` right before going into the `select` call. Some devices such as keyboards might need this functionality to implement features such as auto repeat. It is passed the address of the devices wakeup handler or block handler or both and a pointer to the index of the handler that the device uses to refer to the handler. A `NULL` can be passed for either handler indicating not to register it.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results A wakeup handler or block handler or both are registered with the server.

Returns `Success` on success
`!Success` on failure

RemoveHandlers

```
void RemoveHandlers(int index)
```

Purpose This function is provided to remove the device's block handler or wakeup handler or both. It is passed the index to the handlers that was returned in the `RegisterHandlers` call.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results The device's block handler or wakeup handler or both are removed.

Returns None

NextWakeupHandler

```
void NextWakeupHandler(int index, int nscreen, pointer pbdata,
       unsigned long err, pointer pReadmask)
```

Purpose This function is provided to call the next `wakeupHandler` registered. It must be called by a device's `wakeupHandler` and passes along all the parameters that are passed into the device's `DeviceWakeupHandlerProc`.

Called by The device's `DeviceWakeupHandlerProc.`

Results The wakeup handler that was registered just before the device's `DeviceWakeupHandlerProc` is called.

Returns None

NextBlockHandler

```
void NextBlockHandler(int index, int nscreen, pointer pbdata,
       struct timeval **pptv, pointer pReadmask)
```

Purpose This function is provided to call the next `blockHandler` registered. Is must be called by a device's block handler and passes all the parameters that are passed into the device's `DeviceBlockHandlerProc`.

Called by The device's `DeviceBlockHandlerProc.`

Results The block handler that was registered just before the device's `DeviceBlockHandlerProc` is called.

Returns None

MakeAtom

```
Atom MakeAtom(char *name, unsigned len, Bool makeit)
```

Purpose This function is provided to make an atom for a device to be passed as a parameter to `AssignTypeAndName`. It is passed a `char` pointer to the name of the device, the length of the string, and `makeit` equals FALSE.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results An atom is found.

Returns Atom

AssignTypeAndName

```
void AssignTypeAndName(DeviceIntPtr dev, Atom type, char *name)
```

Purpose This function is provided to assign a type and name to a device. It is passed a pointer to the device, the atom returned from `MakeAtom`, and the `char` pointer to the name of the device.

Called by `DeviceControlProc` during the `DEVICE_INIT` action.

Results The `dev->type` and `dev->name` entries are set to the values specified by the arguments.

Returns None

AddEnableDevice

```
void AddEnabledDevice(int fd)
```

Purpose This function is provided to cause the server to start checking for input on the device corresponding to the given file descriptor.

Called by `DeviceControlProc` during the `DEVICE_ON` action.

Results The device's file descriptor is selected for pending input.

Returns None

RemoveEnableDevice

```
void RemoveEnabledDevice(int fd)
```

Purpose This function is provided to cause OpenWindows to stop checking for input on the device corresponding to the given file descriptor.

Called by DeviceControlProc during the DEVICE_OFF action.

Results The device's file descriptor is no longer selected for pending input.

Returns None

Device Shared Library Functions

The functions in this section are in the device shared libraries. The *DeviceHandlerCompatible, *DeviceControlProc, *DeviceEnqueueProc, and *DeviceReadProc functions are required for each device library. All other functions are optional and depend on the features a particular device supports.

DeviceHandlerCompatible

```
typedef int (*DeviceHandlerCompatible)(int major, int minor,
       int *myMajor, int *myMinor, int (**pContorlProc)());
```

Purpose This function checks for compatibility and returns the device's major and minor numbers as well as a pointer to DeviceControlProc.

Results Compares the device's version number against the version number passed in. If it is incompatible, return !Success; otherwise, fill in the device major and minor number and a pointer to DeviceControlProc.

Returns Success on success
!Success on failure

DeviceControlProc

```
typedef int (*DeviceControlProc)(DevicePtr devptr, int action);
```

Purpose This function allows the server to control the actions of a device.

Results Results depend upon the given action:

`DEVICE_INIT`. The device registers all of its features with the server, opens the device, registers how to read it, and initializes itself.

`DEVICE_ON`. The device turns itself on by calling `AddEnabledDevice`.

`DEVICE_OFF`. The device turns itself off by calling `RemoveEnabledDevice`.

`DEVICE_CLOSE`. The device cleans up its resources and closes itself. The server is about to exit.

Returns `Success` on success
`!Success` on failure

DeviceEnqueueProc

```
typedef void (*DeviceEnqueueProc)(DevicePtr devptr,
       XI_eventPtr Xev);
```

Purpose This function places one or more new `xEvents` on the global event queue.

Results Completes any device specific processing on a given event, converts the event into an `xEvent`, and then places the event on the global event queue by calling `mieqEnqueue.`

Note – The memory associated with the `XI_event` can be freed after `mieqEnqueue` has been called to queue the new `xEvents`.

Returns None

DeviceReadProc

```
typedef XI_eventPtr (*DeviceReadProc)(DevicePtr devptr,
       int * numev, Bool * again);
```

Purpose This function reads data from a device when there is input pending, and returns a pointer to a list of `XI_events`. This routine is only used for devices that can read themselves.

Results If there is no data to be read, this function returns `NULL`, sets `numev` to `0`, and sets `again` to `FALSE`.

If there is data to be read, this function returns a pointer to a list of `XI_events` and sets `numev` to the number of `XI_events` returned. The server uses `again` to determine if the device has more data to be read. If `again` is set to `TRUE`, the server calls this function again without reentering `select`. If `again` is set to `FALSE`, the function is not called again without reentering `select`.

Note – The server passes the list of events back to the device's `enqueue` function one at a time, so the memory for the `XI_events` is released after the device has called `mieqEnqueue` in the `DeviceEnqueueProc`.

Returns A pointer to a list of `XI_events` or `NULL`.
`numev` indicating the number of events returned.
`again` indicating the possibility of this device having more data to be read.

DeviceModifierCheckProc

```
typedef Bool (*DeviceModifierCheckProc)(DevicePtr devptr,
       KeyCode keycode);
```

Purpose This function checks the validity of the given `keycode`. Checking occurs when a client is trying to set the modifier map of a device. This function is only valid for devices that support keys.

Results None

Returns TRUE if the keycode is valid
FALSE if the keycode is not valid

DeviceSetModeProc

```
typedef int (*DeviceSetModeProc)(DevicePtr devptr, int mode);
```

Purpose This function sets the mode of a device. The mode can be either Absolute or Relative. This routine applies only to devices that generate DeviceMotionNotify events.

Results On success, the mode of the device is set to mode.
On failure, the mode is unchanged.

Returns Success on success
!Success on failure

DeviceSetDeviceValuatorsProc

```
typedef int (*DeviceSetDeviceValuatorsProc)(DevicePtr devptr,
       int *valuators, int first_valuator, int num_valuators);
```

Purpose This function sets the valuators of a device to the values in valuators starting with valuator first_valuator and continuing through num_valuators.

Results On success, the value of the specified valuators are changed to valuators.

On failure, the value of the valuators is unchanged.

Returns Success on success
!Success on failure

DeviceChangeDeviceControlProc

```
typedef int (*DeviceChangeDeviceControlProc)(DevicePtr devptr,
        xDeviceCtl *control);
```

Purpose This function changes the specified device controls on the given input device. Currently, only the `DEVICE_RESOLUTION` control is supported.

Results On success, the specified control is changed.
On failure, the control is unchanged.

Returns `Success` on success
`!Success` on failure

DeviceChangeCoreKeyboardProc

```
typedef int (*DeviceChangeCoreKeyboardProc)(DevicePtr devptr,
        Bool isCore);
```

Purpose This function notifies the device that a client has requested that the device is now the core keyboard (`isCore == TRUE`) or that it is now *not* the core keyboard (`isCore == FALSE`). The `DeviceChangeCoreKeyboardProc` function must call `mieqUpdateKbdPtr` to notify the server that the core keyboard has been changed.

Results On success, the specified control is changed.
On failure, the control is unchanged.

Returns `Success` on success
`!Success` on failure

DevicePointerAxisChangeProc

```
typedef int (*DevicePointerAxisChangeProc)(DevicePtr devptr,
       Bool isCore, unsigned char x, unsigned char y);
```

Purpose This function notifies the device that a client has requested that the device is now the core pointer (`isCore == TRUE`) or that it is now *not* the core pointer (`isCore == FALSE`).

If (`isCore == TRUE`), axis number `x` moves the pointer in the X direction and axis number `y` moves the pointer in the Y direction.

`DevicePointerAxisChangeProc` must call `mieqUpdateKbdPtr`() to notify the server that the core keyboard has been changed.

Results On success, the given device becomes the new core pointer, and the old core device becomes an extension device that has its focusability set by its `focusable` flag.

On failure, the core pointer is unchanged.

Returns `Success` on success
`!Success` on failure

DeviceGetMotionProc

```
typedef int (*DeviceGetMotionProc)(DeviceIntPtr devptr,
       INT32 *coords, unsigned long start, unsigned long stop,
       ScreenPtr pScreen);
```

Purpose This function returns any events in the device's motion history buffer that occurred between the `start` and `stop` times.

Called by `ProcGetMotionEvents` in `dix/devices.c`.

Results Copies any events in the device's motion history buffer that occurred between the `start` and `stop` times to coordinates.

Returns Number of events copied to coordinates.

DeviceBellProc

```
typedef void (*DeviceBellProc)(int newpercent,
       DeviceIntPtr devptr);
```

Purpose This function rings the device's bell to the specified percent of maximum.

Results The device's bell is rung.

Returns None

DeviceWakeupHandlerProc

```
typedef void (*DeviceWakeupHandlerProc)(int nscreen,
       pointer pbdata, unsigned long err, pointer pReadMask);
```

Purpose Determined by the device handler implementation.

Results Depends on the device handler implementation.

Returns None

DeviceBlockHandlerProc

```
typedef void (*DeviceBlockHandlerProc)(int nscreen,
       pointer pbdata, struct timeval **pptv,
       pointer pReadmask);
```

Purpose Determined by the device handler implementation.

Results Depends on the device handler implementation.

Returns None

DevicePtrCtrlProc

```
typedef void (*DevicePtrCtrlProc) (DeviceIntPtr devintptr,
       PtrCtrl *ctrl);
```

Purpose This function allows the server to control the actions of a pointer device.

Results Sets the value in the device's `PtrCtrl` structure.

Returns None

DeviceKbdCtrlProc

```
typedef void (*DeviceKbdCtrlProc) (DeviceIntPtr devintptr,
       KeybdCtrl *ctrl);
```

Purpose This function allows the server to control the actions of a keyboard device.

Results Sets the value in the device's `KeybdCtrl` structure.

Returns None

DeviceLedCtrlProc

```
typedef void (*DeviceLedCtrlProc) (DeviceIntPtr devintptr,
       LedCtrl *ctrl);
```

Purpose This function allows the server to control the actions of a device with LEDs.

Results Sets the value in the device's `LedCtrl` structure.

Returns None

DeviceBellCtrlProc

```
typedef void (*DeviceBellCtrlProc) (DeviceIntPtr devintptr,
       BellCtrl *ctrl);
```

Purpose This function allows the server to control the actions of a device with a bell.

Results Sets the value in the device's `BellCtrl` structure.

Returns None

DeviceStringCtrlProc

```
typedef void (*DeviceStringCtrlProc) (DeviceIntPtr devintptr,
       StringCtrl *ctrl);
```

Purpose This function allows the server to control the actions of a device with a display upon which a string can be displayed.

Results Sets the value in the device's `StringCtrl` structure.

Returns None

DeviceIntegerCtrlProc

```
typedef void (*DeviceIntegerCtrlProc) (DeviceIntPtr devintptr,
       IntegerCtrl *ctrl);
```

Purpose This function allows the server to control the actions of a device with a display upon which an integer can be displayed.

Results Sets the value in the device's `IntegerCtrl` structure.

Returns None

Direct Pixel Access DDX Interface 13

This chapter describes the direct pixel access (DPA) interface. DPA allows the window server to directly manipulate pixels in drawables that you control in your DDX handler. The Display PostScript (DPS) extension uses DPA to improve compositing performance. See the *Solaris X Window System Developer's Guide* for information on compositing operators.

The Direct Access Cycle

The fundamental concept of DPA is the *direct access cycle*. In a direct access cycle (or cycle), the DPA user (for example, you or the DPS extension) follows these steps:

1. Call the `directAccessOK()` function to inquire whether DPA is allowed for a given drawable or pair of drawables.
2. If DPA is allowed, call the `directAccessStart()` function to begin a cycle.
3. Access the pixels.
4. Call the `directAccessEnd()` function to end the cycle.

13

Requirements for Drawables Using DPA

DPA can only be used for the pixmap and window drawables on devices with memory-mapped frame buffers that meet the following requirements. (Note that these requirements are similar to the requirements of cfb and mfb packages).

- The byte order and pixel order must match the native order of the server:
  - **SPARC** Big-endian
  - **x86** Little-endian
- Table 13-1 shows how pixels must be packed in memory:

Table 13-1 Required Pixel Packing in Memory

bitsPerPixel	bytesPerPixel
32	4
16	2
8	1
4	1/2
2	1/4
1	1/8

- Given the return values from `directAccessStart()`, `p` and `bytesPerRow`, the pointer to the beginning of a scanline `y` is given by:

```
CARD8* pStart = p + ((y+pDraw->y) * pixelsPerRow)
```

  If `bytesPerPixel >= 1`, the pointer to pixel at (`x,y`) is:

```
pStart + ((x + pDraw->x) * bytesPerPixel)
```

  And if `bytesPerPixel < 1`, the pointer to the byte containing pixel at (`x,y`) is:

```
pStart + ((x + pDraw->x) >> shift)
```

bitsPerPixel	Shift
1	3
2	2
4	1

Initialization

sunDPAScreenRec

```
typedef struct{
    sunDPAMode     mode;
    sunDPAMode     dpsMarkMode;
    sunDPAcessType (*directAccessOK)(DrawablePtr, DrawablePtr);
    Bool           (*directAccessDPS)(DrawablePtr);
    Bool           (*directAccessStart)(DrawablePtr, CARD8**, int*);
    void           (*directAccessEnd)(DrawablePtr);
    CARD32         reserved[8]
} sunDPAScreenRec;
```

Arguments

`dpsMarkMode` is described in "directAccessDPS" on page 275.

`mode` is described in "sunDPAMode" on page 271.

`directAccessOK()`, `directAccessDPS()`, `directAccessStart()` and `directAccessEnd()` are defined in "Device-Supplied Routines" on page 273.

The final member of the structure is an array of integers reserve for future versions of this interface. Set these members to `0`.

sunDPAMode

`mode` is one of these available modes defined in `sunDPAMode`:

```
typedef enum {
    sunDPANone,
    sunDPACustom,
    sunDPAPixmap,
    sunDPAAllDrawables
} sunDPAMode;
```

If the mode is set to sunDPANone, DPA is disabled for screens controlled by your DDX handler.

If your DDX handler's pixmaps are simple-memory pixmaps, such as cfb pixmaps, set the mode to sunDPAPixmap to enable DPA for all pixmaps.

If your DDX handler's windows are memory mapped and the device is stateless, set the mode to sunDPAAllDrawables to enable DPA for windows and pixmaps.

If your DDX handler cannot use either of the predefined implementations, set the mode to sunDPACustom and provide your own DPA routines.

sunDPAMode and sunDPAScreenRec are defined in the dpa/sundpascr.h header file.

sunDPAScreenInit

Call the following initialization function from your DDX handler's InitOutput() function.

```
int
sunDPAScreenInit(pScreen, pDPAdevfuncs)
    ScreenPtr      pScreen;
    sunDPAScreenRec*pDPAdevfuncs;
```

Arguments pDPAdevfuncs is a pointer to a sunDPAScreenRec.

If a handler does not call sunDPAScreenInit, DPA is disabled for screens controlled by your DDX handler.

Since many DDX handlers require very simple and common DPA handler functions, two predefined implementations are provided. For these two modes the function pointers directAccessOK(), directAccessStart(), and directAccessEnd() are ignored.

Device-Supplied Routines

sunDPAAccessType

```
sunDPAAccessType (*directAccessOk)(DrawablePtr pDraw1,
    DrawablePtr pDraw2)
```

Purpose This function determines whether simultaneous DPA is possible for two drawables. You must provide this function if your DDX handler's DPA mode is `sunDPACustom`.

Returns `pDraw1` and `pDraw2` are `sunDPAAccess` types for the two drawables. If `pDraw2` is `NULL`, call `directAccessOK()` to determine whether or not DPA is possible for a single drawable. The return codes are defined in `dpa/sundpatype.h`.

If DPA is not allowed for either of the drawables, `sunDPANeither` should be returned.

If DPA is allowed for both drawables at the same time, `sunDPABoth` should be returned.

If DPA is allowed for the first drawable, but not the second (or if `pDraw2` is `NULL`), `sunDPAOne` should be returned.

If DPA is only allowed for the second drawable, `sunDPATwo` should be returned.

Finally, if DPA is allowed for either of the drawables, but not at the same time, `sunDPAEitherNotBoth` should be returned. This might occur, for example, if the hardware register settings are different for the two drawables.

directAccessStart

```
Bool (*directAccessStart)(DrawablePtr pDraw, CARD8 **p,
    int *pLineBytes)
```

Purpose This function is called to begin a cycle for a drawable. Your DDX handler should set up any device state required to access the pixels in the drawable. Then set the contents of `p` to the pointer at the beginning of the drawable's frame buffer, and set `*pLineBytes` to the number of bytes per scanline in the drawable.

This function must be provided if the DDX handler's DPA mode is `sunDPACustom`.

While a cycle is in progress, the only other DDX functions that might be called are `directAccessStart()` and `pScreen->SourceValidate`. No other functions are called until the cycle has ended.

Returns If the cycle can be started, `directAccessStart()` should return `TRUE`. If a cycle cannot be started, it should return `FALSE`.

directAccessEnd

```
void (*directAccessEnd)(DrawablePtr pDraw)
```

Purpose This function is called to end a cycle for a given drawable. If your DDX handler never needs to do anything at the end of a cycle, this function pointer can be `NULL`.

directAccessDPS

```
Boll (*directAccessDPS)(DrawablePtr pDraw)
```

Purpose This function allows the DPS extension to determine whether or not it should use DPA to mark a given drawable as accessible.

Note that the return value from `directAccessOK()` tells whether DPA is *allowed* for a drawable. `directAccessDPS()` tells you whether DPS *should* use DPA. It is a performance hint. The values returned for given drawable types should be determined during performance tuning. This function must be provided if the handler specified `dpsMarkMode` as `sunDPACustom`. If `dpsMarkMode` is set to `sunDPAAllDrawables` or `sunDPAPixmap`, predefined implementations of `directAccessDPS` will be used. `directAccessStart` and `directAccessEnd` will be used to begin and end a cycle as usual.

Returns If DPS can and should use DPA to mark to the drawable, `directAccessDPS()` should return `TRUE`; otherwise, return `FALSE`.

Note – If `directAccessDPS()` returns `TRUE` for a given drawable, `directAccessStart()` must always succeed for that drawable. This is a requirement due to the design of the DPS extension.

Note – Currently, this function is not called by the window server. The system will behave as though the `dpsMarkMode` were `sunDPANone` for all drawables. This function will be used in a future release.

13

Debug Server Modules 14

A version of the X window server is available for debugging purposes. It is included in the SUNWxwdes (SPARC), SUNWxwdex (x86), and SUNWxwdep (PPC) packages. Use the debug server with dbx(1).

```
example% cd /opt/SUNWddk/ddk_2.5/xserver/bin/sparc
example% dbx Xsun-ddkdebug
```

The source code for some of the dynamic libraries is also in the DDK CD-ROM. Use dbx's `file` and `use` commands to step through the dynamic code.

```
(dbx) stop in miSpritePolyFillRect
(dbx) cont
stopped in miSpritePolyFillRect at 0xeec15e60
miSpritePolyFillRect+0x2c: ld [%fp + 68], %o0
warning: can't find source
/export/ddk/ea2/bin/Xsun/mit/server/ddx/mi/misprite.c
(dbx) use /opt/SUNWddk/ddk_2.4/xserver/server/ddx/mi
(dbx) file misprite.c
```

Now you can step through the code examining values as necessary.

For x86 systems – This does not work on x86 because the `-xs` compiler switch is not supported. However, you can still print out the arguments to functions.

As a device driver developer, you are most likely interested in the initialization stage of your driver. However, since the server loads your driver dynamically, its symbols are not available to you at startup time. You can stop the server before device initialization in the `AddScreen` function. This function contains the address of which it is going to switch to initialize the framebuffer device.

```
(dbx) stop in AddScreen
(dbx) run
AddScreen(pfnInit = &xxxxxxInit () at 0xef7628a4, argc = 1, argv = 0xeffffaac) at 0x51f50
```

The `pfnInit()` function pointer should point to your device driver's initialization function. Now that your dynamic library has been loaded, you can set breakpoints and step through your code in `dbx`.

MIT Shared Memory Extension 15

This chapter describes the functions that a ddx handler may call to enable full functionality of the MIT Shared Memory (MIT_SHM) extension. This extension is an MIT standard that is distributed with X11 Release 5 (X11R5).

The MIT_SHM extension is a version of the `ximage` interface where the actual image data is stored in a shared memory segment. This extension can yield a significant increase in performance for large images.

The following document is part of the MIT_SHM extension, and is online in the `doc/hardcopy/Xext` directory.

- *MIT_SHM - The MIT Shared Memory Extension*, Jonathan Corbet, formatted and edited for release 5 by Keith Packard, MIT X Consortium.

15

MIT Shared Memory Interface

All ddx handlers may use the following functions to implement the MIT Shared Memory extension.

Table 15-1 MIT Shared Memory Extension Functions

Function Name	Description
`ShmRegisterFbFuncs`	Registers the cfb-compatible functions. The ddx handler must accept fake pixmaps. Fake pixmaps are pixmaps with `devPrivates` initialized to NULL and an internal format compatible with cfb. Note that `ShmRegisterFbFuncs` is called in `mpgScreenInit` and `miScreenInit`, so if your handler calls either of these functions, no other work is required.
`ShmRegisterFuncs`	Registers the specified shared memory function vectors. Note that to enable creation of shared memory pixmaps, you must use `ShmRegisterFuncs` or `ShmRegisterFbFuncs`.
`ShmSetPixmapFormat`	Registers the specified pixmap format.

ShmRegisterFbFuncs

```
void ShmRegisterFbFuncs(ScreenPtr pScreen)
```

Purpose This function is provided to register the predefined shared memory functions. The predefined `ShmFuncs` record is registered as follows:

```
ShmFuncs fbFuncs={fbShmCreatePixmap, fbShmPutImage};
```

Called by This function is called during device screen initialization.

Results This function initializes the `shmFuncs` array indexed by the specified screen number with the `ShmFuncs` record.

Returns None.

ShmRegisterFuncs

```
void ShmRegisterFuncs(ScreenPtr pScreen, ShmFuncsPtr funcs)
```

Purpose This function is provided to register the shared memory functions. The ShmFuncsPtr has been defined as follows:

```
typedef struct _ShmFuncs {
PixmapPtr   (* CreatePixmap)();
void        (* PutImage)();
} ShmFuncs, *ShmFuncsPtr;
```

Called by This function is called during device screen initialization.

Results This function initializes the `shmFuncs` array indexed by the specified screen number with `funcs`.

Returns None.

ShmSetPixmapFormat

```
void ShmSetPixmapFormat(ScreenPtr pScreen, int format)
```

Purpose This function is provided to register the shared memory pixmap format. The valid pixmap formats are XYPixmap, XYBitmap or ZPixmap.

Called by This function is called during device screen initialization.

Results This function initializes the `shmPixFormat` array indexed by the specified screen number with `format`.

Returns None.

15

The `OWconfig` *File*

A

The `OWconfig` file is used by the server to dynamically load extensions, XInput modules, and DDX graphics handler modules. By default, the `OWconfig` file is distributed in the `/usr/openwin/server/etc` directory.

The format of the `OWconfig` file is an uncommitted interface between the OpenWindows 3.4 server and dynamically loaded modules. This file is a server-private file. It is read by the OpenWindows server and edited by IHV installation scripts (see Appendix B, “Packaging and Installation Hints”).

For x86 systems – The `OWconfig` file can be edited by the `kdmconfig` utility. This utility runs during installation. You can also invoke `kdmconfig` any time after installation to tailor your configuration.

For PowerPC systems – The OWconfig file can be edited by the `kdmconfig` utility. This utility runs during installation. You can also invoke `kdmconfig` any time after installation to tailor your configuration.

A

SPARC: Sample OWconfig File

Code Example A-1 lists a sample SPARC `OWconfig` file.

Code Example A-1 Sample SPARC `OWconfig` File

```
# Start SUNWxwplt
# Copyright 1993 Sun Microsystems, Inc.
#"@(#)OWconfig1.11 26 May 1993 SMI"
# OWconfig file for OpenWindows X server Version 3.4
#
# WARNING: This file is automatically generated when
# the OpenWindows software package is installed. This file can be
# automatically edited by other optional software packages that
# are installed on the system.
#      ANY CHANGES YOU MAKE TO THIS FILE WILL BE LOST DURING
#      PACKAGE INSTALLATION, REMOVAL AND UPGRADES!
#      The format of this file is private to the OpenWindows
#      X Server and subject to change in future releases.

# X Display
class="XDISPLAY" name="0"
    coreKeyboard="IKBD" corePointer="IMOUSE";

# CG6 display adapter
class="XSCREEN" name="SUNWcg6"
    ddxHandler="ddxSUNWcg6.so.1" ddxInitFunc="sunCG6Init";

# CG3 display adapter
class="XSCREEN" name="SUNWcg3"
    ddxHandler="ddxSUNWcg3.so.1" ddxInitFunc="sunCG3Init";

# CG4 display adapter
class="XSCREEN" name="SUNWcg4"
    ddxHandler="ddxSUNWcg4.so.1" ddxInitFunc="sunCG4Init";

# BW2 display adapter
class="XSCREEN" name="SUNWbw2"
    ddxHandler="ddxSUNWbw2.so.1" ddxInitFunc="sunBW2Init";

# CG8 display adapter
class="XSCREEN" name="SUNWcg8"
    ddxHandler="ddxSUNWcg8.so.1" ddxInitFunc="sunCG8Init";

# CG2 display adapter
class="XSCREEN" name="SUNWcg2"
```

Code Example A-1 Sample SPARC `OWconfig` File *(Continued)*

```
    ddxHandler="ddxSUNWcg2.so.1" ddxInitFunc="sunCG2Init";

# sun Keyboard module
class="XINPUT" name="IKBD"
    ddxHandler="ddxSUNWkbd.so.1"
    ddxInitFunc="ddxSUNWkbdProc";

# sun Mouse module
class="XINPUT" name="IMOUSE"
    ddxHandler="ddxSUNWmouse.so.1"
    ddxInitFunc="ddxSUNWmouseProc";


# sun Dials Compatibility module
class="XINPUT" name="IDIALSC"
    ddxHandler="ddxSUNWdialsCompat.so.1"
    ddxInitFunc="ddxSUNWdialsCompatProc";

# sun Dials module
class="XINPUT" name="IDIALS"
    ddxHandler="ddxSUNWdials.so.1"
    ddxInitFunc="ddxSUNWdialsProc";

# sun Buttons module
class="XINPUT" name="IBUTTONS"
    ddxHandler="ddxSUNWdials.so.1"
    ddxInitFunc="ddxSUNWbuttonsProc";

# Example of a dynamically loaded extension "ACMExtn"
# class="XEXTENSION" name="ACMExtn"
#   sharedObject="ACMExtn.so.1"
#   initFunc="ACMExtnExtensionInit"
#   preLoad="NO";

# End SUNWxwplt
```

A

x86: Sample OWconfig File

Code Example A-2 lists a sample x86 `OWconfig` file.

Code Example A-2 Sample x86 `OWconfig` File

```
# Start SUNWxwpls
# Copyright 1993 Sun Microsystems, Inc.
#"@(#)OWconfig.x861.14 21 Dec 1993 SMI"
# OWconfig file for OpenWindows Version 3.4

# X Display
class="XDISPLAY" name="0"
# Please make sure that one of the two following lines regarding the
# type of mouse is always uncommented.
# It is assumed that you are using a Logitech Mouseman serial mouse by
# default.
#
# Logitech Mouseman Serial Mouse
 coreKeyboard="ATKBD" corePointer="MOUSEMAN-S"


# Logitech Bus Mouse
# coreKeyboard="ATKBD" corePointer="LOGI-B"
    dev0="/dev/fb"
    listOfScreens="my8514";

# Sample XSCREENCONFIG class
class="XSCREENCONFIG" name="my8514"
    device="8514"
    pmifile="/usr/openwin/etc/vesa/8514/ati.pmi"
    res="1024x768";

# Standard VGA display adapter, 640x480 and 16 colors.
class="XSCREEN" name="vga4"
    ddxHandler="ddxSUNWvga4.so.1" ddxInitFunc="vga4Init";

# Standard VGA display adapter, 800x600 and 16 colors.
# Panning within a 640x480 window
class="XSCREEN" name="vga4"
    ddxHandler="ddxSUNWvga4.so.1" ddxInitFunc="vga4Init";

# 8514 display adapter
class="XSCREEN" name="8514"
    ddxHandler="ddxSUNW8514.so.1" ddxInitFunc="i8514Init";
```

Code Example A-2 Sample x86 `OWconfig` File

```
# Super VGA display adapter, 1024x768 and 256 colors.
class="XSCREEN" name="vga8"
    ddxHandler="ddxSUNWvga8.so.1" ddxInitFunc="vga8Init";

# PC Keyboard module
class="XINPUT" name="ATKBD"
    ddxHandler="ddxSUNWatkbd.so.1"
    ddxInitFunc="ATKbdProc"
    layout="1"
    type="101";

# Mouseman module
class="XINPUT" name="MOUSEMAN-S"
    ddxHandler="ddxSUNWx86mouse.so.1"
    ddxInitFunc="ddxSUNWmouseProc"
    buttons="3"
    strmod="vuidm4p"
    dev="/dev/tty00";


# Logitech serial module
#class="XINPUT" name="LOGI-S"
# ddxHandler="ddxSUNWx86mouse.so.1"
# ddxInitFunc="ddxSUNWmouseProc"
# buttons="3"
# strmod="vuidm5p"
# dev="/dev/tty00";

# Logitech bus module
class="XINPUT" name="LOGI-B"
    ddxHandler="ddxSUNWx86mouse.so.1"
    ddxInitFunc="ddxSUNWmouseProc"
    buttons="3"
    strmod="vuidm5p"
    dev="/dev/logi";

# 3 button Kdmouse bus module
#class="XINPUT" name="KDMOUSE-B"
#   ddxHandler="ddxSUNWx86mouse.so.1"
#   ddxInitFunc="ddxSUNWmouseProc"
#   buttons="3"
#   strmod="vuid3ps2"
#   dev="/dev/kdmouse";
```

Code Example A-2 Sample x86 `OWconfig` File

```
# Microsoft serial module
#class="XINPUT" name="MS-S"
#   ddxHandler="ddxSUNWx86mouse.so.1"
#   ddxInitFunc="ddxSUNWmouseProc"
#   buttons="3"
#   strmod="vuidm3p"
#   dev="/dev/tty00";

# Microsoft bus module
#class="XINPUT" name="MS-B"
#   ddxHandler="ddxSUNWx86mouse.so.1"
#   ddxInitFunc="ddxSUNWmouseProc"
#   buttons="3"
#   trmod="vuidm5p"
#   dev="/dev/msm";
# End SUNWxwpls
```

PowerPC: Sample OWconfig File

Code example A-3 lists a sample PowerPC `OWconfig` file.

Code Example A-3 Sample PowerPC `OWconfig` File

```
# Start SUNWxwpls
# Copyright 1993 Sun Microsystems, Inc.
#"@(#)OWconfig.ppc 1.6     95/06/05 SMI"
# OWconfig file for OpenWindows X server Version 3.3
# WARNING: This file is automatically generated when the
#     OpenWindows software package is installed. This file may be
#      automatically edited by other optional software packages
#      that are installed on the system.
#      ANY CHANGES YOU MAKE TO THIS FILE WILL BE LOST DURING
#      PACKAGE INSTALLATION, REMOVAL AND UPGRADES !
#      The format of this file is private to the OpenWindows
#      X Server and subject to change in future releases.

# X Display
class="XDISPLAY" name="0"
    coreKeyboard="IKBD" corePointer="IMOUSE"
    dev0="/dev/fb";
```

Code Example A-3 Sample PowerPC `OWconfig` File *(Continued)*

```
# S3/928 display adapter
class="XSCREEN" name="SUNWs3"
    ddxHandler="ddxSUNWdfb.so.1" ddxInitFunc="sundfbInit";

# Weitek p9000 display adapter
class="XSCREEN" name="SUNWp9000"
    ddxHandler="ddxSUNWdfb.so.1" ddxInitFunc="sundfbInit";

# Weitek p9100 display adapter
class="XSCREEN" name="SUNWp9100"
    ddxHandler="ddxSUNWdfb.so.1" ddxInitFunc="sundfbInit";

# Western Digital display adapter
class="XSCREEN" name="SUNWwd90c24a2"
    ddxHandler="ddxSUNWdfb.so.1" ddxInitFunc="sundfbInit";

# S3/864 display adapter
class="XSCREEN" name="SUNWicecube"
    ddxHandler="ddxSUNWdfb.so.1" ddxInitFunc="sundfbInit";

# Cirrus Logic display adapter
class="XSCREEN" name="SUNWclgd5434"
    ddxHandler="ddxSUNWdfb.so.1" ddxInitFunc="sundfbInit";

# sun Keyboard module
class="XINPUT" name="IKBD"
    ddxHandler="ddxSUNWkbd.so.1"
    ddxInitFunc="ddxSUNWkbdProc";

# sun Mouse module
class="XINPUT" name="IMOUSE"
    ddxHandler="ddxSUNWmouse.so.1"
    ddxInitFunc="ddxSUNWmouseProc";
```

File Format Definition

The `OWconfig` file is composed of a number of *resource* entries, described by a collection of lines similar to a kernel device driver's `.conf` file (see `driver.conf`(4)). A resource is typically a device, such as a frame buffer or a keyboard. Each resource entry consists of a number of "`attribute=value`" pairs, separated by white space (including spaces, tabs, and new line

characters) and terminated by a semicolon (;) character. Any characters following a "#" through the end of the line are treated as a comment and disregarded.

```
#Sample OWconfig entry
class="class name" name="name"
    [property-name=value ...];
```

The quotes around the value strings are required only if the string contains delimiters (such as white space or ";" (semicolon)). The back slash character "\" is used as an escape character. For example, \" could be used to include the " character as part of a string value. The parsing routines strip the quotes surrounding string values and pass just the string to the underlying software. The parsing software treats all values as strings; the interpretation of the string value is up to you.

Each resource entry in the file completely defines an instance of a *class*. For each resource class, there is a set of attributes pertaining to that class. Values for the class and name attributes are required in every resource entry. The class attribute defines the class of the resource. It can be one of the following:

- XDISPLAY
- XSCREENCONFIG
- XSCREEN
- XINPUT
- XEXTENSION

The name attribute identifies the particular resource through a string unique to the class (such as SUNWcg6, IKBD, MIT-SHM). Each class might define additional mandatory attributes specific to that class. Each class is discussed in greater detail starting on page 292.

To avoid name space collisions between multiple vendors, it is strongly recommended (as in `driver.conf`(4)) that the name attribute for vendor-specific classes begin with a vendor-unique string. A reasonably compact and unique choice is the vendor over-the-counter stock symbol. With other classes, such as XEXTENSION, name space collisions can be avoided by registering extension names with the Xregistry (maintained by the MIT X Consortium).

File and Module Search Paths

By default, OpenWindows is installed in `/usr/openwin`. The directory `/usr/openwin/server/etc` contains the default `OWconfig` file that is distributed with the OpenWindows software. Similarly, the directory `/usr/openwin/server/modules` will contain the DDX handler modules, Xinput modules and extension modules that are distributed as part of the X Windows package. These constitute components that are distributed and maintained by Sun.

In addition to this, DDX support utility libraries, such as cfb, mfb, mi, mpg and server private libraries such as font, typescaler, and dga are located in the directory `/usr/openwin/server/lib`.

Since `/usr/openwin` can be an NFS-mounted installation that is shared by multiple machines on the network, you need a machine-specific configuration directory to describe the local system configuration. You must create this machine-specific directory path in your installation scripts since it is not created by default nor required. The file that describes the local configuration is the `OWconfig` file. The server searches for the `OWconfig` file in `/etc/openwin/server/etc`.

For SPARC systems – It is optional to have an `OWconfig` file in `/etc/openwin/server/etc` because by default, `/usr/openwin/server/etc` contains the default `OWconfig` file.

For x86 systems – It is *not* optional to have an `OWconfig` file in `/etc/openwin/server/etc`; the `kdmconfig` utility *always* creates the file in `/etc/openwin/server/etc`. Your installation script can edit the `/etc/openwin/server/etc/OWconfig` file.

For PowerPC systems – It is *not* optional to have an `OWconfig` file in `/etc/openwin/server/etc`; the `kdmconfig` utility *always* creates the file in `/etc/openwin/server/etc`.

The `OWconfig` search path is:

```
/etc/openwin/server/etc:/usr/openwin/server/etc
```

Dynamically loaded modules (XInput, extensions, or DDX handlers) can be located in `/etc/openwin/server/modules`. The search path for loadable modules is:

```
/etc/openwin/server/modules:/usr/openwin/server/modules
```

If an `OWconfig` file is present in both locations, both files are read, and the server merges these files into a single database. If there are conflicting entries in both files (when an entry has the same values for the “class” and “name” attributes in both files), the server merges both entries on a per-attribute basis. That is, the entry from `/etc/openwin/server/etc` will take precedence over the entry from the file in `/usr/openwin/server/etc`. If there are duplicate entries within the same file (when an entry has the same values for the “class” and “name” attributes in the same file), then the last entry for either of these attributes is used.

See Appendix B, “Packaging and Installation Hints” for more details.

The XDISPLAY Class

An XDISPLAY is a collection of graphics output and input devices that the X server manages. It is a collection of Screens, Core Keyboard and Core Pointer.

```
# XDISPLAY

class="XDISPLAY" name="0"
    coreKeyboard="IKBD" corePointer="IMOUSE"
    listOfScreens="myGX:my2ndHead,left";
```

The attributes `coreKeyboard` and `corePointer` select devices of class XINPUT as the core keyboard and pointer respectively.

`listOfScreens` is an optional attribute that is new to this release:

```
[name[:name[,left|right|top|bottom]]]
```

If this attribute is not present, the graphics adapter selection defaults to `/dev/fb`. The value of `listOfScreens` is a colon-separated list of names of objects of class XSCREENCONFIG. The names can be modified by geometry specifiers (left, right, top or bottom). The semantics of these specifiers are equivalent to the command-line modifiers by the same name. If no geometry specifier is entered or an erroneous specifier is read, then the default value is "right". See the `Xsun`(1) man page.

The Screens specified in `listOfScreens` are added in order. In the above example, the server recognizes *myGX* as Screen 0 and *my2ndHead* as Screen 1.

For x86 systems – If the display adapter is not associated with the kernel driver (for vga4, vga8 and 8514) the `listOfScreens` attribute *must* exist. The `kdmconfig` utility will create a `listOfScreens` attribute and value in the XDISPLAY class entry.

The XSCREENCONFIG Class

An XSCREENCONFIG *instantiates* an object of class XSCREEN and abstracts the per-instance configuration information.

```
#XSCREENCONFIG
class="XSCREENCONFIG" name="my8514"
    device="/dev/fb0"          # SPARC example
    device="8514"              # x86 example
    dpix="90" dpiy="90"
    defclass="PseudoColor"
    defdepth="8"
    grayvis="NO"
    res="1024x768"             # x86 example
    pmifile="/usr/openwin/etc/vesa/i8514/ati.pmi";# x86 example
```

The name attribute is referenced in the `listOfScreens` of the XDISPLAY class. The value of the name attribute is not important; however, the actual names generated should be unique within an instance of the `OWconfig` file. It is up to you, the IHV, to generate a meaningful name (*my8514* is an x86 example). The `OWconfig` file specification and the X server do not attach any meaningful semantic to the actual value of this name.

The device attribute is equivalent to the `-dev` command-line option as specified for `Xsun`.

The `dpix`, `dpiy`, `defclass`, `defdepth`, and `grayvis` attributes are optional and are equivalent to the `-dev` command-line option as specified for `Xsun`(1).

The value of the device attribute depends on whether a kernel graphics device driver is associated with the display adapter or frame buffer. If a driver exists (as is always the case on SPARC), the device attribute value is the device special filename associated with the driver (for example, `/dev/fb0`). If a driver does not exist (as can happen with several x86 graphics adapters), the device attribute value is a descriptive name of the graphics adapter (for example, 8514), and corresponds directly to the name of an object of class XSCREEN.

The XSCREEN Class

An XSCREEN is a graphics display adapter, or frame buffer.

```
# XSCREEN
class="XSCREEN" name="SUNWcg6"
    ddxHandler="ddxSUNWcg6.so.1" ddxInitFunc="sunCG6Init";
```

The value of the name attribute depends on whether a kernel graphics device driver is associated with the display adapter or frame buffer. If the kernel driver exists, it is probed with the `VIS_GETIDENTIFIER ioctl` to determine the name of the object of class XSCREEN that is loaded by the server. For more information on drivers, see *Writing Device Drivers*.

For x86 systems – The name attribute is a descriptive name of the graphics adapter and corresponds directly to the value of the device attribute in an object of class XSCREENCONFIG.

The `ddxHandler` follows the naming convention `ddx`<*organization*><*device*>.so.<*majorVersion*>. The initialization function is the single symbolic entry point into the DDX handler. To avoid namespace collisions, it is recommended that IHV's prefix the `InitFunc` name with an

<*organization*><*device*> prefix. It is further recommended that all symbols internal to the DDX handler, and symbols in support libraries linked to the DDX handler (if any), be similarly prefixed to minimize namespace collisions.

The XINPUT Class

The XINPUT class is for X Input Extension modules and X input core Keyboard and Pointer modules.

SPARC: Sample XINPUT Class

```
# sun Keyboard module
class="XINPUT" name="IKBD"
    ddxHandler="ddxSUNWkbd.so.1"
    ddxInitFunc="ddxSUNWKbdProc";
# sun Mouse module
class="XINPUT" name="IMOUSE"
    ddxHandler="ddxSUNWmouse.so.1"
    ddxInitFunc="ddxSUNWmouseProc";
```

x86: Sample XINPUT Class

```
# 3-button Kdmouse bus module
class="XINPUT" name="KDMOUSE-S"
    ddxHandler="ddxSUNWx86mouse.so.1"
    ddxInitFunc="ddxSUNWmouseProc"
    buttons="3"
    strmod="vuid3ps2"
    dev="/dev/kdmouse";
```

PowerPC: Sample XINPUT Class

```
# PowerPC Keyboard module
class="XINPUT" name="ATKBD" layout="1" type=101"
    ddxHandler="ddxSUNWkbd.so.1"
    ddxInitFunc="ddxSUNWkbdProc"
    dev="/dev/vt00";
# PowerPC Mouse module
class="XINPUT" name="KDMOUSE-B"
    ddxHandler="ddxSUNWx86mouse.so.1"
    ddxInitFunc="ddxSUNWmouseProc"
    strmod="vuidps2"
    dev="/dev/kdmouse";
```

XINPUT modules follow the naming convention:

```
ddx<organization><device>.so.<majorVersion>.
```

Devices of class XINPUT are selected as the `coreKeyboard` or `corePointer` devices by setting the attributes in the XDISPLAY class to the appropriate value. See "The XDISPLAY Class" on page 292.

The XEXTENSION Class

The XEXTENSION class is for X Extension modules that are dynamically loaded by the server.

```
# XEXTENSION named ACMExtn
class="XEXTENSION" name="ACMExtn"
    sharedObject="ACMExtn.so.1"
    initFunc="ACMExtnExtensionInit"
    preLoad="NO";
```

In this case, the extension name should be registered in the Xregistry (maintained by the MIT X Consortium) to avoid name space collisions. The value of the `preLoad` attribute can be `YES` or `NO` depending on whether you want the server to load this extension at startup (YES), or when `XQueryExtension` is called (NO). Either way, `XListExtensions` lists all

statically linked extensions in the server and dynamically loadable extensions with an entry in the `OWconfig` file. `XListExtensions` simply lists extensions; it does not cause the extensions to be dynamically loaded.

OWconfig Access Method

The OWconfig Access Method standardizes access to and manipulation of an OpenWindows configuration (OWconfig) database file. If your DDX handler requires configuration information, use this method to access that information specific to your device. Note that not all DDX handlers require configuration information.

OWconfig Database

An OWconfig database is a hierarchical list of name/value pairs. The meaning of a particular name/value pair depends upon its position in the hierarchy, as well as the application's interpretation of its value. More concretely:

- An OWconfig database is a list of "classes"; each "class" has a name.
- A "class" is a list of "instances"; each "instance" has a name.
- An "instance" is a list of "attributes"; each "attribute" has a name and a value.

As an example, a typical OWconfig database file contains a declaration of an instance of class "XDISPLAY" whose name is "0" (for screen 0). This instance of the "XDISPLAY" class may contain definitions for attributes such as "coreKeyboard" and "corePointer." The OWconfig file may contain several declarations of instances of class "XDISPLAY."

The access method does not enforce class/instance/attribute naming conventions, nor does it check values of attributes.

OWconfig API

The C language definition of the OWconfig Access Method API may be found in the include file `/usr/openwin/include/X11/Sunowconfig.h`.

OWconfigGetClassNames

```
char **
OWconfigGetClassNames(char *class)
```

Purpose All users of this function should call `OWconfigFreeClassNames` to free the list and the strings to which it points.

Returns (`char **`) to list of class instance names, or NULL if class did not exist. The end of the list is indicated by a NULL pointer.

Arguments `class`: name of class for which to name all instances.

OWconfigFreeClassNames

```
void
OWconfigFreeClassNames(char **list)
```

Purpose Frees results of `OWconfigGetClassNames`.

Arguments list: NULL terminated list of strings to free.

OWconfigGetAttribute

```
char *
OWconfigGetAttribute(char *class, char *name, char *attribute)
```

Returns (`char *`) to value of attribute or NULL if attribute could not be found. The string returned by this function can be freed using `OWconfigFreeAttribute`.

Arguments `class`: name of class to which named attribute belongs

`name`: name of instance of class to which named attribute belongs

`attribute`: name of sought attribute

OWconfigFreeAttribute

```
void
OWconfigFreeAttribute(char *attribute)
```

Purpose Frees string returned by `OWconfigGetAttribute`.

Arguments attribute: string, allocated by OWconfigGetAttribute, to be freed.

OWconfigGetInstance

```
OWconfigAttributePtr
OWconfigGetInstance(char *class, char *name, int *numberInAttr)
```

Purpose `OWconfigGetInstance` returns a list of attribute definitions. Use `OWconfigFreeInstance` to free the memory allocated to the information returned by `OWconfigGetInstance`.

Arguments `class`: name of class from which to list attributes

`name`: name of instance of class from which to list attributes

`numberInAttr`: receives number of attributes in returned list

Returns `OWconfigAttributePtr` or NULL.

OWconfigFreeInstance

```
void
OWconfigFreeInstance(OWconfigAttributePtr attr, int
       numberInAttr)
```

Frees a list created by OWconfigGetInstance.

Arguments `attr`: list of attributes to free

`numberInAttr`: number of attributes in list

Packaging

The API components of the access method are in the following files:

- `/usr/openwin/lib/libowconfig.so.1`
- `/usr/openwin/lib/libowconfig.so`
- `/usr/openwin/include/X11/Sunowconfig.h`

Typical Usage

If you want to retrieve configuration information for your device from the OWconfig database you will need to use, at a minimum, the `OWconfigGetAttribute` and `OWconfigFreeAttribute` functions. Note that not all DDX handlers require configuration information.

By the time your DDX handler's initialization function is called, the server has loaded into memory a copy of the OWconfig database. The functions in "OWconfig API" on page 297 are provided as a read-only access method to this database. There are two types of configuration information that you may want to access:

- attributes documented in the DDK manual

  To access these attributes use the documented class and name values as part of an appropriate OWconfig function call.

- attributes added to an `OWconfig` file as part of your installation process (configuration information specific to your device)

To access these attributes you must first know how to access the OWconfig attribute that belongs to your device. The following code illustrates how to get this information:

```
int
ddxACMEProc(DevicePtr pAcme, int what)
{
char *tmp;
...
switch(what) {
case DEVICE_INIT:
...

/* The following illustrates how to get configuration */
/* information belonging to this device (ddx) driver ,which */
/* in this case belongs to the XINPUT class, and which */
/* contains an attribute called dev. */
tmp = OWconfigGetAttribute(”XINPUT”,
    ((DeviceIntPtr)pAcme)->devEntry->tag, “dev”);
...
```

The `tag` value is the key to locating information for a particular device driver.

A

Packaging and Installation Hints B

The Loadable DDX interface introduces issues pertaining to packaging and installation of loadable modules (DDX handlers, Xinput modules and X Extension modules). This appendix discusses these issues and assumes familiarity with the Application Packaging & Installation facilities in Solaris 2.x. See the *SunOS 5.x Application Packaging and Installation Guide* for more information.

Installation Hints

Loadable modules and `OWconfig` file entries are installed in either of two directories, as discussed in “File and Module Search Paths” on page 291. The directories in `/etc/openwin` are intended to be machine-specific, or local, whereas the directories in `/usr/openwin` could be either local to the machine or NFS mounted from a remote filesystem. The `/etc/openwin` location is recommended for most loadable modules installed by IHVs (Independent Hardware Vendors). The exception is when a module is being installed on a server for shared use by a number of workstations requiring the module. In this case, install the module in the same directory in which either the SUNWxwplt (SPARC) or SUNWxwpls (x86) package was installed.

Whether you install a module in the `/etc` or `/usr` location, your installation script should always checks for an `OWconfig` file and the relevant entries in that location. If an `OWconfig` file does not exist in the installation location, the installation script should create it with the relevant module entries inserted in

the file. The package should also have a corresponding removal script that removes any entries inserted by it into the `OWconfig` file. It should delete the file if (and only if) it becomes empty as a result of the deletions.

If you use the `/etc` location for installation, the installation script takes into account the fact that there might not be sufficient space in the `/` filesystem to accommodate large loadable modules. The recommended approach is to install the DDX modules in a subdirectory under `/opt/<package_name>`, and populate the `/etc/openwin/server/modules` directory with symbolic links. Install and edit the `OWconfig` file in the `/etc` location directly, not via symbolic links.

Packaging Hints

Follow the following convention for package names:

```
<organization><package-descriptor>
```

For example:

- SUNWxwplt Sun's OpenWindows required package for SPARC
- SUNWxwpls Sun's OpenWindows required package for x86
- ACMEowdyn ACME dynamo frame buffer's DDX handler package

The typical convention is that packages edit the `OWconfig` file to insert entries with the following comment lines containing the package name. The package in this example is ACMEowdyn.

```
# Start ACMEowdyn
# [a number of lines containing the actual OWconfig entry]
# End ACMEowdyn
```

The SUNWxwplt package, for example, marks all of the default entries it installs (in `/usr` location) as follows:

```
# Start SUNWxwplt
# [a number of lines containing the default OWconfig entries]
# End SUNWxwplt
```

Package Delivery Example

The following is an example of the packaging scripts and prototype files for delivering a package containing the DDX handler module for the ACME dynamo graphics display adapter. All of these examples are for the ACMEowdyn package.

Code Example B-1 `pkginfo` File

```
PKG=ACMEowdyn
NAME=ACME Dynamo Display Adapter Support
ARCH=sparc
VERSION=1.0.0,REV=2.2.2
CATEGORY=system,graphics
PRODNAME=Dynamo
PRODVERS=2.3
DESC="OpenWindows dynamically loaded drivers for the Dynamo
display adapter. Not needed if you do not have a Dynamo display
adapter installed on your system."
BASEDIR=/etc
VENDOR="ACME Display Adapters, Inc."
HOTLINE="1-800-USA-ACME"
EMAIL="hotline@ACME.COM"
MAXINST=1000
CLASSES=base OWconfig
```

Code Example B-2 `Prototype` File

```
i pkginfo
i copyright
i depend
i i.OWconfig
i r.OWconfig
d base openwin 0775 root bin
d base openwin/server 0775 root bin
d base openwin/server/etc 0775 root bin
e OWconfig openwin/server/etc/OWconfig 0755 root bin
d base openwin/server/modules 0775 root bin
f base openwin/server/modules/ddxACMEdyn.so.1 0755 bin bin
```

Put the following code in a stub file named `OWconfig`.

Code Example B-3 `OWconfig` File

```
# Start ACMEowdyn
# ACME dynamo display adapter
class="XSCREEN" name="ACMEdyn"
    ddxHandler="ddxACMEdyn.so.1" ddxInitFunc="ACMEdynInit";
# End ACMEowdyn
```

Code Example B-4 `i.OWconfig` File

```
#
# Installation script for the OWconfig class
# If an OWconfig file existed, remove any entry belonging to
# this package, and append a new entry.
#
while read src dst
do
    if [ -r $dst ]
    then
        # An OWconfig file already exists
        if [ -w $dst ]
        then
            # It's editable by this script, edit it.
            cp $dst /tmp/$$OWconfig || exit 2
            sed -e "/# Start ACMEowdyn/,/# End ACMEowdyn/d" \
            /tmp/$$OWconfig > $dst || exit 2
            cat $src >> $dst || exit 2
            rm -f /tmp/$$OWconfig
        else
            # An OWconfig file exists that's not editable !
            exit 2
        fi
    else
        # An OWconfig file was not present
        cat $src >> $dst || exit 2
    fi
done
exit 0
```

Code Example B-5 `r.OWconfig` File

```
#
# Removal script for the OWconfig class
# Remove any entries that belong to this package.
# Delete the file if it’s empty.
#
while read dst
do
    sed -e ~~/# Start ACMEowdyn/,/# End ACMEowdyn/d” $dst > \
    /tmp/$$OWconfig || exit 2
    if [ -s /tmp/$$OWconfig ]
    then
        mv /tmp/$$OWconfig $dst || exit 2
    else
        rm $dst || exit 2
    fi
done
exit 0
```

Code Example B-6 `depend` File

```
P SUNWcar      Core Architecture, (Root)
P SUNWkvm      Core Architecture, (Kvm)
P SUNWcsr      Core Sparc, (Root)
P SUNWcsu      Core Sparc, (Usr)
P SUNWcsd      Core Sparc Devices
P SUNWxwplt    OpenWindows required core package for SPARC
P SUNWxwpls    OpenWindows required core package for x86
```

Code Example B-7 `copyright` File

```
Copyright 1993 ACME Display Adapters, Inc.
    <insert your copyright information here>
All Rights Reserved.
```

B

Virtual User Input Device Interface C

This appendix discusses the manipulation of workstation data, which is mostly global data related to input and input devices. This chapter also explains the mechanism that sets up input devices to generate event codes and how a device driver conforms to the Virtual User Input Device (vuid) interface.

Virtual User Input Device (vuid)

The vuid is a possible interface between input devices and the device handler. Device drivers in OpenWindows 3.3 must read themselves and are *not required* to generate vuid events. Devices can generate vuid events, a variation of the vuid format, or a totally new format. The vuid format provided in this appendix is an example format.

What Kind of Devices?

Vuid is targeted to input devices that gather command data from users. Examples of these devices are: mice, keyboards, joysticks, light pens, knobs, sliders, buttons, and ascii terminals. The vuid interface is not designed to support input devices that produce extremely large amounts of data, such as input scanners, disk drives, and voice packets.

C

Vuid Station Codes

This section defines the layout of the address space of vuid station codes. It explains how to extend the vuid address space.

Address Space Layout

The address space for vuid events is 16-bits long, from 0 to 65535 inclusive. It is broken into 256 segments that are 256 entries long (`VUID_SEG_SIZE`). The top 8 bits contain a vuid segment identifier value. The bottom 8 bits contains a segment-specific value from 0 to 255. Some segments are predefined and some are available for expansion. Here is how the address space is currently broken down:

- `ASCII_DEVID` (0x00) — ASCII codes, which include `META` codes.
- `TOP_DEVID` (0x01) — Top codes, which are ASCII with the 9th bit on.
- Reserved (0x02 to 0x7F) — For Sun vuid implementations.
- Reserved for Sun customers (0x80 to 0xFF) — If you are writing a new vuid, you can use a segment in here.

Adding a New Segment

The central registry of virtual user input devices is `usr/include/sys/vuid_event.h`. To allocate a new vuid you must modify this file:

- Choose an unused portion of the address space. Vuids from 0x00 to 0x7F are reserved for use by Sun. Vuids from 0x80 to 0xFF are reserved for Sun customers.
- Add the new device with a `*_DEVID` `#define` in this file. Briefly describe the purpose or usage or both of the device. Mention the place where more information can be found.
- Add the new device to the `Vuid_device` enumeration with a `VUID_`*devname* entry.
- List the specific event codes in another header file that is specific to the new device. `ASCII_DEVID`, `TOP_DEVID` and `WORKSTATION_DEVID` events are listed in `vuid_event.h`.

Firm Events

A stream of firm events is what your driver is expected to emit when called through the `read` system call. This stream is a byte stream that encodes `Firm_event` structures. A firm event is a structure comprising an ID that indicates what kind of event it is, the value of the event, and a time when this event occurred; it also carries some information that allows the event's eventual consumer to maintain the complete state of its input system.

The Firm_event Structure

The firm_event structure is defined in `usr/include/sys/vuid_event.h`:

```
typedef struct firm_event {
    u_short        id;
    u_char         pair_type;
    u_char         pair;
    int            value;
    struct timeval time;
} Firm_event;

#define FE_PAIR_NONE      0
#define FE_PAIR_SET       1
#define FE_PAIR_DELTA     2
#define FE_PAIR_ABSOLUTE  3
```

`id` — is the event's unique identifier. It is either the id of an existing vuid event (if you are trying to emulate part of the vuid) or one you created.

`value` — is the event's value. It is often 0 (up) or 1 (down). For valuators it is a 32-bit integer.

`time` — is the event's timestamp of when the event occurred. The timestamp is not defined to be meaningful except to compare with other `Firm_event` time stamps. In the kernel, a call to `uniqtime`, which takes a pointer to a `struct timeval`, gets you a close-to-current unique time. In user processes, a call to `gettimeofday`(2) gets time from the same source (but it is not guaranteed to be unique).

Pairs

The `pair_type` and `pair` fields enable a consumer of events to maintain input state in an event-independent way. The `pair` field is critical for an input state maintenance package—one that is designed to know about the semantics of particular events, to maintain correct data for corresponding absolute, delta, and paired-state variables. Some examples help make this clear:

- You have a tablet emitting absolute locations. Depending on the client, the absolute location is important (for digitizing) or the difference between the current location and the previous location is important (for computing acceleration while tracking a cursor).
- You have a keyboard in which the user has typed **^c**. Your driver first emits a SHIFT_CTRL event as the control key goes down. Next your driver emits a ^C event (one of the events from the ASCII vuid segment) as the “c” key goes down. Now the application that your are driving happens to be using the “c” key as a shift key in some specialized application.

The vuid supports a notion of updating a companion event at the same time that a single event is generated. In the first situation, your want your tablet to update companion absolute and relative event values with a single event. In the second situation, you want your keyboard to update companion ^C and “c” event values with a single event. The vuid supports this notion of updating a companion event in such a way as to be independent from these two particular cases. `pair_type` defines the type of companion event:

`FE_PAIR_NONE` — is the common case in which `pair` is not defined, that is, there is no companion.

`FE_PAIR_SET` — is used for ASCII controlled events in which `pair` is the uncontrolled *base* event, that is, ^C and “c” or “C”, depending on the state of the shift key. The use of this pair type is not restricted to ASCII situations. This pair type simply says to set the *pair*th event in `id`’s vuid segment to `value`.

`FE_PAIR_DELTA` — identifies `pair` as the delta companion to `id`. This means that the `pair`th event in `id`’s vuid segment is set to the delta of `id`’s current value and `value`. Always create vuid valuator events as delta/absolute pairs. For example, the events `LOC_X_DELTA` and `LOC_X_ABSOLUTE` are pairs and the events `LOC_Y_DELTA` and `LOC_Y_ABSOLUTE` are pairs.

FE_PAIR_ABSOLUTE — identifies pair as the absolute companion to id. This means that the pairth event in id's vuid segment is set to the sum of id's current value and value. Always create vuid valuator events as delta/absolute pairs.

As indicated, pair must be in the same vuid segment as id.

Device Controls

A vuid driver responds to a variety of device controls.

Output Mode

It is more common to start from an existing device driver that already speaks its own native protocol and flush this old protocol in favor of the vuid protocol. In this case, you might want to operate in both modes. VUID*FORMAT ioctls are used to control which byte stream format an input device emits.

VUIDSFORMAT sets the input device byte stream format to one of:

- VUID_NATIVE — the device's native byte stream format (it could be vuid).
- VUID_FIRM_EVENT — the byte stream format is Firm_events.

  An errno of ENOTTY or EINVAL indicates that a device cannot speak Firm_events.

VUIDGFORMAT gets the input device byte stream format.

C

Dynamically Loadable Extensions D

X extensions must meet the following criteria to be dynamically loadable by the server:

- The extension must be decoupled from the DIX and DDX layers of the server. This means that the extension must not require any server code changes to the DIX or DDX code. Implement all extensions with X11R5 wrappers around DDX vectors.
- The extension must not depend on any resource `devPrivates`. An exception is the `Screen devPrivates`, which can be dynamically reallocated, unlike other resource `devPrivates` (such as Window and GC) that can only be allocated before any resources are instantiated.

Follow these steps to make an X extension meet these criteria:

1. Compile and link the extension as a shared object.

```
example% cc -K PIC ... *.c
example% ld -G -z text *.o ... -o ACMExtn.so.1
```

For x86 systems – On some SunPro development system releases, `-z text` flags errors against non-relocatable sections in instances where no problems exist. In general, you can build the shared object without the -z text flag.

2. **Create an entry for the extension in the `OWconfig` file.**
   See Appendix A, “The OWconfig File” and Appendix B, “Packaging and Installation Hints” for information on adding this entry.

3. **Install the shared object into the modules directory.**
   The server searches the following path for extension modules listed in the `OWconfig` file:
   `/etc/openwin/server/modules:/usr/openwin/server/modules`.
   See Appendix B, “Packaging and Installation Hints” for more information.

4. **Start the server and verify if the extension is listed with `xdpyinfo`.**
   `XListExtensions` lists the extension as available if an entry in the `OWconfig` file exists, without actually forcing the extension to be loaded.

5. **Invoke `XQueryExtension` or make an extension request to verify that the extension actually gets dynamically loaded.**

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Z