Definition of the Porting Layer for the X v11 Sample Server

Susan Angebranndt

Raymond Drewry

Philip Karlton

Todd Newman

Digital Equipment Corporation

minor revisions by

Bob Scheifler

Massachusetts Institute of Technology

Revised for Release 4 and Release 5 by

Keith Packard

MIT X Consortium

Revised for Release 6 by

David P. Wiggins

X Consortium

The following document explains the structure of the X Window System display server and the interfaces among the larger pieces. It is intended as a reference for programmers who are implementing an X Display Server on their workstation hardware. It is included with the X Window System source tape, along with the document "Strategies for Porting the X v11 Sample Server." The order in which you should read these documents is:

1) Read the first section of the "Strategies for Porting" document (Overview of Porting Process).

2) Skim over this document (the Definition document).

3) Skim over the remainder of the Strategies document.

4) Start planning and working, referring to the Strategies and Definition documents.

You may also want to look at the following documents:

- "The X Window System" for an overview of X.
- "Xlib - C Language X Interface" for a view of what the client programmer sees.
- "X Window System Protocol" for a terse description of the byte stream protocol between the client and server.

LK201 and DEC are trademarks of Digital Equipment Corporation. Macintosh and Apple are trademarks of Apple Computer, Inc. PostScript is a trademark of Adobe Systems, Inc. Ethernet is a trademark of Xerox Corporation. X Window System is a trademark of X Consortium, Inc. Cray is a trademark of Cray Research, Inc.

To understand this document and the accompanying source code, you should know the C language. You should be familiar with 2D graphics and windowing concepts such as clipping, bitmaps, fonts, etc. You should have a general knowledge of the X Window System. To implement the server code on your hardware, you need to know a lot about your hardware, its graphic display device(s), and (possibly) its networking and multitasking facilities.

This document depends a lot on the source code, so you should have a listing of the code handy.

Some source on the distribution tape is directly compilable on your machine. Some of it will require modification. Other parts may have to be completely written from scratch.

The tape also includes source for a sample implementation of a display server which runs on a variety of color and monochrome displays which you will find useful for implementing any type of X server.

1. The X Window System

The X Window System, or simply "X," is a windowing system that provides high-performance, high-level, device-independent graphics.

X is a windowing system designed for bitmapped graphic displays. The display can have a simple, monochrome display or it can have a color display with up to 32 bits per pixel with a special graphics processor doing the work. (In this document, monochrome means a black and white display with one bit per pixel. Even though the usual meaning of monochrome is more general, this special case is so common that we decided to reserve the word for this purpose.)

X is designed for a networking environment where users can run applications on machines other than their own workstations. Sometimes, the connection is over an Ethernet network with a protocol such as TCP/IP; but, any "reliable" byte stream is allowable. A high-bandwidth byte stream is preferable; RS-232 at 9600 baud would be slow without compression techniques.

X by itself allows great freedom of design. For instance, it does not include any user interface standard. Its intent is to "provide mechanism, not policy." By making it general, it can be the foundation for a wide variety of interactive software.

For a more detailed overview, see the document "The X Window System." For details on the byte stream protocol, see "X Window System protocol."

2. OVERVIEW OF THE SERVER

The display server manages windows and simple graphics requests for the user on behalf of different client applications. The client applications can be running on any machine on the network. The server mainly does three things:

- Responds to protocol requests from existing clients (mostly graphic and text drawing commands)
- Sends device input (keystrokes and mouse actions) and other events to existing clients
- Maintains client connections

The server code is organized into four major pieces:

- Device Independent (DIX) layer - code shared among all implementations
- Operating System (OS) layer - code that is different for each operating system but is shared among all graphic devices for this operating system
- Device Dependent (DDX) layer - code that is (potentially) different for each combination of operating system and graphic device
- Extension Interface - a standard way to add features to the X server

The "porting layer" consists of the OS and DDX layers; these are actually parallel and neither one is on top of the other. The DIX layer is intended to be portable without change to target systems and is not detailed here, although several routines in DIX that are called by DDX are documented. Extensions incorporate new functionality into the server; and require additional functionality over a simple DDX.

The following sections outline the functions of the layers. Section 3 briefly tells what you need to know about the DIX layer. The OS layer is explained in Section 4. Section 5 gives the theory of operation and procedural interface for the DDX layer. Section 6 describes the functions which exist for the extension writer.

2.1. Notes On Resources and Large Structs

X resources are C structs inside the server. Client applications create and manipulate these objects according to the rules of the X byte stream protocol. Client applications refer to resources with resource IDs, which are 32-bit integers that are sent over the network. Within the server, of course, they are just C structs, and we refer to them by pointers.

The DDX layer has several kinds of resources:

- Window
- Pixmap
- Screen
- Device

- Colormap
- Font
- Cursor
- Graphics Contexts

The type names of the more important server structs usually end in "Rec," such as "DeviceRec;" the pointer types usually end in "Ptr," such as "DevicePtr."

The structs and important defined constants are declared in .h files that have names that suggest the name of the object. For instance, there are two .h files for windows, window.h and windowstr.h. window.h defines only what needs to be defined in order to use windows without peeking inside of them; windowstr.h defines the structs with all of their components in great detail for those who need it.

Three kinds of fields are in these structs:

- Attribute fields - struct fields that contain values like normal structs
- Pointers to procedures, or structures of procedures, that operate on the object
- A private field (or two) used by your DDX code to keep private data (probably a pointer to another data structure), or an array of private fields, which is sized as the server initializes.

DIX calls through the struct's procedure pointers to do its tasks. These procedures are set either directly or indirectly by DDX procedures. Most of the procedures described in the remainder of this document are accessed through one of these structs. For example, the procedure to create a pixmap is attached to a ScreenRec and might be called by using the expression

```
(* pScreen->CreatePixmap)(pScreen, width, height, depth).
```

All procedure pointers must be set to some routine unless noted otherwise; a null pointer will have unfortunate consequences.

Procedure routines will be indicated in the documentation by this convention:

```
void pScreen->MyScreenRoutine(arg, arg, ...)
```

as opposed to a free routine, not in a data structure:

```
void MyFreeRoutine(arg, arg, ...)
```

The attribute fields are mostly set by DIX; DDX should not modify them unless noted otherwise.

3. DIX LAYER

The DIX layer is the machine and device independent part of X. The source should be common to all operating systems and devices. The port process should not include changes to this part, therefore internal interfaces to DIX modules are not discussed, except for public interfaces to the DDX and the OS layers.

In the process of getting your server to work, if you think that DIX must be modified for purposes other than bug fixes, you may be doing something wrong. Keep looking for a more compatible solution. When the next release of the X server code is available, you should be able to just drop in the new DIX code and compile it. If you change DIX, you will have to remember what changes you made and will have to change the new sources before you can update to the new version.

The heart of the DIX code is a loop called the dispatch loop. Each time the processor goes around the loop, it sends off accumulated input events from the input devices to the clients, and it processes requests from the clients. This loop is the most organized way for the server to process the asynchronous requests that it needs to process. Most of these operations are performed by OS and DDX routines that you must supply.

4. OS LAYER

This part of the source consists of a few routines that you have to rewrite for each operating system. These OS functions maintain the client connections and schedule work to be done for clients. They also provide an interface to font files, font name to file name translation, and low level memory management.

```
void OsInit()
```

OsInit initializes your OS code, performing whatever tasks need to be done. Frequently there is not much to be done. The sample server implementation is in Xserver/os/osinit.c.

4.1. Scheduling and Request Delivery

The main dispatch loop in DIX creates the illusion of multitasking between different windows, while the server is itself but a single process. The dispatch loop breaks up the work for each client into small digestible parts. Some parts are requests from a client, such as individual graphic commands. Some parts are events delivered to the client, such as keystrokes from the user. The processing of events and requests for different clients can be interleaved with one another so true multitasking is not needed in the server.

You must supply some of the pieces for proper scheduling between clients.

```
int WaitForSomething(pClientReady)
        int *pClientReady;
```

WaitForSomething is the scheduler procedure you must write that will suspend your server process until something needs to be done. This call should make the server suspend until one or more of the following occurs:

- There is an input event from the user or hardware (see SetInputCheck())
- There are requests waiting from known clients, in which case you should return a count of clients stored in pClientReady
- A new client tries to connect, in which case you should create the client and then continue waiting

Before WaitForSomething() computes the masks to pass to select, it needs to see if there is anything to do on the work queue; if so, it must call a DIX routine called ProcessWorkQueue.

```
extern WorkQueuePtr     workQueue;

if (workQueue)
        ProcessWorkQueue ();
```

If WaitForSomething() decides it is about to do something that might block (in the sample server, before it calls select()) it must call a DIX routine called BlockHandler().

```
void BlockHandler(pTimeout, pReadmask)
        pointer pTimeout;
        pointer pReadmask;
```

The types of the arguments are for agreement between the OS and DDX implementations, but the pTimeout is a pointer to the information determining how long the block is allowed to last, and the pReadmask is a pointer to the information describing the descriptors that will be waited on.

In the sample server, pTimeout is a struct timeval **, and pReadmask is the address of the select() mask for reading.

The DIX BlockHandler() iterates through the Screens, for each one calling its BlockHandler. A BlockHandler is declared thus:

```
void xxxBlockHandler(nscreen, pbdata, pptv, pReadmask)
        int nscreen;
        pointer pbdata;
        struct timeval ** pptv;
        pointer pReadmask;
```

The arguments are the index of the Screen, the blockData field of the Screen, and the arguments to the DIX BlockHandler().

Immediately after WaitForSomething returns from the block, even if it didn't actually block, it must call the DIX routine WakeupHandler().

```
void WakeupHandler(result, pReadmask)
        int result;
        pointer pReadmask;
```

Once again, the types are not specified by DIX. The result is the success indicator for the thing that (may have) blocked, and the pReadmask is a mask of the descriptors that came active. In the sample server, result is the result from select(), and pReadmask is the address of the select() mask for reading.

The DIX WakeupHandler() calls each Screen's WakeupHandler. A WakeupHandler is declared thus:

```
void xxxWakeupHandler(nscreen, pbdata, err, pReadmask)
        int nscreen;
        pointer pbdata;
        unsigned long result;
        pointer pReadmask;
```

The arguments are the index of the Screen, the blockData field of the Screen, and the arguments to the DIX WakeupHandler().

In addition to the per-screen BlockHandlers, any module may register block and wakeup handlers (only together) using:

```
Bool RegisterBlockAndWakeupHandlers (blockHandler, wakeupHandler, blockData)
        BlockHandlerProcPtr    blockHandler;
        WakeupHandlerProcPtr   wakeupHandler;
        pointer blockData;
```

A FALSE return code indicates that the registration failed for lack of memory. To remove a registered Block handler at other than server reset time (when they are all removed automatically), use:

```
RemoveBlockAndWakeupHandlers (blockHandler, wakeupHandler, blockData)
        BlockHandlerProcPtr   blockHandler;
        WakeupHandlerProcPtr  wakeupHandler;
        pointer blockData;
```

All three arguments must match the values passed to RegisterBlockAndWakeupHandlers.

These registered block handlers are called after the per-screen handlers:

```
void (*BlockHandler) (blockData, pptv, pReadmask)
        pointer        blockData;
        OSTimePtr pptv;
        pointer        pReadmask;
```

Any wakeup handlers registered with RegisterBlockAndWakeupHandlers will be called before the Screen handlers:

```
void (*WakeupHandler) (blockData, err, pReadmask)
        pointer        blockData;
        int err;
        pointer pReadmask;
```

The WaitForSomething on the sample server also has a built in screen saver that darkens the screen if no input happens for a period of time. The sample server implementation is in Xserver/os/WaitFor.c.

Note that WaitForSomething() may be called when you already have several outstanding things (events, requests, or new clients) queued up. For instance, your server may have just done a large graphics request, and it may have been a long time since WaitForSomething() was last called. If many clients have lots of requests queued up, DIX will only service some of them for a given client before going on to the next client (see isItTimeToYield, below). Therefore, WaitForSomething() will have to report that these same clients still have requests queued up the next time around.

An implementation should return information on as many outstanding things as it can. For instance, if your implementation always checks for client data first and does not report any input events until there is no client data left, your mouse and keyboard might get locked out by an application that constantly barrages the server with graphics drawing requests.

A list of indexes (client->index) for clients with data ready to be read or processed should be returned in pClientReady, and the count of indexes returned as the result value of the call. These are not clients that have full requests ready, but any clients who have any data ready to be read or processed. The DIX dispatcher will process requests from each client in turn by calling ReadRequestFromClient(), below.

WaitForSomething() must create new clients as they are requested (by whatever mechanism at the transport level). A new client is created by calling the DIX routine:

```
ClientPtr NextAvailableClient(ospriv)
        pointer ospriv;
```

This routine returns NULL if a new client cannot be allocated (e.g. maximum number of clients reached). The ospriv argument will be stored into the OS private field (pClient->osPrivate), to store OS private information about the client. In the sample server, the osPrivate field contains the number of the socket for this client. See also "New Client Connections." NextAvailableClient() will call InsertFakeRequest(), so you must be prepared for this.

If there are outstanding input events, you should make sure that the two SetInputCheck() locations are unequal. The DIX dispatcher will call your implementation of ProcessInputEvents() until the SetInputCheck() locations are equal.

The sample server contains an implementation of WaitForSomething(). The following two routines indicate to WaitForSomething() what devices should be waited for. fd is an OS dependent type; in the sample server it is an open file descriptor.

```
int AddEnabledDevice(fd)
        int fd;

int RemoveEnabledDevice(fd)
        int fd;
```

These two routines are usually called by DDX from the initialize cases of the Input Procedures that are stored in the DeviceRec (the routine passed to AddInputDevice()). The sample server implementation of AddEnabledDevice and RemoveEnabledDevice are in Xserver/os/connection.c.

4.2. New Client Connections

The process whereby a new client-server connection starts up is very dependent upon what your byte stream mechanism. This section describes byte stream initiation using examples from the TCP/IP implementation on the sample server.

The first thing that happens is a client initiates a connection with the server. How a client knows to do this depends upon your network facilities and the Xlib implementation. In a typical scenario, a user named Fred on his X workstation is logged onto a Cray supercomputer running a command shell in an X window. Fred can type shell commands and have the Cray respond as though the X server were a dumb terminal. Fred types in a command to run an X client application that was linked with Xlib. Xlib looks at the shell environment variable DISPLAY, which has the value "fredsbittube:0.0." The host name of Fred's workstation is "fredsbittube," and the 0s are for multiple screens and multiple X server processes. (Precisely what happens on your system depends upon how X and Xlib are implemented.)

The client application calls a TCP routine on the Cray to open a TCP connection for X to communicate with the network node "fredsbittube." The TCP software on the Cray does this by looking up the TCP address of "fredsbittube" and sending an open request to TCP port 6000 on fredsbittube.

All X servers on TCP listen for new clients on port 6000 by default; this is known as a "well-known port" in IP terminology.

The server receives this request from its port 6000 and checks where it came from to see if it is on the server's list of "trustworthy" hosts to talk to. Then, it opens another port for communications with the client. This is the byte stream that all X communications will go over.

Actually, it is a bit more complicated than that. Each X server process running on the host machine is called a "display." Each display can have more than one screen that it manages. "corporatehydra:3.2" represents screen 2 on display 3 on the multi-screened network node corporatehydra. The open request would be sent on well-known port number 6003.

Once the byte stream is set up, what goes on does not depend very much upon whether or not it is TCP. The client sends an xConnClientPrefix struct (see Xproto.h) that has the version numbers for the version of Xlib it is running, some byte-ordering information, and two character strings used for authorization. If the server does not like the authorization strings or the version numbers do not match within the rules, or if anything else is wrong, it sends a failure response with a reason string.

If the information never comes, or comes much too slowly, the connection should be broken off. You must implement the connection timeout. The sample server implements this by keeping a timestamp for each still-connecting client and, each time just before it attempts to accept new connections, it closes any connection that are too old. The connection timeout can be set from the command line.

You must implement whatever authorization schemes you want to support. The sample server on the distribution tape supports a simple authorization scheme. The only interface seen by DIX is:

```
char *
ClientAuthorized(client, proto_n, auth_proto, string_n, auth_string)
    ClientPtr client;
    unsigned int proto_n;
    char *auth_proto;
    unsigned int string_n;
    char *auth_string;
```

DIX will only call this once per client, once it has read the full initial connection data from the client. If the connection should be accepted ClientAuthorized() should return NULL, and otherwise should return an error message string.

Accepting new connections happens internally to WaitForSomething(). WaitForSomething() must call the DIX routine NextAvailableClient() to create a client object. Processing of the initial connection data will be handled by DIX. Your OS layer must be able to map from a client to whatever information your OS code needs to communicate on the given byte stream to the client. DIX uses this ClientPtr to refer to the client from now on. The sample server uses the osPrivate field in the ClientPtr to store the file descriptor for the socket, the input and output buffers, and authorization information.

To initialize the methods you choose to allow clients to connect to your server, main() calls the routine

```
void CreateWellKnownSockets()
```

This routine is called only once, and not called when the server is reset. To recreate any sockets during server resets, the following routine is called from the main loop:

```
void ResetWellKnownSockets()
```

Sample implementations of both of these routines are found in Xserver/os/connection.c.

For more details, see the section called "Connection Setup" in the X protocol specification.

4.3. Reading Data from Clients

Requests from the client are read in as a byte stream by the OS layer. They may be in the form of several blocks of bytes delivered in sequence; requests may be broken up over block boundaries or there may be many requests per block. Each request carries with it length information. It is the responsibility of the following routine to break it up into request blocks.

```
int ReadRequestFromClient(who)
        ClientPtr who;
```

You must write the routine ReadRequestFromClient() to get one request from the byte stream belonging to client "who." You must swap the third and fourth bytes (the second 16-bit word) according to the byte-swap rules of the protocol to determine the length of the request. This length is measured in 32-bit words, not in bytes. Therefore, the theoretical maximum request is 256K. (However, the maximum length allowed is dependent upon the server's input buffer. This size is sent to the client upon connection. The maximum size is the constant MAX_REQUEST_SIZE in Xserver/include/os.h) The rest of the request you return is assumed NOT to be correctly swapped for internal use, because that is the responsibility of DIX.

The 'who' argument is the ClientPtr returned from WaitForSomething. The return value indicating status should be set to the (positive) byte count if the read is successful, 0 if the read was blocked, or a negative error code if an error happened.

You must then store a pointer to the bytes of the request in the client request buffer field; who->requestBuffer. This can simply be a pointer into your buffer; DIX may modify it in place but will not otherwise cause damage. Of course, the request must be contiguous; you must shuffle it around in your buffers if not.

The sample server implementation is in Xserver/os/io.c.

DIX can insert data into the client stream, and can cause a "replay" of the current request.

```
Bool InsertFakeRequest(client, data, count)
    ClientPtr client;
    char *data;
    int count;

int ResetCurrentRequest(client)
    ClientPtr client;
```

InsertFakeRequest() must insert the specified number of bytes of data into the head of the input buffer for the client. This may be a complete request, or it might be a partial request. For example, NextAvailableCient() will insert a partial request in order to read the initial connection data sent by the client. The routine returns FALSE if memory could not be allocated. ResetCurrentRequest() should "back up" the input buffer so that the currently executing request will be reexecuted. DIX may have altered some values (e.g. the overall request length), so you must recheck to see if you still have a complete request. ResetCurrentRequest() should always cause a yield (isItTimeToYield).

4.4. Sending Events, Errors And Replies To Clients

```
int WriteToClient(who, n, buf)
        ClientPtr who;
        int n;
        char *buf;
```

WriteToClient should write n bytes starting at buf to the ClientPtr "who". It returns the number of bytes written, but for simplicity, the number returned must be either the same value as the number requested, or -1, signaling an error. The sample server implementation is in Xserver/os/io.c.

```
void SendErrorToClient(client, majorCode, minorCode, resId, errorCode)
    ClientPtr client;
    unsigned int majorCode;
    unsigned int minorCode;
    XID resId;
    int errorCode;
```

SendErrorToClient can be used to send errors back to clients, although in most cases your request function should simply return the error code, having set client->errorValue to the appropriate error value to return to the client, and DIX will call this function with the correct opcodes for you.

```
void FlushAllOutput()

void FlushIfCriticalOutputPending()

void SetCriticalOutputPending()
```

These three routines may be implemented to support buffered or delayed writes to clients, but at the very least, the stubs must exist. FlushAllOutput() unconditionally flushes all output to clients; FlushIfCriticalOutputPending() flushes output only if SetCriticalOutputPending() has be called since the last time output was flushed. The sample server implementation is in Xserver/os/io.c and actually ignores requests to flush output on a per-client basis if it knows that there are requests in that client's input queue.

4.5. Font Support

In the sample server, fonts are encoded in disk files or fetched from the font server. For disk fonts, there is one file per font, with a file name like "fixed.pcf". Font server fonts are read over the network using the X Font Server Protocol. The disk directories containing disk fonts and the names of the font servers are listed together in the current "font path."

In principle, you can put all your fonts in ROM or in RAM in your server. You can put them all in one library file on disk. You could generate them on the fly from stroke descriptions. By placing the appropriate code in the Font Library, you will automatically export fonts in that format both through the X server and the Font server.

With the incorporation of font-server based fonts and the Speedo donation from Bitstream, the font interfaces have been moved into a separate library, now called the Font Library (../fonts/lib). These routines are shared between the X server and the Font server, so instead of this document specifying what you must implement, simply refer to the font library interface specification for the details. All of the interface code to the Font library is contained in dix/dixfonts.c

4.6. Memory Management

Memory management is based on functions in the C runtime library. Xalloc(), Xrealloc(), and Xfree() work just like malloc(), realloc(), and free(), except that you can pass a null pointer to Xrealloc() to have it allocate anew or pass a null pointer to Xfree() and nothing will happen. The versions in the sample server also do some checking that is useful for debugging. Consult a C runtime library reference manual for more details.

The macros ALLOCATE_LOCAL and DEALLOCATE_LOCAL are provided in Xserver/include/os.h. These are useful if your compiler supports alloca() (or some method of allocating memory from the stack); and are defined appropriately on systems which support it.

Treat memory allocation carefully in your implementation. Memory leaks can be very hard to find and are frustrating to a user. An X server could be running for days or weeks without being reset, just like a regular terminal. If you leak a few dozen k per day, that will add up and will cause problems for users that leave their workstations on.

4.7. Client Scheduling

The X server has the ability to schedule clients much like an operating system would, suspending and restarting them without regard for the state of their input buffers. This functionality allows the X server to suspend one client and continue processing requests from other clients while waiting for a long-term network activity (like loading a font) before continuing with the first client.

```
Bool isItTimeToYield;
```

isItTimeToYield is a global variable you can set if you want to tell DIX to end the client's "time slice" and start paying attention to the next client. After the current request is finished, DIX will move to the next client.

In the sample server, ReadRequestFromClient() sets isItTimeToYield after 10 requests packets in a row are read from the same client.

This scheduling algorithm can have a serious effect upon performance when two clients are drawing into their windows simultaneously. If it allows one client to run until its request queue is empty by ignoring isItTimeToYield, the client's queue may in fact never empty and other clients will be blocked out. On the other hand, if it switchs between different clients too quickly, performance may suffer due to too much switching between contexts. For example, if a graphics processor needs to be set up with drawing modes before drawing, and two different clients are drawing with different modes into two different windows, you may switch your graphics processor modes so often that performance is impacted.

See the Strategies document for heuristics on setting isItTimeToYield.

The following functions provide the ability to suspend request processing on a particular client, resuming it at some later time:

```
int IgnoreClient (who)
	ClientPtr who;

int AttendClient (who)
	ClientPtr who;
```

Ignore client is responsible for pretending that the given client doesn't exist. WaitForSomething should not return this client as ready for reading and should not return if only this client is ready. AttendClient undoes whatever IgnoreClient did, setting it up for input again.

Three functions support "process control" for X clients:

```
Bool ClientSleep (client, function, closure)
	ClientPtr	client;
	Bool		(*function)();
	pointer		closure;
```

This suspends the current client (the calling routine is responsible for making its way back to Dispatch()). No more X requests will be processed for this client until ClientWakeup is called.

```
Bool ClientSignal (client)
	ClientPtr	client;
```

This function causes a call to the (*function) parameter passed to ClientSleep to be queued on the work queue. This does not automatically "wakeup" the client, but the function called is free to do so by calling:

```
ClientWakeup (client)
	ClientPtr	client;
```

This re-enables X request processing for the specified client.

4.8. Other OS Functions

```
void
ErrorF(char *f, ...)

void
FatalError(char *f, ...)

void
Error(str)
    char *str;
```

You should write these three routines to provide for diagnostic output from the dix and ddx layers, although implementing them to produce no output will not affect the correctness of your server. ErrorF() and FatalError() take a printf() type of format specification in the first argument and an implementation-dependent number of arguments following that. Normally, the formats passed to ErrorF() and FatalError() should be terminated with a newline. Error() provides an os interface for printing out the string passed as an argument followed by a meaningful explanation of the last system error. Normally the string does not contain a newline, and it is only called by the ddx layer. In the sample implementation, Error() uses the perror() function.

After printing the message arguments, FatalError() must be implemented such that the server will call AbortDDX() to give the ddx layer a chance to reset the hardware, and then terminate the server; it must not return.

The sample server implementation for these routines is in Xserver/os/util.c.

4.9. Idiom Support

The DBE specification introduces the notion of idioms, which are groups of X requests which can be executed more efficiently when taken as a whole compared to being performed individually and sequentially. This following server internal support to allows DBE implementations, as well as other parts of the server, to do idiom processing.

```
xReqPtr PeekNextRequest(xReqPtr req, ClientPtr client, Bool readmore)
```

If req is NULL, the return value will be a pointer to the start of the complete request that follows the one currently being executed for the client. If req is not NULL, the function assumes that req is a pointer to a request in the client's request buffer, and the return value will be a pointer to the the start of the complete request that follows req. If the complete request is not available, the function returns NULL; pointers to partial requests will never be returned. If (and only if) readmore is TRUE, PeekNextRequest should try to read an additional request from the client if one is not already available in the client's request buffer. If PeekNextRequest reads more data into the request buffer, it should not move or change the existing data.

```
void SkipRequests(xReqPtr req, ClientPtr client, int numskipped)
```

The requests for the client up to and including the one specified by req will be skipped. numskipped must be the number of requests being skipped. Normal request processing will resume with the request that follows req. The caller must not have modified the contents of the request buffer in any way (e.g., by doing byte swapping in place).

Additionally, two macros in os.h operate on the xReq pointer returned by PeekNextRequest:

```
int ReqLen(xReqPtr req, ClientPtr client)
```

The value of ReqLen is the request length in bytes of the given xReq.

```
otherReqTypePtr CastxReq(xReq *req, otherReqTypePtr)
```

The value of CastxReq is the conversion of the given request pointer to an otherReqTypePtr (which should be a pointer to a protocol structure type). Only those fields which come after the length field of otherReqType may be accessed via the returned pointer.

Thus the first two fields of a request, reqType and data, can be accessed directly using the xReq * returned by PeekNextRequest. The next field, the length, can be accessed with ReqLen. Fields beyond that can be accessed with CastxReq. This complexity was necessary because of the reencoding of core protocol that can happen due to the BigRequests extension.

5. DDX LAYER

This section describes the interface between DIX and DDX. While there may be an OS-dependent driver interface between DDX and the physical device, that interface is left to the DDX implementor and is not specified here.

The DDX layer does most of its work through procedures that are pointed to by different structs. As previously described, the behavior of these resources is largely determined by these procedure pointers. Most of these routines are for graphic display on the screen or support functions thereof. The rest are for user input from input devices.

5.1. INPUT

In this document "input" refers to input from the user, such as mouse, keyboard, and bar code readers. X input devices are of several types: keyboard, pointing device, and many others. The core server has support for extension devices as described by the X Input Extension document; the interfaces used by that extension are described elsewhere. The core devices are actually implemented as two collections of devices, the mouse is a ButtonDevice, a ValuatorDevice and a PtrFeedbackDevice while the keyboard is a KeyDevice, a FocusDevice and a KbdFeedbackDevice. Each part implements a portion of the functionality of the device. This abstraction is hidden from view for core devices by DIX.

You, the DDX programmer, are responsible for some of the routines in this section. Others are DIX routines that you should call to do the things you need to do in these DDX routines. Pay attention to which is which.

5.1.1. Input Device Data Structures

DIX keeps a global directory of devices in a central data structure called InputInfo. For each device there is a device structure called a DeviceRec. DIX can locate any DeviceRec through InputInfo. In addition, it has a special pointer to identify the main pointing device and a special pointer to identify the main keyboard.

The DeviceRec (Xserver/include/input.h) is a device-independent structure that contains the state of an input device. A DevicePtr is simply a pointer to a DeviceRec.

An xEvent describes an event the server reports to a client. Defined in Xproto.h, it is a huge struct of union of structs that have fields for all kinds of events. All of the variants overlap, so that the struct is actually very small in memory.

5.1.2. Processing Events

The main DDX input interface is the following routine:

```
void ProcessInputEvents()
```

You must write this routine to deliver input events from the user. DIX calls it when input is pending (see next section), and possibly even when it is not. You should write it to get events from each device and deliver the events to DIX. To deliver the events to DIX, DDX should call the following routine:

```
void DevicePtr->processInputProc(pEvent, device, count)
        xEventPtr events;
        DeviceIntPtr device;
        int count;
```

This is the "input proc" for the device, a DIX procedure. DIX will fill in this procedure pointer to one of its own routines by the time ProcessInputEvents() is called the first time. Call this input proc routine as many times as needed to deliver as many events as should be delivered. DIX will buffer them up and send them out as needed. Count is set to the number of event records which make up one atomic device event and is always 1 for the core devices (see the X Input Extension for descriptions of devices which may use count > 1).

For example, your ProcessInputEvents() routine might check the mouse and the keyboard. If the keyboard had several keystrokes queued up, it could just call the keyboard's processInputProc as many times as needed to flush its internal queue.

event is an xEvent struct you pass to the input proc. When the input proc returns, it is finished with the event rec, and you can fill in new values and call the input proc again with it.

You should deliver the events in the same order that they were generated.

For keyboard and pointing devices the xEvent variant should be keyButtonPointer. Fill in the following fields in the xEvent record:

type	is one of the following: KeyPress, KeyRelease, ButtonPress, ButtonRelease, or MotionNotify
detail	for KeyPress or KeyRelease fields, this should be the key number (not the ASCII code); otherwise unused
time	is the time that the event happened (32-bits, in milliseconds, arbitrary origin)
rootX	is the x coordinate of cursor
rootY	is the y coordinate of cursor

The rest of the fields are filled in by DIX.

The time stamp is maintained by your code in the DDX layer, and it is your responsibility to stamp all events correctly.

The x and y coordinates of the pointing device and the time must be filled in for all event types including keyboard events.

The pointing device must report all button press and release events. In addition, it should report a MotionNotify event every time it gets called if the pointing device has moved since the last notify. Intermediate pointing device moves are stored in a special GetMotionEvents buffer, because most client programs are not interested in them.

There are quite a collection of sample implementations of this routine, one for each supported device.

5.1.3. Telling DIX When Input is Pending

In the server's dispatch loop, DIX checks to see if there is any device input pending whenever WaitForSomething() returns. If the check says that input is pending, DIX calls the DDX routine ProcessInputEvents().

This check for pending input must be very quick; a procedure call is too slow. The code that does the check is a hardwired IF statement in DIX code that simply compares the values pointed to by two pointers. If the values are different, then it assumes that input is pending and ProcessInputEvents() is called by DIX.

You must pass pointers to DIX to tell it what values to compare. The following procedure is used to set these pointers:

```
void SetInputCheck(p1, p2)
        long *p1, *p2;
```

You should call it sometime during initialization to indicate to DIX the correct locations to check. You should pay special attention to the size of what they actually point to, because the locations are assumed to be longs.

These two pointers are initialized by DIX to point to arbitrary values that are different. In other words, if you forget to call this routine during initialization, the worst thing that will happen is that ProcessInputEvents will be called when there are no events to process.

p1 and p2 might point at the head and tail of some shared memory queue. Another use would be to have one point at a constant 0, with the other pointing at some mask containing 1s for each input device that has something pending.

The DDX layer of the sample server calls SetInputCheck() once when the server's private internal queue is initialized. It passes pointers to the queue's head and tail. See Xserver/mi/mieq.c.

```
int TimeSinceLastInputEvent()
```

DDX must time stamp all hardware input events. But DIX sometimes needs to know the time and the OS layer needs to know the time since the last hardware input event in order for the screen saver to work. TimeSinceLastInputEvent() returns the this time in milliseconds.

5.1.4. Controlling Input Devices

You must write four routines to do various device-specific things with the keyboard and pointing device. They can have any name you wish because you pass the procedure pointers to DIX routines.

```
int pInternalDevice->valuator->GetMotionProc(pdevice, coords, start, stop, pScreen)
        DeviceIntPtr pdevice;
        xTimecoord * coords;
        unsigned long start;
        unsigned long stop;
        ScreenPtr pScreen;
```

You write this DDX routine to fill in coords with all the motion events that have times (32-bit count of milliseconds) between time start and time stop. It should return the number of motion events returned. If there is no motion events support, this routine should do nothing and return zero. The maximum number of coords to return is set in InitPointerDeviceStruct(), below.

When the user drags the pointing device, the cursor position theoretically sweeps through an infinite number of points. Normally, a client that is concerned with points other than the starting and ending points will receive a pointer-move event only as often as the server generates them. (Move events do not queue up; each new one replaces the last in the queue.) A server, if desired, can implement a scheme to save these intermediate events in a motion buffer. A client application, like a paint program, may then request that these events be delivered to it through the GetMotionProc routine.

```
void pInternalDevice->bell->BellProc(percent, pDevice, ctrl, unknown)
        int percent;
        DeviceIntPtr pDevice;
        pointer ctrl;
        int class;
```

You need to write this routine to ring the bell on the keyboard. loud is a number from 0 to 100, with 100 being the loudest. Class is either BellFeedbackClass or KbdFeedbackClass (from XI.h).

```
void pInternalDevice->somedevice->CtrlProc(device, ctrl)
        DevicePtr device;
        SomethingCtrl *ctrl;
```

You write two versions of this procedure, one for the keyboard and one for the pointing device. DIX calls it to inform DDX when a client has requested changes in the current settings for the particular device. For a keyboard, this might be the repeat threshold and rate. For a pointing device, this might be a scaling factor (coarse or fine) for position reporting. See input.h for the ctrl structures.

5.1.5. Input Initialization

Input initialization is a bit complicated. It all starts with InitInput(), a routine that you write to call AddInputDevice() twice (once for pointing device and once for keyboard.) You also want to call RegisterKeyboardDevice() and RegisterPointerDevice() on them.

When you Add the devices, a routine you supply for each device gets called to initialize them. Your individual initialize routines must call InitKeyboardDeviceStruct() or InitPointerDeviceStruct(), depending upon which it is. In other words, you indicate twice that the keyboard is the keyboard and the pointer is the pointer.

```
void InitInput(argc, argv)
   int argc;
   char **argv;
```

InitInput is a DDX routine you must write to initialize the input subsystem in DDX. It must call AddInputDevice() for each device that might generate events. In addition, you must register the main keyboard and pointing devices by calling RegisterPointerDevice() and RegisterKeyboardDevice().

```
DevicePtr AddInputDevice(deviceProc, autoStart)
        DeviceProc deviceProc;
        Bool autoStart;
```

AddInputDevice is a DIX routine you call to create a device object. deviceProc is a DDX routine that is called by DIX to do various operations. AutoStart should be TRUE for devices that need to be turned on at initialization time with a special call, as opposed to waiting for some client application to turn them on. This routine returns NULL if sufficient memory cannot be allocated to install the device.

Note also that except for the main keyboard and pointing device, an extension is needed to provide for a client interface to a device.

```
void RegisterPointerDevice(device)
        DevicePtr device;
```

RegisterPointerDevice is a DIX routine that your DDX code calls that makes that device the main pointing device. This routine is called once upon initialization and cannot be called again.

```
void RegisterKeyboardDevice(device)
        DevicePtr device;
```

RegisterKeyboardDevice makes the given device the main keyboard. This routine is called once upon initialization and cannot be called again.

The following DIX procedures return the specified DevicePtr. They may or may not be useful to DDX implementors.

```
DevicePtr LookupKeyboardDevice()
```

LookupKeyboardDevice returns pointer for current main keyboard device.

```
DevicePtr LookupPointerDevice()
```

LookupPointerDevice returns pointer for current main pointing device.

A DeviceProc (the kind passed to AddInputDevice()) in the following form:

```
Bool pInternalDevice->DeviceProc(device, action);
	DeviceIntPtr device;
	int action;
```

You must write a DeviceProc for each device. device points to the device record. action tells what action to take; it will be one of these defined constants (defined in input.h):

- DEVICE_INIT - At DEVICE_INIT time, the device should initialize itself by calling InitPointerDeviceStruct(), InitKeyboardDeviceStruct(), or a similar routine (see below) and "opening" the device if necessary. If you return a non-zero (i.e., != Success) value from the DEVICE_INIT call, that device will be considered unavailable. If either the main keyboard or main pointing device cannot be initialized, the DIX code will refuse to continue booting up.
- DEVICE_ON - If the DeviceProc is called with DEVICE_ON, then it is allowed to start putting events into the client stream by calling through the ProcessInputProc in the device.
- DEVICE_OFF - If the DeviceProc is called with DEVICE_OFF, no further events from that device should be given to the DIX layer. The device will appear to be dead to the user.
- DEVICE_CLOSE - At DEVICE_CLOSE (terminate or reset) time, the device should be totally closed down.

```
void InitPointerDeviceStruct(device, map, mapLength,
		GetMotionEvents, ControlProc, numMotionEvents)
	DevicePtr device;
	CARD8 *map;
	int mapLength;
	ValuatorMotionProcPtr ControlProc;
	PtrCtrlProcPtr GetMotionEvents;
	int numMotionEvents;
```

InitPointerDeviceStruct is a DIX routine you call at DEVICE_INIT time to declare some operating routines and data structures for a pointing device. map and mapLength are as described in the X Window System protocol specification. ControlProc and GetMotionEvents are DDX routines, see above.

numMotionEvents is for the motion-buffer-size for the GetMotionEvents request. A typical length for a motion buffer would be 100 events. A server that does not implement this capability should set numMotionEvents to zero.

```
void InitKeyboardDeviceStruct(device, pKeySyms, pModifiers, Bell, ControlProc)
	DevicePtr device;
	KeySymsPtr pKeySyms;
	CARD8 *pModifiers;
	BellProcPtr Bell;
	KbdCtrlProcPtr ControlProc;
```

You call this DIX routine when a keyboard device is initialized and its device procedure is called with DEVICE_INIT. The formats of the keysyms and modifier maps are defined in Xserver/include/input.h. They describe the layout of keys on the keyboards, and the glyphs associated with them. (See the next section for information on setting up the modifier map and the keysym map.) ControlProc and Bell are DDX routines, see above.

5.1.6. Keyboard Mapping and Keycodes

When you send a keyboard event, you send a report that a given key has either been pressed or has been released. There must be a keycode for each key that identifies the key; the keycode-to-key mapping can be any mapping you desire, because you specify the mapping in a table you set up for DIX. However, you are restricted by the protocol specification to keycode values in the range 8 to 255 inclusive.

The keycode mapping information that you set up consists of the following:

- A minimum and maximum keycode number
- An array of sets of keysyms for each key, that is of length maxkeycode - minkeycode + 1. Each element of this array is a list of codes for symbols that are on that key. There is no limit to the number of symbols that can be on a key.

Once the map is set up, DIX keeps and maintains the client's changes to it.

The X protocol defines standard names to indicate the symbol(s) printed on each keycap. (See X11/keysym.h)

Legal modifier keys must generate both up and down transitions. When a client tries to change a modifier key (for instance, to make "A" the "Control" key), DIX calls the following routine, which should retuurn TRUE if the key can be used as a modifier on the given device:

```
Bool LegalModifier(key, pDev)
   unsigned int key;
   DevicePtr pDev;
```

5.2. Screens

Different computer graphics displays have different capabilities. Some are simple monochrome frame buffers that are just lying there in memory, waiting to be written into. Others are color displays with many bits per pixel using some color lookup table. Still others have high-speed graphic processors that prefer to do all of the work themselves, including maintaining their own high-level, graphic data structures.

5.2.1. Screen Hardware Requirements

The only requirement on screens is that you be able to both read and write locations in the frame buffer. All screens must have a depth of 32 or less (unless you use an X extension to allow a greater depth). All screens must fit into one of the classes listed in the section in this document on Visuals and Depths.

X uses the pixel as its fundamental unit of distance on the screen. Therefore, most programs will measure everything in pixels.

The sample server assumes square pixels. Serious WYSIWYG (what you see is what you get) applications for publishing and drawing programs will adjust for different screen resolutions automatically. Considerable work is involved in compensating for non-square pixels (a bit in the DDX code for the sample server but quite a bit in the client applications).

5.2.2. Data Structures

X supports multiple screens that are connected to the same server. Therefore, all the per-screen information is bundled into one data structure of attributes and procedures, which is the ScreenRec (see Xserver/include/scrnintstr.h). The procedure entry points in a ScreenRec operate on regions, colormaps, cursors, and fonts, because these resources can differ in format from one screen to another.

Windows are areas on the screen that can be drawn into by graphic routines. "Pixmaps" are off-screen graphic areas that can be drawn into. They are both considered drawables and are described in the section on Drawables. All graphic operations work on drawables, and operations are available to copy patches from one drawable to another.

The pixel image data in all drawables is in a format that is private to DDX. In fact, each instance of a drawable is associated with a given screen. Presumably, the pixel image data for pixmaps is chosen to be conveniently understood by the hardware. All screens in a single server must be able to handle all pixmaps depths declared in the connection setup information.

Pixmap images are transferred to the server in one of two ways: XYPixmap or ZPimap. XYPixmaps are a series of bitmaps, one for each bit plane of the image, using the bitmap padding rules from the connection setup. ZPixmaps are a series of bits, nibbles, bytes or words, one for each pixel, using the format rules (padding and so on) for the appropriate depth.

All screens in a given server must agree on a set of pixmap image formats (PixmapFormat) to support (depth, number of bits per pixel, etc.).

There is no color interpretation of bits in the pixmap. Pixmaps do not contain pixel values. The interpretation is made only when the bits are transferred onto the screen.

The screenInfo structure (in scrnintstr.h) is a global data structure that has a pointer to an array of ScreenRecs, one for each screen on the server. (These constitute the one and only description of each screen in the server.) Each screen has an identifying index (0, 1, 2, ...). In addition, the screenInfo struct contains global server-wide details, such as the bit- and byte- order in all bit images, and the list of pixmap image formats that are supported. The X protocol insists that these must be the same for all screens on the server.

5.2.3. Output Initialization

```
InitOutput(pScreenInfo, argc, argv)
    ScreenInfo *pScreenInfo;
    int argc;
    char **argv;
```

Upon initialization, your DDX routine InitOutput() is called by DIX. It is passed a pointer to screenInfo to initialize. It is also passed the argc and argv from main() for your server for the command-line arguments. These arguments may indicate what or how many screen device(s) to use or in what way to use them. For instance, your server command line may allow a "-D" flag followed by the name of the screen device to use.

Your InitOutput() routine should initialize each screen you wish to use by calling AddScreen(), and then it should initialize the pixmap formats that you support by storing values directly into the screenInfo data structure. You should also set certain implementation-dependent numbers and procedures in your screenInfo, which determines the pixmap and scanline padding rules for all screens in the server.

```
int AddScreen(scrInitProc, argc, argv)
    Bool (*scrInitProc)();
    int argc;
    char **argv;
```

You should call AddScreen(), a DIX procedure, in InitOutput() once for each screen to add it to the screenInfo database. The first argument is an initialization procedure for the screen that you supply. The second and third are the argc and argv from main(). It returns the screen number of the screen installed, or -1 if there is either insufficient memory to add the screen, or (*scrInitProc) returned FALSE.

The scrInitProc should be of the following form:

```
Bool scrInitProc(iScreen, pScreen, argc, argv)
    int iScreen;
```

```
        ScreenPtr pScreen;
        int argc;
        char **argv;
```

iScreen is the index for this screen; 0 for the first one initialized, 1 for the second, etc. pScreen is the pointer to the screen's new ScreenRec. argc and argv are as before. Your screen initialize procedure should return TRUE upon success or FALSE if the screen cannot be initialized (for instance, if the screen hardware does not exist on this machine).

This procedure must determine what actual device it is supposed to initialize. If you have a different procedure for each screen, then it is no problem. If you have the same procedure for multiple screens, it may have trouble figuring out which screen to initialize each time around, especially if InitOutput() does not initialize all of the screens. It is probably easiest to have one procedure for each screen.

The initialization procedure should fill in all the screen procedures for that screen (windowing functions, region functions, etc.) and certain screen attributes for that screen.

5.2.4. Region Routines in the ScreenRec

A region is a dynamically allocated data structure that describes an irregularly shaped piece of real estate in XY pixel space. You can think of it as a set of pixels on the screen to be operated upon with set operations such as AND and OR.

A region is frequently implemented as a list of rectangles or bitmaps that enclose the selected pixels. Region operators control the "clipping policy," or the operations that work on regions. (The sample server uses YX-banded rectangles. Unless you have something already implemented for your graphics system, you should keep that implementation.) The procedure pointers to the region operators are located in the ScreenRec data structure. The definition of a region can be found in the file Xserver/include/regionstr.h. The region code is found in Xserver/mi/miregion.c. DDX implementations using other region formats will need to supply different versions of the region operators.

Since the list of rectangles is unbounded in size, part of the region data structure is usually a large, dynamically allocated chunk of memory. As your region operators calculate logical combinations of regions, these blocks may need to be reallocated by your region software. For instance, in the sample server, a RegionRec has some header information and a pointer to a dynamically allocated rectangle list. Periodically, the rectangle list needs to be expanded with Xrealloc(), whereupon the new pointer is remembered in the RegionRec.

Most of the region operations come in two forms: a function pointer in the Screen structure, and a macro. The server can be compiled so that the macros make direct calls to the appropriate functions (instead of indirecting through a screen function pointer), or it can be compiled so that the macros are identical to the function pointer forms. Making direct calls is faster on many architectures.

```
    RegionPtr pScreen->RegionCreate( rect, size)
        BoxPtr rect;
        int size;

    macro: RegionPtr REGION_CREATE(pScreen, rect, size)
```

RegionCreate creates a region that describes ONE rectangle. The caller can avoid unnecessary reallocation and copying by declaring the probable maximum number of rectangles that this region will need to describe itself. Your region routines, though, cannot fail just because the region grows beyond this size. The caller of this routine can pass almost anything as the size; the value is merely a good guess as to the maximum size until it is proven wrong by subsequent use. Your region procedures are then on their own in estimating how big the region will get. Your implementation might ignore size, if applicable.

void pScreen->RegionInit (pRegion, rect, size)
RegionPtr pRegion;
BoxPtr rect;
int size;

macro: REGION_INIT(pScreen, pRegion, rect, size)

Given an existing raw region structure (such as an local variable), this routine fills in the appropriate fields to make this region as usable as one returned from RegionCreate. This avoids the additional dynamic memory allocation overhead for the region structure itself.

Bool pScreen->RegionCopy(dstrgn, srcrgn)
RegionPtr dstrgn, srcrgn;

macro: Bool REGION_COPY(pScreen, dstrgn, srcrgn)

RegionCopy copies the description of one region, srcrgn, to another already-created region, dstrgn; returning TRUE if the copy succeeded, and FALSE otherwise.

void pScreen->RegionDestroy(pRegion)
RegionPtr pRegion;

macro: REGION_DESTROY(pScreen, pRegion)

RegionDestroy destroys a region and frees all allocated memory.

void pScreen->RegionUninit (pRegion)
RegionPtr pRegion;

macro: REGION_UNINIT(pScreen, pRegion)

Frees everything except the region structure itself, useful when the region was originally passed to RegionInit instead of received from RegionCreate. When this call returns, pRegion must not be reused until it has been RegionInit'ed again.

Bool pScreen->Intersect(newReg, reg1, reg2)
RegionPtr newReg, reg1, reg2;

macro: Bool REGION_INTERSECT(pScreen, newReg, reg1, reg2)

Bool pScreen->Union(newReg, reg1, reg2)
RegionPtr newReg, reg1, reg2;

macro: Bool REGION_UNION(pScreen, newReg, reg1, reg2)

Bool pScreen->Subtract(newReg, regMinuend, regSubtrahend)
RegionPtr newReg, regMinuend, regSubtrahend;

macro: Bool REGION_UNION(pScreen, newReg, regMinuend, regSubtrahend)

Bool pScreen->Inverse(newReg, pReg, pBox)
RegionPtr newReg, pReg;
BoxPtr pBox;

```
macro: Bool REGION_INVERSE(pScreen, newReg, pReg,  pBox)
```

The above four calls all do basic logical operations on regions. They set the new region (which already exists) to describe the logical intersection, union, set difference, or inverse of the region(s) that were passed in. Your routines must be able to handle a situation where the newReg is the same region as one of the other region arguments.

The subtract function removes the Subtrahend from the Minuend and puts the result in newReg.

The inverse function returns a region that is the pBox minus the region passed in. (A true "inverse" would make a region that extends to infinity in all directions but has holes in the middle.) It is undefined for situations where the region extends beyond the box.

Each routine must return the value TRUE for success.

```
void pScreen->RegionReset(pRegion, pBox)
        RegionPtr pRegion;
        BoxPtr pBox;

macro: REGION_RESET(pScreen, pRegion, pBox)
```

RegionReset sets the region to describe one rectangle and reallocates it to a size of one rectangle, if applicable.

```
void  pScreen->TranslateRegion(pRegion, x, y)
        RegionPtr pRegion;
        int x, y;

macro: REGION_TRANSLATE(pScreen, pRegion, x, y)
```

TranslateRegion simply moves a region +x in the x direction and +y in the y direction.

```
int  pScreen->RectIn(pRegion, pBox)
        RegionPtr pRegion;
        BoxPtr pBox;

macro: int RECT_IN_REGION(pScreen, pRegion, pBox)
```

RectIn returns one of the defined constants rgnIN, rgnOUT, or rgnPART, depending upon whether the box is entirely inside the region, entirely outside of the region, or partly in and partly out of the region. These constants are defined in Xserver/include/region.h.

```
Bool pScreen->PointInRegion(pRegion, x, y, pBox)
        RegionPtr pRegion;
        int x, y;
        BoxPtr pBox;

macro: Bool POINT_IN_REGION(pScreen, pRegion, x, y, pBox)
```

PointInRegion returns true if the point x, y is in the region. In addition, it fills the rectangle pBox with coordinates of a rectangle that is entirely inside of pRegion and encloses the point. In the mi implementation, it is the largest such rectangle. (Due to the sample server implementation, this comes cheaply.)

This routine used by DIX when tracking the pointing device and deciding whether to report mouse events or change the cursor. For instance, DIX needs to change the cursor when it moves from one window to another. Due to overlapping windows, the shape to check may be irregular. A PointInRegion() call for every pointing device movement may be too expensive. The pBox is a kind of wake-up box; DIX need not call PointInRegion() again until the cursor wanders outside of the returned box.

```
Bool pScreen->RegionNotEmpty(pRegion)
	RegionPtr pRegion;

macro: Bool REGION_NOTEMPTY(pScreen, pRegion)
```

RegionNotEmpty is a boolean function that returns true or false depending upon whether the region encloses any pixels.

```
void pScreen->RegionEmpty(pRegion)
	RegionPtr pRegion;

macro: REGION_EMPTY(pScreen, pRegion)
```

RegionEmpty sets the region to be empty.

```
BoxPtr pScreen->RegionExtents(pRegion)
	RegionPtr pRegion;

macro: REGION_EXTENTS(pScreen, pRegion)
```

RegionExtents returns a rectangle that is the smallest possible superset of the entire region. The caller will not modify this rectangle, so it can be the one in your region struct.

```
Bool pScreen->RegionAppend (pDstRgn, pRegion)
	RegionPtr pDstRgn;
	RegionPtr pRegion;

macro: Bool REGION_APPEND(pScreen, pDstRgn, pRegion)

Bool pScreen->RegionValidate (pRegion, pOverlap)
	RegionPtr pRegion;
	Bool *pOverlap;

macro: Bool REGION_VALIDATE(pScreen, pRegion, pOverlap)
```

These functions provide an optimization for clip list generation and must be used in conjunction. The combined effect is to produce the union of a collection of regions, by using RegionAppend several times, and finally calling RegionValidate which takes the intermediate representation (which needn't be a valid region) and produces the desired union. pOverlap is set to TRUE if any of the original regions overlap; FALSE otherwise.

```
RegionPtr pScreen->BitmapToRegion (pPixmap)
	PixmapPtr pPixmap;

macro: RegionPtr BITMAP_TO_REGION(pScreen, pPixmap)
```

Given a depth-1 pixmap, this routine must create a valid region which includes all the areas of the pixmap filled with 1's and excludes the areas filled with 0's. This routine returns NULL if out of memory.

```
RegionPtr pScreen->RectsToRegion (nrects, pRects, ordering)
        int nrects;
        xRectangle *pRects;
        int ordering;

macro: RegionPtr RECTS_TO_REGION(pScreen, nrects, pRects, ordering)
```

Given a client-supplied list of rectangles, produces a region which includes the union of all the rectangles. Ordering may be used as a hint which describes how the rectangles are sorted. As the hint is provided by a client, it must not be required to be correct, but the results when it is not correct are not defined (core dump is not an option here).

```
void pScreen->SendGraphicsExpose(client,pRegion,drawable,major,minor)
        ClientPtr client;
        RegionPtr pRegion;
        XID drawable;
        int major;
        int minor;
```

SendGraphicsExpose dispatches a list of GraphicsExposure events which span the region to the specified client. If the region is empty, or a NULL pointer, a NoExpose event is sent instead.

5.2.5. Cursor Routines for a Screen

A cursor is the visual form tied to the pointing device. The default cursor is an "X" shape, but the cursor can have any shape. When a client creates a window, it declares what shape the cursor will be when it strays into that window on the screen.

For each possible shape the cursor assumes, there is a CursorRec data structure. This data structure contains a pointer to a CursorBits data structure which contains a bitmap for the image of the cursor and a bitmap for a mask behind the cursor, in addition, the CursorRec data structure contains foreground and background colors for the cursor. The CursorBits data structure is shared among multiple CursorRec structures which use the same font and glyph to describe both source and mask. The cursor image is applied to the screen by applying the mask first, clearing 1 bits in its form to the background color, and then overwriting on the source image, in the foreground color. (One bits of the source image that fall on top of zero bits of the mask image are undefined.) This way, a cursor can have transparent parts, and opaque parts in two colors. X allows any cursor size, but some hardware cursor schemes allow a maximum of N pixels by M pixels. Therefore, you are allowed to transform the cursor to a smaller size, but be sure to include the hot-spot.

CursorBits in Xserver/include/cursorstr.h is a device-independent structure containing a device-independent representation of the bits for the source and mask. (This is possible because the bitmap representation is the same for all screens.)

When a cursor is created, it is "realized" for each screen. At realization time, each screen has the chance to convert the bits into some other representation that may be more convenient (for instance, putting the cursor into off-screen memory) and set up its device-private area in either the CursorRec data structure or CursorBits data structure as appropriate to possibly point to whatever data structures are needed. It is more memory-conservative to share realizations by using the CursorBits private field, but this makes the assumption that the realization is independent of the colors used (which is typically true). For instance, the following are the device private entries for a particular screen and cursor:

```
pCursor->devPriv[pScreen->myNum]
pCursor->bits->devPriv[pScreen->myNum]
```

This is done because the change from one cursor shape to another must be fast and responsive; the cursor image should be able to flutter as fast as the user moves it across the screen.

You must implement the following routines for your hardware:

```
Bool pScreen->RealizeCursor( pScr, pCurs)
	ScreenPtr pScr;
	CursorPtr pCurs;

Bool pScreen->UnrealizeCursor( pScr, pCurs)
	ScreenPtr pScr;
	CursorPtr pCurs;
```

RealizeCursor and UnrealizeCursor should realize (allocate and calculate all data needed) and unrealize (free the dynamically allocated data) a given cursor when DIX needs them. They are called whenever a device-independent cursor is created or destroyed. The source and mask bits pointed to by fields in pCurs are undefined for bits beyond the right edge of the cursor. This is so because the bits are in Bitmap format, which may have pad bits on the right edge. You should inhibit UnrealizeCursor() if the cursor is currently in use; this happens when the system is reset.

```
Bool pScreen->DisplayCursor( pScr, pCurs)
	ScreenPtr pScr;
	CursorPtr pCurs;
```

DisplayCursor should change the cursor on the given screen to the one passed in. It is called by DIX when the user moves the pointing device into a different window with a different cursor. The hotspot in the cursor should be aligned with the current cursor position.

```
void pScreen->RecolorCursor( pScr, pCurs, displayed)
	ScreenPtr pScr;
	CursorPtr pCurs;
	Bool displayed;
```

RecolorCursor notifies DDX that the colors in pCurs have changed and indicates whether this is the cursor currently being displayed. If it is, the cursor hardware state may have to be updated. Whether displayed or not, state created at RealizeCursor time may have to be updated. A generic version, miRecolorCursor, may be used that does an unrealize, a realize, and possibly a display (in micursor.c); however this constrains UnrealizeCursor and RealizeCursor to always return TRUE as no error indication is returned here.

```
void pScreen->ConstrainCursor( pScr, pBox)
	ScreenPtr pScr;
	BoxPtr pBox;
```

ConstrainCursor should cause the cursor to restrict its motion to the rectangle pBox. DIX code is capable of enforcing this constraint by forcefully moving the cursor if it strays out of the rectangle, but ConstrainCursor offers a way to send a hint to the driver or hardware if such support is available. This can prevent the cursor from wandering out of the box, then jumping back, as DIX forces it back.

```
void pScreen->PointerNonInterestBox( pScr, pBox)
	ScreenPtr pScr;
	BoxPtr pBox;
```

PointerNonInterestBox is DIX's way of telling the pointing device code not to report motion events while the cursor is inside a given rectangle on the given screen. It is optional and, if not implemented, it should do nothing. This routine is called only when the client has declared that it is not interested in motion events in a given window. The rectangle you get may be a subset of that window. It saves DIX code the time required to discard uninteresting mouse motion events. This is only a hint, which may speed performance. Nothing in DIX currently calls PointerNonInterestBox.

```
void pScreen->CursorLimits( pScr, pCurs, pHotBox, pTopLeftBox)
        ScreenPtr pScr;
        CursorPtr pCurs;
        BoxPtr pHotBox;
        BoxPtr pTopLeftBox;     /* return value */
```

CursorLimits should calculate the box that the cursor hot spot is physically capable of moving within, as a function of the screen pScr, the device-independent cursor pCurs, and a box that DIX hypothetically would want the hot spot confined within, pHotBox. This routine is for informing DIX only; it alters no state within DDX.

```
Bool pScreen->SetCursorPosition( pScr, newx, newy, generateEvent)
        ScreenPtr pScr;
        int newx;
        int newy;
        Bool generateEvent;
```

SetCursorPosition should artificially move the cursor as though the user had jerked the pointing device very quickly. This is called in response to the WarpPointer request from the client, and at other times. If generateEvent is True, the device should decide whether or not to call ProcessInputEvents() and then it must call DevicePtr->processInputProc. Its effects are, of course, limited in value for absolute pointing devices such as a tablet.

```
void NewCurrentScreen(newScreen, x, y)
    ScreenPtr newScreen;
    int x,y;
```

If your ddx provides some mechanism for the user to magically move the pointer between multiple screens, you need to inform DIX when this occurs. You should call NewCurrentScreen to accomplish this, specifying the new screen and the new x and y coordinates of the pointer on that screen.

5.2.6. Visuals, Depths and Pixmap Formats for Screens

The "depth" of a image is the number of bits that are used per pixel to display it.

The "bits per pixel" of a pixmap image that is sent over the client byte stream is a number that is either 4, 8, 16, 24 or 32. It is the number of bits used per pixel in Z format. For instance, a pixmap image that has a depth of six is best sent in Z format as 8 bits per pixel.

A "pixmap image format" or a "pixmap format" is a description of the format of a pixmap image as it is sent over the byte stream. For each depth available on a server, there is one and only one pixmap format. This pixmap image format gives the bits per pixel and the scanline padding unit. (For instance, are pixel rows padded to bytes, 16-bit words, or 32-bit words?)

For each screen, you must decide upon what depth(s) it supports. You should only count the number of bits used for the actual image. Some displays store additional bits to indicate what window this pixel is in, how close this object is to a viewer, transparency, and other data; do not count these bits.

A "display class" tells whether the display is monochrome or color, whether there is a lookup table, and how the lookup table works.

A "visual" is a combination of depth, display class, and a description of how the pixel values result in a color on the screen. Each visual has a set of masks and offsets that are used to separate a pixel value into its red, green, and blue components and a count of the number of colormap entries. Some of these fields are only meaningful when the class dictates so. Each visual also has a screen ID telling which screen it is usable on. Note that the depth does not imply the number of map_entries; for instance, a display can have 8 bits per pixel but only 254 colormap entries for use by applications (the other two being reserved by hardware for the cursor).

Each visual is identified by a 32-bit visual ID which the client uses to choose what visual is desired on a given window. Clients can be using more than one visual on the same screen at the same time.

The class of a display describes how this translation takes place. There are three ways to do the translation.

- Pseudo - The pixel value, as a whole, is looked up in a table of length map_entries to determine the color to display.
- True - The pixel value is broken up into red, green, and blue fields, each of which are looked up in separate red, green, and blue lookup tables, each of length map_entries.
- Gray - The pixel value is looked up in a table of length map_entries to determine a gray level to display.

In addition, the lookup table can be static (resulting colors are fixed for each pixel value) or dynamic (lookup entries are under control of the client program). This leads to a total of six classes:

- Static Gray - The pixel value (of however many bits) determines directly the level of gray that the pixel assumes.
- Gray Scale - The pixel value is fed through a lookup table to arrive at the level of gray to display for the given pixel.
- Static Color - The pixel value is fed through a fixed lookup table that yields the color to display for that pixel.
- PseudoColor - The whole pixel value is fed through a programmable lookup table that has one color (including red, green, and blue intensities) for each possible pixel value, and that color is displayed.
- True Color - Each pixel value consists of one or more bits that directly determine each primary color intensity after being fed through a fixed table.
- Direct Color - Each pixel value consists of one or more bits for each primary color. Each primary color value is individually looked up in a table for that primary color, yielding an intensity for that primary color. For each pixel, the red value is looked up in the red table, the green value in the green table, and the blue value in the blue table.

Here are some examples:

> A simple monochrome 1 bit per pixel display is Static Gray.
>
> A display that has 2 bits per pixel for a choice between the colors of black, white, green and violet is Static Color.
>
> A display that has three bits per pixel, where each bit turns on or off one of the red, green or blue guns, is in the True Color class.
>
> If you take the last example and scramble the correspondence between pixel values and colors it becomes a Static Color display.
>
> A display has 8 bits per pixel. The 8 bits select one entry out of 256 entries in a lookup table, each entry consisting of 24 bits (8bits each for red, green, and blue). The display can show any 256 of 16 million colors on the screen at once. This is a pseudocolor display. The client application gets to fill

the lookup table in this class of display.

Imagine the same hardware from the last example. Your server software allows the user, on the command line that starts up the server program, to fill the lookup table to his liking once and for all. From then on, the server software would not change the lookup table until it exits. For instance, the default might be a lookup table with a reasonable sample of colors from throughout the color space. But the user could specify that the table be filled with 256 steps of gray scale because he knew ahead of time he would be manipulating a lot of black-and-white scanned photographs and not very many color things. Clients would be presented with this unchangeable lookup table. Although the hardware qualifies as a PseudoColor display, the facade presented to the X client is that this is a Static Color display.

You have to decide what kind of display you have or want to pretend you have. When you initialize the screen(s), this class value must be set in the VisualRec data structure along with other display characteristics like the depth and other numbers.

The allowable DepthRec's and VisualRec's are pointed to by fields in the ScreenRec. These are set up when InitOutput() is called; you should Xalloc() appropriate blocks or use static variables initialized to the correct values.

5.2.7. Colormaps for Screens

A colormap is a device-independent mapping between pixel values and colors displayed on the screen.

Different windows on the same screen can have different colormaps at the same time. At any given time, the most recently installed colormap(s) will be in use in the server so that its (their) windows' colors will be guaranteed to be correct. Other windows may be off-color. Although this may seem to be chaotic, in practice most clients use the default colormap for the screen.

The default colormap for a screen is initialized when the screen is initialized. It always remains in existence and is not owned by any regular client. It is owned by client 0 (the server itself). Many clients will simply use this default colormap for their drawing. Depending upon the class of the screen, the entries in this colormap may be modifiable by client applications.

5.2.7.1. Colormap Routines

You need to implement the following routines to handle the device-dependent aspects of color maps. You will end up placing pointers to these procedures in your ScreenRec data structure(s). The sample server implementations of many of these routines are in both cfbcmap.c and mfbcmap.c; since mfb does not do very much with color, the cfb versions are typically more useful prototypes.

```
Bool pScreen->CreateColormap(pColormap)
	ColormapPtr pColormap;
```

This routine is called by the DIX CreateColormap routine after it has allocated all the data for the new colormap and just before it returns to the dispatcher. It is the DDX layer's chance to initialize the colormap, particularly if it is a static map. See the following section for more details on initializing colormaps. The routine returns FALSE if creation failed, such as due to memory limitations. Notice that the colormap has a devPriv field from which you can hang any colormap specific storage you need. Since each colormap might need special information, we attached the field to the colormap and not the visual.

```
void pScreen->DestroyColormap(pColormap)
	ColormapPtr pColormap;
```

This routine is called by the DIX FreeColormap routine after it has uninstalled the colormap and notified all interested parties, and before it has freed any of the colormap storage. It is the DDX layer's chance to free any data it added to the colormap.

```
void pScreen->InstallColormap(pColormap)
        ColormapPtr pColormap;
```

InstallColormap should fill a lookup table on the screen with which the colormap is associated with the colors in pColormap. If there is only one hardware lookup table for the screen, then all colors on the screen may change simultaneously.

In the more general case of multiple hardware lookup tables, this may cause some other colormap to be uninstalled, meaning that windows that subscribed to the colormap that was uninstalled may end up being off-color. See the note, below, about uninstalling maps.

```
void pScreen->UninstallColormap(pColormap)
        ColormapPtr pColormap;
```

UninstallColormap should remove pColormap from screen pColormap->pScreen. Some other map, such as the default map if possible, should be installed in place of pColormap if applicable. If pColormap is the default map, do nothing. If any client has requested ColormapNotify events, the DDX layer must notify the client. (The routine WalkTree() is be used to find such windows. The DIX routines TellNoMap(), TellNewMap() and TellGainedMap() are provided to be used as the procedure parameter to WalkTree. These procedures are in Xserver/dix/colormap.c.)

```
int pScreen->ListInstalledColormaps(pScreen, pCmapList)
        ScreenPtr pScreen;
        XID *pCmapList;
```

ListInstalledColormaps fills the pCMapList in with the resource ids of the installed maps and returns a count of installed maps. pCmapList will point to an array of size MaxInstalledMaps that was allocated by the caller.

```
void pScreen->StoreColors (pmap, ndef, pdefs)
        ColormapPtr pmap;
        int ndef;
        xColorItem *pdefs;
```

StoreColors changes some of the entries in the colormap pmap. The number of entries to change are ndef, and pdefs points to the information describing what to change. Note that partial changes of entries in the colormap are allowed. Only the colors indicated in the flags field of each xColorItem need to be changed. However, all three color fields will be sent with the proper value for the benefit of screens that may not be able to set part of a colormap value. If the screen is a static class, this routine does nothing. The structure of colormap entries is nontrivial; see colormapst.h and the definition of xColorItem in Xproto.h for more details.

```
void pScreen->ResolveColor(pRed, pGreen, pBlue, pVisual)
        unsigned short *pRed, *pGreen, *pBlue;
        VisualPtr pVisual;
```

Given a requested color, ResolveColor returns the nearest color that this hardware is capable of displaying on this visual. In other words, this rounds off each value, in place, to the number of bits per primary color that your screen can use. Remember that each screen has one of these routines. The level of roundoff should be what you would expect from the value you put in the bits_per_rgb field of the pVisual.

Each value is an unsigned value ranging from 0 to 65535. The bits least likely to be used are the lowest ones.

For example, if you had a pseudocolor display with any number of bits per pixel that had a lookup table supplying 6 bits for each color gun (a total of 256K different colors), you would round off each value to 6 bits. Please don't simply truncate these values to the upper 6 bits, scale the result so that the maximum value seen by the client will be 65535 for each primary. This makes color values more portable between different depth displays (a 6-bit truncated white will not look white on an 8-bit display).

5.2.7.2. Initializing a Colormap

When a client requests a new colormap and when the server creates the default colormap, the procedure CreateColormap in the DIX layer is invoked. That procedure allocates memory for the colormap and related storage such as the lists of which client owns which pixels. It then sets a bit, BeingCreated, in the flags field of the ColormapRec and calls the DDX layer's CreateColormap routine. This is your chance to initialize the colormap. If the colormap is static, which you can tell by looking at the class field, you will want to fill in each color cell to match the hardwares notion of the color for that pixel. If the colormap is the default for the screen, which you can tell by looking at the IsDefault bit in the flags field, you should allocate BlackPixel and WhitePixel to match the values you set in the pScreen structure. (Of course, you picked those values to begin with.)

You can also wait and use AllocColor() to allocate blackPixel and whitePixel after the default colormap has been created. If the default colormap is static and you initialized it in pScreen->CreateColormap, then use can use AllocColor afterwards to choose pixel values with the closest rgb values to those desired for blackPixel and whitePixel. If the default colormap is dynamic and uninitialized, then the rgb values you request will be obeyed, and AllocColor will again choose pixel values for you. These pixel values can then be stored into the screen.

There are two ways to fill in the colormap. The simplest way is to use the DIX function AllocColor.

```
int AllocColor (pmap, pred, pgreen, pblue, pPix, client)
    ColormapPtr       pmap;
    unsigned short    *pred, *pgreen, *pblue;
    Pixel             *pPix;
    int               client;
```

This takes three pointers to 16 bit color values and a pointer to a suggested pixel value. The pixel value is either an index into one colormap or a combination of three indices depending on the type of pmap. If your colormap starts out empty, and you don't deliberately pick the same value twice, you will always get your suggested pixel. The truly nervous could check that the value returned in *pPix is the one AllocColor was called with. If you don't care which pixel is used, or would like them sequentially allocated from entry 0, set *pPix to 0. This will find the first free pixel and use that.

AllocColor will take care of all the bookkeeping and will call StoreColors to get the colormap rgb values initialized. The hardware colormap will be changed whenever this colormap is installed.

If for some reason AllocColor doesn't do what you want, you can do your own bookkeeping and call StoreColors yourself. This is much more difficult and shouldn't be necessary for most devices.

5.2.8. Fonts for Screens

A font is a set of bitmaps that depict the symbols in a character set. Each font is for only one typeface in a given size, in other words, just one bitmap for each character. Parallel fonts may be available in a variety of sizes and variations, including "bold" and "italic." X supports fonts for 8-bit and 16-bit character codes (for oriental languages that have more than 256 characters in the font). Glyphs are bitmaps for individual characters.

The source comes with some useful font files in an ASCII, plain-text format that should be comprehensible on a wide variety of operating systems. The text format, referred to as BDF, is a slight extension of the current Adobe 2.1 Bitmap Distribution Format (Adobe Systems, Inc.).

A short paper in PostScript format is included with the sample server that defines BDF. It includes helpful pictures, which is why it is done in PostScript and is not included in this document.

Your implementation should include some sort of font compiler to read these files and generate binary files that are directly usable by your server implementation. The sample server comes with the source for a font compiler.

It is important the font properties contained in the BDF files are preserved across any font compilation. In particular, copyright information cannot be casually tossed aside without legal ramifications. Other properties will be important to some sophisticated applications.

All clients get font information from the server. Therefore, your server can support any fonts it wants to. It should probably support at least the fonts supplied with the X11 tape. In principle, you can convert fonts from other sources or dream up your own fonts for use on your server.

5.2.8.1. Portable Compiled Format

A font compiler is supplied with the sample server. It has compile-time switches to convert the BDF files into a portable binary form, called Portable Compiled Format or PCF. This allows for an arbitrary data format inside the file, and by describing the details of the format in the header of the file, any PCF file can be read by any PCF reading client. By selecting the format which matches the required internal format for your renderer, the PCF reader can avoid reformatting the data each time it is read in. The font compiler should be quite portable.

The fonts included with the tape are stored in fonts/bdf. The font compiler is found in fonts/tools/bdftopcf.

5.2.8.2. Font Realization

Each screen configured into the server has an opportunity at font-load time to "realize" a font into some internal format if necessary. This happens every time the font is loaded into memory.

A font (FontRec in Xserver/include/dixfontstr.h) is a device-independent structure containing a device-independent representation of the font. When a font is created, it is "realized" for each screen. At this point, the screen has the chance to convert the font into some other format. The DDX layer can also put information in the devPrivate storage.

```
Bool pScreen->RealizeFont(pScr, pFont)
	ScreenPtr pScr;
	FontPtr pFont;

Bool pScreen->UnrealizeFont(pScr, pFont)
	ScreenPtr pScr;
	FontPtr pFont;
```

RealizeFont and UnrealizeFont should calculate and allocate these extra data structures and dispose of them when no longer needed. These are called in response to OpenFont and CloseFont requests from the client. The sample server implementation is in mfbfont.c (which does very little).

5.2.9. Other Screen Routines

You must supply several other screen-specific routines for your X server implementation. Some of these are described in other sections:

- GetImage() is described in the Drawing Primitives section.
- GetSpans() is described in the Pixblit routine section.
- Several window and pixmap manipulation procedures are described in the Window section under Drawables.
- The CreateGC() routine is described under Graphics Contexts.

```
void pScreen->QueryBestSize(kind, pWidth, pHeight)
	int kind;
	unsigned short *pWidth, *pHeight;
	ScreenPtr pScreen;
```

QueryBestSize() returns the best sizes for cursors, tiles, and stipples in response to client requests. kind is one of the defined constants CursorShape, TileShape, or StippleShape (defined in X.h). For CursorShape, return the maximum width and height for cursors that you can handle. For TileShape and StippleShape, start with the suggested values in pWidth and pHeight and modify them in place to be optimal values that are greater than or equal to the suggested values. The sample server implementation is in Xserver/mfb/mfbmisc.c.

```
pScreen->SourceValidate(pDrawable, x, y, width, height)
	DrawablePtr pDrawable;
	int x, y, width, height;
```

SourceValidate should be called by CopyArea/CopyPlane primitives when the source drawable is not the same as the destination, and the SourceValidate function pointer in the screen is non-null. If you know that you will never need SourceValidate, you can avoid this check. Currently, SourceValidate is used by the mi software cursor code to remove the cursor from the screen when the source rectangle overlaps the cursor position. x,y,width,height describe the source rectangle (source relative, that is) for the copy operation.

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

SaveScreen() is used for Screen Saver support (see WaitForSomething()). pScreen is the screen to save.

```
Bool pScreen->CloseScreen(pScreen)
   ScreenPtr pScreen;
```

When the server is reset, it calls this routine for each screen.

```
Bool pScreen->CreateScreenResources(pScreen)
   ScreenPtr pScreen;
```

If this routine is not NULL, it will be called once per screen per server initialization/reset after all modules have had a chance to register their devPrivates on all structures that support them (see the section on devPrivates below). If you need to create any resources that have dynamic devPrivates as part of your screen initialization, you should do so in this function instead of in the screen init function passed to AddScreen to guarantee that the resources have a complete set of devPrivates. This routine returns TRUE if successful.

5.3. Drawables

A drawable is a descriptor of a surface that graphics are drawn into, either a window on the screen or a pixmap in memory.

Each drawable has a type, class, ScreenPtr for the screen it is associated with, depth, position, size, and serial number. The type is one of the defined constants DRAWABLE_PIXMAP, DRAWABLE_WINDOW and UNDRAWABLE_WINDOW. (An undrawable window is used for window class InputOnly.) The serial number is guaranteed to be unique across drawables, and is used in determining the validity of the clipping information in a GC. The screen selects the set of procedures used to manipulate and draw into the drawable. Position is used (currently) only by windows; pixmaps must set these fields to 0,0 as this reduces the amount of conditional code executed throughout the mi code. Size indicates the actual client-specified size of the drawable. There are, in fact, no other fields that a window drawable and pixmap drawable have in common besides those mentioned here.

Both PixmapRecs and WindowRecs are structs that start with a drawable and continue on with more fields. Pixmaps have devPrivate pointers which usually point to the pixmap data but could conceivably be used for anything that DDX wants. Both windows and pixmaps have an array of devPrivates unions, one entry of which will probably be used for DDX specific data. Entries in this array are allocated using Allocate{Window|Pixmap}PrivateIndex() (see Wrappers and devPrivates below). This is done because different graphics hardware has different requirements for management; if the graphics is always handled by a processor with an independent address space, there is no point having a pointer to the bit image itself.

The definition of a drawable and a pixmap can be found in the file Xserver/include/pixmapstr.h. The definition of a window can be found in the file Xserver/include/windowstr.h.

5.3.1. Pixmaps

A pixmap is a three-dimensional array of bits stored somewhere offscreen, rather than in the visible portion of the screen's display frame buffer. It can be used as a source or destination in graphics operations. There is no implied interpretation of the pixel values in a pixmap, because it has no associated visual or colormap. There is only a depth that indicates the number of significant bits per pixel. Also, there is no implied physical size for each pixel; all graphic units are in numbers of pixels. Therefore, a pixmap alone does not constitute a complete image; it represents only a rectangular array of pixel values.

Note that the pixmap data structure is reference-counted.

The server implementation is free to put the pixmap data anywhere it sees fit, according to its graphics hardware setup. Many implementations will simply have the data dynamically allocated in the server's address space. More sophisticated implementations may put the data in undisplayed framebuffer storage.

In addition to dynamic devPrivates (see the section on devPrivates below), the pixmap data structure has two fields that are private to the device. Although you can use them for anything you want, they have intended purposes. devKind is intended to be a device specific indication of the pixmap location (host memory, off-screen, etc.). In the sample server, since all pixmaps are in memory, devKind stores the width of the pixmap in bitmap scanline units. devPrivate is probably a pointer to the bits in the pixmap.

A bitmap is a pixmap that is one bit deep.

```
PixmapPtr pScreen->CreatePixmap(pScreen, width, height, depth)
	ScreenPtr pScreen;
	int width, height, depth;
```

This ScreenRec procedure must create a pixmap of the size requested. It must allocate a PixmapRec and fill in all of the fields. The reference count field must be set to 1. If width or height are zero, no space should be allocated for the pixmap data, and if the implementation is using the devPrivate field as a pointer to the pixmap data, it should be set to NULL. If successful, it returns a pointer to the new pixmap; if not, it returns NULL. See Xserver/mfb/mfbpixmap.c for the sample server implementation.

```
Bool pScreen->DestroyPixmap(pPixmap)
	PixmapPtr pPixmap;
```

This ScreenRec procedure must "destroy" a pixmap. It should decrement the reference count and, if zero, it must deallocate the PixmapRec and all attached devPrivate blocks. If successful, it returns TRUE. See Xserver/mfb/mfbpixmap.c for the sample server implementation.

```
Bool
pScreen->ModifyPixmapHeader(pPixmap, width, height, depth, bitsPerPixel, devKind, pPixData)
	PixmapPtr   pPixmap;
	int         width;
	int         height;
	int         depth;
	int         bitsPerPixel;
	int         devKind;
	pointer     pPixData;
```

This routine takes a pixmap header (the PixmapRec plus all the dynamic devPrivates) and initializes the fields of the PixmapRec to the parameters of the same name. pPixmap must have been created via pScreen->CreatePixmap with a zero width or height to avoid allocating space for the pixmap data. pPixData is assumed to be the pixmap data; it will be stored in an implementation-dependent place (usually pPixmap->devPrivate.ptr). This routine returns TRUE if successful. See Xserver/mi/miscrinit.c for the sample server implementation.

```
PixmapPtr
GetScratchPixmapHeader(pScreen, width, height, depth, bitsPerPixel, devKind, pPixData)
	ScreenPtr   pScreen;
	int         width;
	int         height;
	int         depth;
	int         bitsPerPixel;
	int         devKind;
	pointer     pPixData;

void FreeScratchPixmapHeader(pPixmap)
	PixmapPtr pPixmap;
```

DDX should use these two DIX routines when it has a buffer of raw image data that it wants to manipulate as a pixmap temporarily, usually so that some other part of the server can be leveraged to perform some operation on the data. The data should be passed in pPixData, and will be stored in an implementation-dependent place (usually pPixmap->devPrivate.ptr). The other fields go into the corresponding PixmapRec fields. If successful, GetScratchPixmapHeader returns a valid PixmapPtr which can be used anywhere the server expects a pixmap, else it returns NULL. The pixmap should be released when no longer needed (usually within the same function that allocated it) with FreeScratchPixmapHeader.

5.3.2. Windows

A window is a visible, or potentially visible, rectangle on the screen. DIX windowing functions maintain an internal n-ary tree data structure, which represents the current relationships of the mapped windows. Windows that are contained in another window are children of that window and are clipped to the boundaries of the parent. The root window in the tree is the window for the entire screen. Sibling windows constitute a doubly-linked list; the parent window has a pointer to the head and tail of this list. Each child also has a pointer to its parent.

The border of a window is drawn by a DDX procedure when DIX requests that it be drawn. The contents of the window is drawn by the client through requests to the server.

Window painting is orchestrated through an expose event system. When a region is exposed, DIX generates an expose event, telling the client to repaint the window and passing the region that is the minimal area needed to be repainted.

As a favor to clients, the server may retain the output to the hidden parts of windows in off-screen memory; this is called "backing store". When a part of such a window becomes exposed, it can quickly move pixels into place instead of triggering an expose event and waiting for a client on the other end of the network to respond. Even if the network response is insignificant, the time to intelligently paint a section of a window is usually more than the time to just copy already-painted sections. At best, the repainting involves blanking out the area to a background color, which will take about the same amount of time. In this way, backing store can dramatically increase the performance of window moves.

On the other hand, backing store can be quite complex, because all graphics drawn to hidden areas must be intercepted and redirected to the off-screen window sections. Not only can this be complicated for the server programmer, but it can also impact window painting performance. The backing store implementation can choose, at any time, to forget pieces of backing that are written into, relying instead upon expose events to repaint for simplicity.

In X, the decision to use the backing-store scheme is made by you, the server implementor. X provides hooks for implementing backing store, therefore the decision to use this strategy can be made on the fly. For example, you may use backing store only for certain windows that the user requests or you may use backing store until memory runs out, at which time you start dropping pieces of backing as needed to make more room.

When a window operation is requested by the client, such as a window being created or moved, a new state is computed. During this transition, DIX informs DDX what rectangles in what windows are about to become obscured and what rectangles in what windows have become exposed. This provides a hook for the implementation of backing store. If DDX is unable to restore exposed regions, DIX generates expose events to the client. It is then the client's responsibility to paint the window parts that were exposed but not restored.

If a window is resized, pixels sometimes need to be moved, depending upon the application. The client can request "Gravity" so that certain blocks of the window are moved as a result of a resize. For instance, if the window has controls or other items that always hang on the edge of the window, and that edge is moved as a result of the resize, then those pixels should be moved to avoid having the client repaint it. If the client needs to repaint it anyway, such an operation takes time, so it is desirable for the server to approximate the appearance of the window as best it can while waiting for the client to do it perfectly. Gravity is used for that, also.

The window has several fields used in drawing operations:

- clipList - This region, in conjunction with the client clip region in the gc, is used to clip output. clipList has the window's children subtracted from it, in addition to pieces of sibling windows that overlap this window. To get the list with the children included (subwindow-mode is IncludeInferiors), the routine NotClippedByChildren(pWin) returns the unclipped region.
- borderClip is the region used by CopyWindow and includes the area of the window, its children, and the border, but with the overlapping areas of sibling children removed.

Most of the other fields are for DIX use only.

5.3.2.1. Window Procedures in the ScreenRec

You should implement all of the following procedures and store pointers to them in the screen record.

The device-independent portion of the server "owns" the window tree. However, clever hardware might want to know the relationship of mapped windows. There are pointers to procedures in the ScreenRec data structure that are called to give the hardware a chance to update its internal state. These are helpers and hints to DDX only; they do not change the window tree, which is only changed by DIX.

```
Bool pScreen->CreateWindow(pWin)
        WindowPtr pWin;
```

This routine is a hook for when DIX creates a window. It should fill in the "Window Procedures in the WindowRec" below and also allocate the devPrivate block for it.

See Xserver/mfb/mfbwindow.c for the sample server implementation.

```
Bool pScreen->DestroyWindow(pWin);
        WindowPtr pWin;
```

This routine is a hook for when DIX destroys a window. It should deallocate the devPrivate block for it and any other blocks that need to be freed, besides doing other cleanup actions.

See Xserver/mfb/mfbwindow.c for the sample server implementation.

```
Bool pScreen->PositionWindow(pWin, x, y);
        WindowPtr pWin;
        int x, y;
```

This routine is a hook for when DIX moves or resizes a window. It should do whatever private operations need to be done when a window is moved or resized. For instance, if DDX keeps a pixmap tile used for drawing the background or border, and it keeps the tile rotated such that it is longword aligned to longword locations in the frame buffer, then you should rotate your tiles here. The actual graphics involved in moving the pixels on the screen and drawing the border are handled by CopyWindow(), below.

See Xserver/mfb/mfbwindow.c for the sample server implementation.

```
Bool pScreen->RealizeWindow(pWin);
        WindowPtr pWin;

Bool  pScreen->UnrealizeWindow(pWin);
        WindowPtr pWin;
```

These routines are hooks for when DIX maps (makes visible) and unmaps (makes invisible) a window. It should do whatever private operations need to be done when these happen, such as allocating or deallocating structures that are only needed for visible windows. RealizeWindow does NOT draw the window border, background or contents; UnrealizeWindow does NOT erase the window or generate exposure events for underlying windows; this is taken care of by DIX. DIX does, however, call PaintWindowBackground() and PaintWindowBorder() to perform some of these.

```
Bool pScreen->ChangeWindowAttributes(pWin, vmask)
        WindowPtr pWin;
        unsigned long vmask;
```

ChangeWindowAttributes is called whenever DIX changes window attributes, such as the size, front-to-back ordering, title, or anything of lesser severity that affects the window itself. The sample server implements this routine. It computes accelerators for quickly putting up background and border tiles. (See description of the set of routines stored in the WindowRec.)

```
int pScreen->ValidateTree(pParent,  pChild, kind)
        WindowPtr pParent, pChild;
        VTKind kind;
```

ValidateTree calculates the clipping region for the parent window and all of its children. This routine must be provided. The sample server has a machine-independent version in Xserver/mi/mivaltree.c. This is a very difficult routine to replace.

```
void pScreen->PostValidateTree(pParent,  pChild, kind)
        WindowPtr pParent, pChild;
        VTKind kind;
```

If this routine is not NULL, DIX calls it shortly after calling ValidateTree, passing it the same arguments. This is useful for managing multi-layered framebuffers. The sample server sets this to NULL.

```
void pScreen->WindowExposures(pWin, pRegion, pBSRegion)
        WindowPtr pWin;
        RegionPtr pRegion;
        RegionPtr pBSRegion;
```

The WindowExposures() routine paints the border and generates exposure events for the window. pRegion is an unoccluded region of the window, and pBSRegion is an occluded region that has backing store. Since exposure events include a rectangle describing what was exposed, this routine may have to send back a series of exposure events, one for each rectangle of the region. The count field in the expose event is a hint to the client as to the number of regions that are after this one. This routine must be provided. The sample server has a machine-independent version in Xserver/mi/miexpose.c.

```
void pScreen->ClipNotify (pWin, dx, dy)
        WindowPtr pWin;
        int dx, dy;
```

Whenever the cliplist for a window is changed, this function is called to perform whatever hardware manipulations might be necessary. When called, the clip list and border clip regions in the window are set to the new values. dx,dy are the distance that the window has been moved (if at all).

5.3.2.2. Window Painting Procedures

In addition to the procedures listed above, there are four routines which manipulate the actual window image directly. In the sample server, mi implementations will work for most purposes and mfb/cfb routines speed up situations, such as solid backgrounds/borders or tiles that are 8, 16 or 32 pixels square.

These three routines are used for systems that implement a backing-store scheme for it to know when to stash away areas of pixels and to restore or reposition them.

```
void pScreen->ClearToBackground(pWin, x, y, w, h, generateExposures);
        WindowPtr pWin;
        int x, y, w, h;
        Bool generateExposures;
```

This routine is called on a window in response to a ClearToBackground request from the client. This request has two different but related functions, depending upon generateExposures.

If generateExposures is true, the client is declaring that the given rectangle on the window is incorrectly painted and needs to be repainted. The sample server implementation calculates the exposure region and hands it to the DIX procedure HandleExposures(), which calls the WindowExposures() routine, below, for the window and all of its child windows.

If generateExposures is false, the client is trying to simply erase part of the window to the background fill style. ClearToBackground should write the background color or tile to the rectangle in question (probably using PaintWindowBackground). If w or h is zero, it clears all the way to the right or lower edge of the window.

The sample server implementation is in Xserver/mi/miwindow.c.

```
void pScreen->PaintWindowBackground(pWin, region, kind)
        WindowPtr pWin;
        RegionPtr region;
        int kind;       /* must be PW_BACKGROUND */

void pScreen->PaintWindowBorder(pWin, region, kind)
        WindowPtr pWin;
        RegionPtr region;
        int kind;       /* must be PW_BORDER */
```

These two routines are for painting pieces of the window background or border. They both actually paint the area designated by region. The kind parameter is a defined constant that is always PW_BACKGROUND or PW_BORDER, as shown. Therefore, you can use the same routine for both. The defined constant tells the routine whether to use the window's border fill style or its background fill style to paint the given region. Both fill styles consist of a union which holds a tile pointer and a pixel value, along with a separate variable which indicates which entry is valid. For PW_BORDER, borderIsPixel != 0 indicates that the border PixUnion contains a pixel value, else a tile. For PW_BACKGROUND there are four values, contained in backgroundState; None, ParentRelative, BackgroundPixmap and BackgroundPixel. None indicates that the region should be left unfilled, while ParentRelative indicates that the background of the parent is inherited (see the Protocol document for the exact semantics).

```
void pScreen->CopyWindow(pWin, oldpt, oldRegion);
        WindowPtr pWin;
        DDXPointRec oldpt;
        RegionPtr oldRegion;
```

CopyWindow is called when a window is moved, and graphically moves to pixels of a window on the screen. It should not change any other state within DDX (see PositionWindow(), above).

oldpt is the old location of the upper-left corner. oldRegion is the old region it is coming from. The new location and new region is stored in the WindowRec. oldRegion might modified in place by this routine (the sample implementation does this).

CopyArea could be used, except that this operation has more complications. First of all, you do not want to copy a rectangle onto a rectangle. The original window may be obscured by other windows, and the new window location may be similarly obscured. Second, some hardware supports multiple windows with multiple depths, and your routine needs to take care of that.

The pixels in oldRegion (with reference point oldpt) are copied to the window's new region (pWin->borderClip). pWin->borderClip is gotten directly from the window, rather than passing it as a parameter.

The sample server implementation is in Xserver/mfb/mfbwindow.c.

5.3.2.3. Screen Operations for Backing Store

Each ScreenRec has six functions which provide the backing store interface. For screens not supporting backing store, these pointers may be nul. Servers that implement some backing store scheme must fill in the procedure pointers for the procedures below, and must maintain the backStorage field in each window struct. The sample implementation is in mi/mibstore.c.

```
void pScreen->SaveDoomedAreas(pWin, pRegion, dx, dy)
	WindowPtr pWin;
	RegionPtr pRegion;
	int dx, dy;
```

This routine saves the newly obscured region, pRegion, in backing store. dx, dy indicate how far the window is being moved, useful as the obscured region is relative to the window as it will appear in the new location, rather then relative to the bits as the are on the screen when the function is invoked.

```
RegionPtr pScreen->RestoreAreas(pWin, pRegion)
	WindowPtr pWin;
	RegionPtr pRegion;
```

This looks at the exposed region of the window, pRegion, and tries to restore to the screen the parts that have been saved. It removes the restored parts from the backing storage (because they are now on the screen) and subtracts the areas from the exposed region. The returned region is the area of the window which should have expose events generated for and can be either a new region, pWin->exposed, or NULL. The region left in pRegion is set to the area of the window which should be painted with the window background.

```
RegionPtr pScreen->TranslateBackingStore(pWin, dx, dy, oldClip, oldx, oldy)
	WindowPtr pWin;
	int dx, dy;
	RegionPtr oldClip;
	int oldx, oldy;
```

This is called when the window is moved or resized so that the backing store can be translated if necessary. oldClip is the old cliplist for the window, which is used to save doomed areas if the window is moved underneath its parent as a result of bitgravity. The returned region represents occluded areas of the window for which the backing store contents are invalid.

```
void pScreen->ExposeCopy(pSrc, pDst, pGC, prgnExposed, srcx, srcy, dstx, dsty, plane)
	WindowPtr pSrc;
	DrawablePtr pDst;
	GCPtr pGC;
	RegionPtr prgnExposed;
	int srcx;
	int srcy;
	int dstx;
	int dsty;
	unsigned long plane;
```

Copies a region from the backing store of pSrc to pDst.

```
RegionPtr pScreen->ClearBackingStore(pWindow, x, y, w, h, generateExposures)
	WindowPtr pWindow;
	int x;
	int y;
```

```
        int w;
        int h;
        Bool generateExposures;
```

Clear the given area of the backing pixmap with the background of the window. If generateExposures is TRUE, generate exposure events for the area. Note that if the area has any part outside the saved portions of the window, we do not allow the count in the expose events to be 0, since there will be more expose events to come.

```
    void pScreen->DrawGuarantee(pWindow, pGC, guarantee)
        WindowPtr pWindow;
        GCPtr pGC;
        int guarantee;
```

This informs the backing store layer that you are about to validate a gc with a window, and that subsequent output to the window is (or is not) guaranteed to be already clipped to the visible regions of the window.

5.3.2.4. Screen Operations for Multi-Layered Framebuffers

The following screen functions are useful if you have a framebuffer with multiple sets of independent bit planes, e.g. overlays or underlays in addition to the "main" planes. If you have a simple single-layer framebuffer, you should probably use the mi versions of these routines in mi/miwindow.c. This can be easily accomplished by calling miScreenInit.

```
  void pScreen->MarkWindow(pWin)
      WindowPtr pWin;
```

This formerly dix function MarkWindow has moved to ddx and is accessed via this screen function. This function should store something, usually a pointer to a device-dependent structure, in pWin->valdata so that ValidateTree has the information it needs to validate the window.

```
  Bool pScreen->MarkOverlappedWindows(parent, firstChild, ppLayerWin)
      WindowPtr parent;
      WindowPtr firstChild;
      WindowPtr * ppLayerWin;
```

This formerly dix function MarkWindow has moved to ddx and is accessed via this screen function. In the process, it has grown another parameter: ppLayerWin, which is filled in with a pointer to the window at which save under marking and ValidateTree should begin. In the single-layered framebuffer case, pLayerWin == pWin.

```
  Bool pScreen->ChangeSaveUnder(pLayerWin, firstChild)
      WindowPtr pLayerWin;
      WindowPtr firstChild;
```

The dix functions ChangeSaveUnder and CheckSaveUnder have moved to ddx and are accessed via this screen function. pLayerWin should be the window returned in the ppLayerWin parameter of MarkOverlappedWindows. The function may turn on backing store for windows that might be covered, and may partially turn off backing store for windows. It returns TRUE if PostChangeSaveUnder needs to be called to finish turning off backing store.

```
  void pScreen->PostChangeSaveUnder(pLayerWin, firstChild)
      WindowPtr pLayerWin;
      WindowPtr firstChild;
```

The dix function DoChangeSaveUnder has moved to ddx and is accessed via this screen function. This function completes the job of turning off backing store that was started by ChangeSaveUnder.

```
void pScreen->MoveWindow(pWin, x, y, pSib, kind)
	WindowPtr pWin;
	int x;
	int y;
	WindowPtr pSib;
	VTKind kind;
```

The formerly dix function MoveWindow has moved to ddx and is accessed via this screen function. The new position of the window is given by x,y. kind is VTMove if the window is only moving, or VTOther if the border is also changing.

```
void pScreen->ResizeWindow(pWin, x, y, w, h, pSib)
	WindowPtr pWin;
	int x;
	int y;
	unsigned int w;
	unsigned int h;
	WindowPtr pSib;
```

The formerly dix function SlideAndSizeWindow has moved to ddx and is accessed via this screen function. The new position is given by x,y. The new size is given by w,h.

```
WindowPtr pScreen->GetLayerWindow(pWin)
	WindowPtr pWin
```

This is a new function which returns a child of the layer parent of pWin.

```
void pScreen->HandleExposures(pWin)
	WindowPtr pWin;
```

The formerly dix function HandleExposures has moved to ddx and is accessed via this screen function. This function is called after ValidateTree and uses the information contained in valdata to send exposures to windows.

```
void pScreen->ReparentWindow(pWin, pPriorParent)
	WindowPtr pWin;
	WindowPtr pPriorParent;
```

This function will be called when a window is reparented. At the time of the call, pWin will already be spliced into its new position in the window tree, and pPriorParent is its previous parent. This function can be NULL.

```
void pScreen->SetShape(pWin)
	WindowPtr pWin;
```

The formerly dix function SetShape has moved to ddx and is accessed via this screen function. The window's new shape will have already been stored in the window when this function is called.

```
void pScreen->ChangeBorderWidth(pWin, width)
	WindowPtr pWin;
	unsigned int width;
```

The formerly dix function ChangeBorderWidth has moved to ddx and is accessed via this screen function. The new border width is given by width.

```
void pScreen->MarkUnrealizedWindow(pChild, pWin, fromConfigure)
     WindowPtr pChild;
     WindowPtr pWin;
     Bool fromConfigure;
```

This function is called for windows that are being unrealized as part of an UnrealizeTree. pChild is the window being unrealized, pWin is an ancestor, and the fromConfigure value is simply propogated from UnrealizeTree.

5.4. Graphics Contexts and Validation

This graphics context (GC) contains state variables such as foreground and background pixel value (color), the current line style and width, the current tile or stipple for pattern generation, the current font for text generation, and other similar attributes.

In many graphics systems, the equivalent of the graphics context and the drawable are combined as one entity. The main distinction between the two kinds of status is that a drawable describes a writing surface and the writings that may have already been done on it, whereas a graphics context describes the drawing process. A drawable is like a chalkboard. A GC is like a piece of chalk.

Unlike many similar systems, there is no "current pen location." Every graphic operation is accompanied by the coordinates where it is to happen.

The GC also includes two vectors of procedure pointers, the first operate on the GC itself and are called GC funcs. The second, called GC ops, contains the functions that carry out the fundamental graphic operations such as drawing lines, polygons, arcs, text, and copying bitmaps. The DDX graphic software can, if it wants to be smart, change these two vectors of procedure pointers to take advantage of hardware/firmware in the server machine, which can do a better job under certain circumstances. To reduce the amount of memory consumed by each GC, it is wise to create a few "boilerplate" GC ops vectors which can be shared by every GC which matches the constraints for that set. Also, it is usually reasonable to have every GC created by a particular module to share a common set of GC funcs. Samples of this sort of sharing can be seen in cfb/cfbgc.c and mfb/mfbgc.c.

The DDX software is notified any time the client (or DIX) uses a changed GC. For instance, if the hardware has special support for drawing fixed-width fonts, DDX can intercept changes to the current font in a GC just before drawing is done. It can plug into either a fixed-width procedure that makes the hardware draw characters, or a variable-width procedure that carefully lays out glyphs by hand in software, depending upon the new font that is selected.

A definition of these structures can be found in the file Xserver/include/gcstruct.h.

Also included in each GC is an array of devPrivates which portions of the DDX can use for any reason. Entries in this array are allocated with AllocateGCPrivateIndex() (see Wrappers and Privates below).

The DIX routines available for manipulating GCs are CreateGC, ChangeGC, CopyGC, SetClipRects, SetDashes, and FreeGC.

```
GCPtr CreateGC(pDrawable, mask, pval, pStatus)
   DrawablePtr pDrawable;
   BITS32 mask;
   XID *pval;
   int *pStatus;
```

```
int ChangeGC(pGC, mask, pval)
   GCPtr pGC;
   BITS32 mask;
   XID *pval;

int CopyGC(pgcSrc, pgcDst, mask)
   GCPtr pgcSrc;
   GCPtr pgcDst;
   BITS32 mask;

int SetClipRects(pGC, xOrigin, yOrigin, nrects, prects, ordering)
   GCPtr pGC;
   int xOrigin, yOrigin;
   int nrects;
   xRectangle *prects;
   int ordering;

SetDashes(pGC, offset, ndash, pdash)
   GCPtr pGC;
   unsigned offset;
   unsigned ndash;
   unsigned char *pdash;

int FreeGC(pGC, gid)
   GCPtr pGC;
   GContext gid;
```

As a convenience, each Screen structure contains an array of GCs that are preallocated, one at each depth the screen supports. These are particularly useful in the mi code. Two DIX routines must be used to get these GCs:

```
GCPtr GetScratchGC(depth, pScreen)
   int depth;
   ScreenPtr pScreen;

FreeScratchGC(pGC)
   GCPtr pGC;
```

Always use these two routines, don't try to extract the scratch GC yourself -- someone else might be using it, so a new one must be created on the fly.

If you need a GC for a very long time, say until the server is restarted, you should not take one from the pool used by GetScratchGC, but should get your own using CreateGC or CreateScratchGC. This leaves the ones in the pool free for routines that only need it for a little while and don't want to pay a heavy cost to get it.

```
GCPtr CreateScratchGC(pScreen, depth)
   ScreenPtr pScreen;
   int depth;
```

NULL is returned if the GC cannot be created. The GC returned can be freed with FreeScratchGC.

5.4.1. Details of operation

At screen initialization, a screen must supply a GC creation procedure. At GC creation, the screen must fill in GC funcs and GC ops vectors (Xserver/include/gcstruct.h). For any particular GC, the func vector must remain constant, while the op vector may vary. This invariant is to ensure that Wrappers work correctly.

When a client request is processed that results in a change to the GC, the device-independent state of the GC is updated. This includes a record of the state that changed. Then the ChangeGC GC func is called. This is useful for graphics subsystems that are able to process state changes in parallel with the server CPU. DDX may opt not to take any action at GC-modify time. This is more efficient if multiple GC-modify requests occur between draws using a given GC.

Validation occurs at the first draw operation that specifies the GC after that GC was modified. DIX calls then the ValidateGC GC func. DDX should then update its internal state. DDX internal state may be stored as one or more of the following: 1) device private block on the GC; 2) hardware state; 3) changes to the GC ops.

The GC contains a serial number, which is loaded with a number fetched from the window that was drawn into the last time the GC was used. The serial number in the drawable is changed when the drawable's clipList or absCorner changes. Thus, by comparing the GC serial number with the drawable serial number, DIX can force a validate if the drawable has been changed since the last time it was used with this GC.

In addition, the drawable serial number is always guaranteed to have the most significant bit set to 0. Thus, the DDX layer can set the most significant bit of the serial number to 1 in a GC to force a validate the next time the GC is used. DIX also uses this technique to indicate that a change has been made to the GC by way of a SetGC, a SetDashes or a SetClip request.

5.4.2. GC Handling Routines

The ScreenRec data structure has a pointer for CreateGC().

```
Bool pScreen->CreateGC(pGC)
	GCPtr pGC;
```

This routine must fill in the fields of a dynamically allocated GC that is passed in. It does NOT allocate the GC record itself or fill in the defaults; DIX does that.

This must fill in both the GC funcs and ops; none of the drawing functions will be called before the GC has been validated, but the others (dealing with allocating of clip regions, changing and destroying the GC, etc.) might be.

The GC funcs vector contains pointers to 7 routines and a devPrivate field:

```
pGC->funcs->ChangeGC(pGC, changes)
	GCPtr pGC;
	unsigned long changes;
```

This GC func is called immediately after a field in the GC is changed. changes is a bit mask indicating the changed fields of the GC in this request.

The ChangeGC routine is useful if you have a system where state-changes to the GC can be swallowed immediately by your graphics system, and a validate is not necessary.

```
pGC->funcs->ValidateGC(pGC, changes, pDraw)
	GCPtr pGC;
```

```
        unsigned long changes;
        DrawablePtr pDraw;
```

ValidateGC is called by DIX just before the GC will be used when one of many possible changes to the GC or the graphics system has happened. It can modify a devPrivates field of the GC or its contents, change the op vector, or change hardware according to the values in the GC. It may not change the device-independent portion of the GC itself.

In almost all cases, your ValidateGC() procedure should take the regions that drawing needs to be clipped to and combine them into a composite clip region, which you keep a pointer to in the private part of the GC. In this way, your drawing primitive routines (and whatever is below them) can easily determine what to clip and where. You should combine the regions clientClip (the region that the client desires to clip output to) and the region returned by NotClippedByChildren(), in DIX. An example is in Xserver/mfb/mfbgc.c.

Some kinds of extension software may cause this routine to be called more than originally intended; you should not rely on algorithms that will break under such circumstances.

See the Strategies document for more information on creatively using this routine.

```
    pGC->funcs->CopyGC(pGCSrc, mask, pGCDst)
        GCPtr pGCSrc;
        unsigned long mask;
        GCPtr pGCDst;
```

This routine is called by DIX when a GC is being copied to another GC. This is for situations where dynamically allocated chunks of memory are hanging off a GC devPrivates field which need to be transferred to the destination GC.

```
    pGC->funcs->DestroyGC(pGC)
        GCPtr pGC;
```

This routine is called before the GC is destroyed for the entity interested in this GC to clean up after itself. This routine is responsible for freeing any auxiliary storage allocated.

5.4.3. GC Clip Region Routines

The GC clientClip field requires three procedures to manage it. These procedures are in the GC funcs vector. The underlying principle is that dix knows nothing about the internals of the clipping information, (except when it has come from the client), and so calls ddX whenever it needs to copy, set, or destroy such information. It could have been possible for dix not to allow ddX to touch the field in the GC, and require it to keep its own copy in devPriv, but since clip masks can be very large, this seems like a bad idea. Thus, the server allows ddX to do whatever it wants to the clientClip field of the GC, but requires it to do all manipulation itself.

```
    void pGC->funcs->ChangeClip(pGC, type, pValue, nrects)
        GCPtr pGC;
        int type;
        char *pValue;
        int nrects;
```

This routine is called whenever the client changes the client clip region. The pGC points to the GC involved, the type tells what form the region has been sent in. If type is CT_NONE, then there is no client clip. If type is CT_UNSORTED, CT_YBANDED or CT_YXBANDED, then pValue pointer to a list of rectangles, nrects long. If type is CT_REGION, then pValue pointer to a RegionRec from the mi region code. If type is CT_PIXMAP pValue is a pointer to a pixmap. (The defines for CT_NONE, etc. are in Xserver/include/gc.h.) This routine is responsible for incrementing any necessary reference counts (e.g. for a pixmap clip mask) for the new clipmask and freeing anything that used to be in the GC's clipMask field. The lists of rectangles passed in can be freed with Xfree(), the regions can be destroyed with the RegionDestroy field in the screen, and pixmaps can be destroyed by calling the screen's DestroyPixmap function. DIX and MI code expect what they pass in to this to be freed or otherwise inaccessible, and will never look inside what's been put in the GC. This is a good place to be wary of storage leaks.

In the sample server, this routine transforms either the bitmap or the rectangle list into a region, so that future routines will have a more predictable starting point to work from. (The validate routine must take this client clip region and merge it with other regions to arrive at a composite clip region before any drawing is done.)

```
void pGC->funcs->DestroyClip(pGC)
        GCPtr pGC;
```

This routine is called whenever the client clip region must be destroyed. The pGC points to the GC involved. This call should set the clipType field of the GC to CT_NONE. In the sample server, the pointer to the client clip region is set to NULL by this routine after destroying the region, so that other software (including ChangeClip() above) will recognize that there is no client clip region.

```
void pGC->funcs->CopyClip(pgcDst, pgcSrc)
        GCPtr pgcDst, pgcSrc;
```

This routine makes a copy of the clipMask and clipType from pgcSrc into pgcDst. It is responsible for destroying any previous clipMask in pgcDst. The clip mask in the source can be the same as the clip mask in the dst (clients do the strangest things), so care must be taken when destroying things. This call is required because dix does not know how to copy the clip mask from pgcSrc.

5.5. Drawing Primitives

The X protocol (rules for the byte stream that goes between client and server) does all graphics using primitive operations, which are called Drawing Primitives. These include line drawing, area filling, arcs, and text drawing. Your implementation must supply 16 routines to perform these on your hardware. (The number 16 is arbitrary.)

More specifically, 16 procedure pointers are in each GC op vector. At any given time, ALL of them MUST point to a valid procedure that attempts to do the operation assigned, although the procedure pointers may change and may point to different procedures to carry out the same operation. A simple server will leave them all pointing to the same 16 routines, while a more optimized implementation will switch each from one procedure to another, depending upon what is most optimal for the current GC and drawable.

The sample server contains a considerable chunk of code called the mi (machine independent) routines, which serve as drawing primitive routines. Many server implementations will be able to use these as-is, because they work for arbitrary depths. They make no assumptions about the formats of pixmaps and frame buffers, since they call a set of routines known as the "Pixblit Routines" (see next section). They do assume that the way to draw is through these low-level routines that apply pixel values rows at a time. If your hardware or firmware gives more performance when things are done differently, you will want to take this fact into account and rewrite some or all of the drawing primitives to fit your needs.

5.5.1. GC Components

This section describes the fields in the GC that affect each drawing primitive. The only primitive that is not affected is GetImage, which does not use a GC because its destination is a protocol-style bit image. Since each drawing primitive mirrors exactly the X protocol request of the same name, you should refer to the X protocol specification document for more details.

ALL of these routines MUST CLIP to the appropriate regions in the drawable. Since there are many regions to clip to simultaneously, your ValidateGC routine should combine these into a unified clip region to which your drawing routines can quickly refer. This is exactly what the cfb and mfb routines supplied with the sample server do. The mi implementation passes responsibility for clipping while drawing down to the Pixblit routines.

Also, all of them must adhere to the current plane mask. The plane mask has one bit for every bit plane in the drawable; only planes with 1 bits in the mask are affected by any drawing operation.

All functions except for ImageText calls must obey the alu function. This is usually Copy, but could be any of the allowable 16 raster-ops.

All of the functions, except for CopyArea, might use the current foreground and background pixel values. Each pixel value is 32 bits. These correspond to foreground and background colors, but you have to run them through the colormap to find out what color the pixel values represent. Do not worry about the color, just apply the pixel value.

The routines that draw lines (PolyLine, PolySegment, PolyRect, and PolyArc) use the line width, line style, cap style, and join style. Line width is in pixels. The line style specifies whether it is solid or dashed, and what kind of dash. The cap style specifies whether Rounded, Butt, etc. The join style specifies whether joins between joined lines are Miter, Round or Beveled. When lines cross as part of the same polyline, they are assumed to be drawn once. (See the X protocol specification for more details.)

Zero-width lines are NOT meant to be really zero width; this is the client's way of telling you that you can optimize line drawing with little regard to the end caps and joins. They are called "thin" lines and are meant to be one pixel wide. These are frequently done in hardware or in a streamlined assembly language routine.

Lines with widths greater than zero, though, must all be drawn with the same algorithm, because client software assumes that every jag on every line at an angle will come at the same place. Two lines that should have one pixel in the space between them (because of their distance apart and their widths) should have such a one-pixel line of space between them if drawn, regardless of angle.

The solid area fill routines (FillPolygon, PolyFillRect, PolyFillArc) all use the fill rule, which specifies subtle interpretations of what points are inside and what are outside of a given polygon. The PolyFillArc routine also uses the arc mode, which specifies whether to fill pie segments or single-edge slices of an ellipse.

The line drawing, area fill, and PolyText routines must all apply the correct "fill style." This can be either a solid foreground color, a transparent stipple, an opaque stipple, or a tile. Stipples are bitmaps where the 1 bits represent that the foreground color is written, and 0 bits represent that either the pixel is left alone (transparent) or that the background color is written (opaque). A tile is a pixmap of the full depth of the GC that is applied in its full glory to all areas. The stipple and tile patterns can be any rectangular size, although some implementations will be faster for certain sizes such as 8x8 or 32x32. The mi implementation passes this responsibility down to the Pixblit routines.

See the X protocol document for full details. The description of the CreateGC request has a very good, detailed description of these attributes.

5.5.2. The Primitives

The Drawing Primitives are as follows:

```
RegionPtr pGC->ops->CopyArea(src, dst, pGC, srcx, srcy, w, h, dstx, dsty)
	DrawablePtr dst, src;
	GCPtr pGC;
	int srcx, srcy, w, h, dstx, dsty;
```

CopyArea copies a rectangle of pixels from one drawable to another of the same depth. To effect scrolling, this must be able to copy from any drawable to itself, overlapped. No squeezing or stretching is done because the source and destination are the same size. However, everything is still clipped to the clip regions of the destination drawable.

If pGC->graphicsExposures is True, any portions of the destination which were not valid in the source (either occluded by covering windows, or outside the bounds of the drawable) should be collected together and returned as a region (if this resultant region is empty, NULL can be returned instead). Furthermore, the invalid bits of the source are not copied to the destination and (when the destination is a window) are filled with the background tile. The sample routine miHandleExposures generates the appropriate return value and fills the invalid area using pScreen->PaintWindowBackground.

For instance, imagine a window that is partially obscured by other windows in front of it. As text is scrolled on your window, the pixels that are scrolled out from under obscuring windows will not be available on the screen to copy to the right places, and so an exposure event must be sent for the client to correctly repaint them. Of course, if you implement some sort of backing store, you could do this without resorting to exposure events.

An example implementation is mfbCopyArea() in Xserver/mfb/mfbbitblt.c.

```
RegionPtr pGC->ops->CopyPlane(src, dst, pGC, srcx, srcy, w, h, dstx, dsty, plane)
	DrawablePtr dst, src;
	GCPtr pGC;
	int srcx, srcy, w, h, dstx, dsty;
	unsigned long plane;
```

CopyPlane must copy one plane of a rectangle from the source drawable onto the destination drawable. Because this routine only copies one bit out of each pixel, it can copy between drawables of different depths. This is the only way of copying between drawables of different depths, except for copying bitmaps to pixmaps and applying foreground and background colors to it. All other conditions of CopyArea apply to CopyPlane too.

An example implementation is mfbCopyPlane() in Xserver/mfb/mfbbitblt.c.

```
void pGC->ops->PolyPoint(dst, pGC, mode, n, pPoint)
	DrawablePtr dst;
	GCPtr pGC;
	int mode;
	int n;
	DDXPointPtr pPoint;
```

PolyPoint draws a set of one-pixel dots (foreground color) at the locations given in the array. mode is one of the defined constants Origin (absolute coordinates) or Previous (each coordinate is relative to the last). Note that this does not use the background color or any tiles or stipples.

Example implementations are mfbPolyPoint() in Xserver/mfb/mfbpolypnt.c and miPolyPoint in Xserver/mi/mipolypnt.c.

```
void pGC->ops->Polylines(dst, pGC, mode, n, pPoint)
	DrawablePtr dst;
	GCPtr pGC;
	int mode;
	int n;
	DDXPointPtr pPoint;
```

Similar to PolyPoint, Polylines draws lines between the locations given in the array. Zero-width lines are NOT meant to be really zero width; this is the client's way of telling you that you can maximally optimize line drawing with little regard to the end caps and joins. mode is one of the defined constants Previous or Origin, depending upon whether the points are each relative to the last or are absolute.

Example implementations are miWideLine() and miWideDash() in mi/miwideline.c and miZeroLine() in mi/mizerline.c.

```
void pGC->ops->PolySegment(dst, pGC, n, pPoint)
	DrawablePtr dst;
	GCPtr pGC;
	int n;
	xSegment *pSegments;
```

PolySegments draws unconnected lines between pairs of points in the array; the array must be of even size; no interconnecting lines are drawn.

An example implementation is miPolySegment() in mipolyseg.c.

```
void pGC->ops->PolyRectangle(dst, pGC, n, pRect)
	DrawablePtr dst;
	GCPtr pGC;
	int n;
	xRectangle *pRect;
```

PolyRectangle draws outlines of rectangles for each rectangle in the array.

An example implementation is miPolyRectangle() in Xserver/mi/mipolyrect.c.

```
void pGC->ops->PolyArc(dst, pGC, n, pArc)
	DrawablePtr dst;
	GCPtr pGC;
	int n;
	xArc*pArc;
```

PolyArc draws connected conic arcs according to the descriptions in the array. See the protocol specification for more details.

Example implementations are miZeroPolyArc in Xserver/mi/mizerarc. and miPolyArc() in Xserver/mi/miarc.c.

```
void pGC->ops->FillPolygon(dst, pGC, shape, mode, count, pPoint)
	DrawablePtr dst;
	GCPtr pGC;
```

```
        int shape;
        int mode;
        int count;
        DDXPointPtr pPoint;
```

FillPolygon fills a polygon specified by the points in the array with the appropriate fill style. If necessary, an extra border line is assumed between the starting and ending lines. The shape can be used as a hint to optimize filling; it indicates whether it is convex (all interior angles less than 180), nonconvex (some interior angles greater than 180 but border does not cross itself), or complex (border crosses itself). You can choose appropriate algorithms or hardware based upon mode. mode is one of the defined constants Previous or Origin, depending upon whether the points are each relative to the last or are absolute.

An example implementation is miFillPolygon() in Xserver/mi/mipoly.c.

```
    void pGC->ops->PolyFillRect(dst, pGC, n, pRect)
        DrawablePtr dst;
        GCPtr pGC;
        int n;
        xRectangle *pRect;
```

PolyFillRect fills multiple rectangles.

Example implementations are mfbPolyFillRect() in Xserver/mfb/mfbfillrct.c and miPolyFillRect() in Xserver/mi/mifillrct.c.

```
    void pGC->ops->PolyFillArc(dst, pGC, n, pArc)
        DrawablePtr dst;
        GCPtr pGC;
        int n;
        xArc *pArc;
```

PolyFillArc fills a shape for each arc in the list that is bounded by the arc and one or two line segments with the current fill style.

An example implementation is miPolyFillArc() in Xserver/mi/mifillarc.c.

```
    void pGC->ops->PutImage(dst, pGC, depth, x, y, w, h, leftPad, format, pBinImage)
        DrawablePtr dst;
        GCPtr pGC;
        int x, y, w, h;
        int format;
        char *pBinImage;
```

PutImage copies a pixmap image into the drawable. The pixmap image must be in X protocol format (either Bitmap, XYPixmap, or ZPixmap), and format tells the format. (See the X protocol specification for details on these formats). You must be able to accept all three formats, because the client gets to decide which format to send. Either the drawable and the pixmap image have the same depth, or the source pixmap image must be a Bitmap. If a Bitmap, the foreground and background colors will be applied to the destination.

An example implementation is miPutImage() in Xserver/mfb/mibitblt.c.

```
    void pScreen->GetImage(src, x, y, w, h, format, planeMask, pBinImage)
         DrawablePtr src;
```

```
int x, y, w, h;
unsigned int format;
unsigned long planeMask;
char *pBinImage;
```

GetImage copies the bits from the source drawable into the destination pointer. The bits are written into the buffer according to the server-defined pixmap padding rules. pBinImage is guaranteed to be big enough to hold all the bits that must be written.

This routine does not correspond exactly to the X protocol GetImage request, since DIX has to break the reply up into buffers of a size requested by the transport layer. If format is ZPixmap, the bits are written in the ZFormat for the depth of the drawable; if there is a 0 bit in the planeMask for a particular plane, all pixels must have the bit in that plane equal to 0. If format is XYPixmap, planemask is guaranteed to have a single bit set; the bits should be written in Bitmap format, which is the format for a single plane of an XYPixmap.

An example implementation is miGetImage() in Xserver/mi/mibitblt.c.

```
void pGC->ops->ImageText8(pDraw, pGC, x, y, count, chars)
	DrawablePtr pDraw;
	GCPtr pGC;
	int x, y;
	int count;
	char *chars;
```

ImageText8 draws text. The text is drawn in the foreground color; the background color fills the remainder of the character rectangles. The coordinates specify the baseline and start of the text.

An example implementation is miImageText8() in Xserver/mi/mipolytext.c.

```
int pGC->ops->PolyText8(pDraw, pGC, x, y, count, chars)
	DrawablePtr pDraw;
	GCPtr pGC;
	int x, y;
	int count;
	char *chars;
```

PolyText8 works like ImageText8, except it draws with the current fill style for special effects such as shaded text. See the X protocol specification for more details.

An example implementation is miPolyText8() in Xserver/mi/mipolytext.c.

```
int pGC->ops->PolyText16(pDraw, pGC, x, y, count, chars)
	DrawablePtr pDraw;
	GCPtr pGC;
	int x, y;
	int count;
	unsigned short *chars;

void pGC->ops->ImageText16(pDraw, pGC, x, y, count, chars)
	DrawablePtr pDraw;
	GCPtr pGC;
	int x, y;
	int count;
```

```
unsigned short *chars;
```

These two routines are the same as the "8" versions, except that they are for 16-bit character codes (useful for oriental writing systems).

The primary difference is in the way the character information is looked up. The 8-bit and the 16-bit versions obviously have different kinds of character values to look up; the main goal of the lookup is to provide a pointer to the CharInfo structs for the characters to draw and to pass these pointers to the Glyph routines. Given a CharInfo struct, lower-level software can draw the glyph desired with little concern for other characteristics of the font.

16-bit character fonts have a row-and-column scheme, where the 2bytes of the character code constitute the row and column in a square matrix of CharInfo structs. Each font has row and column minimum and maximum values; the CharInfo structures form a two-dimensional matrix.

Example implementations are miPolyText16() and miImageText16() in Xserver/mi/mipolytext.c.

See the X protocol specification for more details on these graphic operations.

There is a hook in the GC ops, called LineHelper, that used to be used in the sample implementation by the code for wide lines. It no longer servers any purpose in the sample servers, but still exists, #ifdef'ed by NEED_LINEHELPER, in case someone needs it.

5.6. Pixblit Procedures

The Drawing Primitive functions must be defined for your server. One possible way to do this is to use the mi routines from the sample server. If you choose to use the mi routines (even part of them!) you must implement these Pixblit routines. These routines read and write pixel values and deal directly with the image data.

The Pixblit routines for the sample server are part of the "mfb" routines (for Monochrome Frame Buffer), and "cfb" routines (for Color Frame Buffer). As with the mi routines, the mfb and cfb routines are portable but are not as portable as the mi routines.

The mfb routines only work for monochrome frame buffers, the simplest type of display. Furthermore, they only work for screens that organize their bits in rows of pixels on the screen. (See the Strategies document for more details on porting mfb.) The cfb routines work for packed-pixel displays from 2 to 32 bits in depth, although they have a bit of code which has been tuned to run on 8-bit (1 pixel per byte) displays.

In other words, if you have a "normal" frame buffer type display, you can probably use either the mfb or cfb code, and the mi code. If you have a stranger hardware, you will have to supply your own Pixblit routines, but you can use the mi routines on top of them. If you have better ways of doing some of the Drawing Primitive functions, then you may want to supply some of your own Drawing Primitive routines. (Even people who write their own Drawing Primitives save at least some of the mi code for certain special cases that their hardware or library or fancy algorithm does not handle.)

The client, DIX, and the machine-independent routines do not carry the final responsibility of clipping. They all depend upon the Pixblit routines to do their clipping for them. The rule is, if you touch the frame buffer, you clip.

(The higher level routines may decide to clip at a high level, but this is only for increased performance and cannot substitute for bottom-level clipping. For instance, the mi routines, DIX, or the client may decide to check all character strings to be drawn and chop off all characters that would not be displayed. If so, it must retain the character on the edge that is partly displayed so that the Pixblit routines can clip off precisely at the right place.)

To make this easier, all of the reasons to clip can be combined into one region in your ValidateGC procedure. You take this composite clip region with you into the Pixblit routines. (The sample server does this.)

Also, FillSpans() has to apply tile and stipple patterns. The patterns are all aligned to the window origin so that when two people write patches that are contiguous, they will merge nicely. (Really, they are aligned to the patOrg point in the GC. This defaults to (0, 0) but can be set by the client to anything.)

However, the mi routines can translate (relocate) the points from window-relative to screen-relative if desired. If you set the miTranslate field in the GC (set it in the CreateGC or ValidateGC routine), then the mi output routines will translate all coordinates. If it is false, then the coordinates will be passed window-relative. Screens with no hardware translation will probably set miTranslate to TRUE, so that geometry (e.g. polygons, rectangles) can be translated, rather than having the resulting list of scanlines translated; this is good because the list vertices in a drawing request will generally be much smaller than the list of scanlines it produces. Similarly, hardware that does translation can set miTranslate to FALSE, and avoid the extra addition per vertex, which can be (but is not always) important for getting the highest possible performance. (Contrast the behavior of GetSpans, which is not expected to be called as often, and so has different constraints.) The miTranslate field is settable in each GC, if , for example, you are mixing several kinds of destinations (offscreen pixmaps, main memory pixmaps, backing store, and windows), all of which have different requirements, on one screen.

As with other drawing routines, there are fields in the GC to direct higher code to the correct routine to execute for each function. In this way, you can optimize for special cases, for example, drawing solids versus drawing stipples.

The Pixblit routines are broken up into three sets. The Span routines simply fill in rows of pixels. The Glyph routines fill in character glyphs. The PushPixels routine is a three-input bitblt for more sophisticated image creation.

It turns out that the Glyph and PushPixels routines actually have a machine-independent implementation that depends upon the Span routines. If you are really pressed for time, you can use these versions, although they are quite slow.

5.6.1. Span Routines

For these routines, all graphic operations have been reduced to "spans." A span is a horizontal row of pixels. If you can design these routines which write into and read from rows of pixels at a time, you can use the mi routines.

Each routine takes a destination drawable to draw into, a GC to use while drawing, the number of spans to do, and two pointers to arrays that indicate the list of starting points and the list of widths of spans.

```
void pGC->ops->FillSpans(dst, pGC, nSpans, pPoints, pWidths, sorted)
	DrawablePtr dst;
	GCPtr pGC;
	int nSpans;
	DDXPointPtr pPoints;
	int *pWidths;
	int sorted;
```

FillSpans should fill horizontal rows of pixels with the appropriate patterns, stipples, etc., based on the values in the GC. The starting points are in the array at pPoints; the widths are in pWidths. If sorted is true, the scan lines are in increasing y order, in which case you may be able to make assumptions and optimizations.

GC components: alu, clipOrg, clientClip, and fillStyle.

GC mode-dependent components: fgPixel (for fillStyle Solid); tile, patOrg (for fillStyle Tile); stipple, patOrg, fgPixel (for fillStyle Stipple); and stipple, patOrg, fgPixel and bgPixel (for fillStyle OpaqueStipple).

```
void pGC->ops->SetSpans(pDrawable, pGC, pSrc, ppt, pWidths, nSpans, sorted)
        DrawablePtr pDrawable;
        GCPtr pGC;
        char *pSrc;
        DDXPointPtr pPoints;
        int *pWidths;
        int nSpans;
        int sorted;
```

For each span, this routine should copy pWidths bits from pSrc to pDrawable at pPoints using the raster-op from the GC. If sorted is true, the scan lines are in increasing y order. The pixels in pSrc are padded according to the screen's padding rules. These can be used to support interesting extension libraries, for example, shaded primitives. It does not use the tile and stipple.

GC components: alu, clipOrg, and clientClip

The above functions are expected to handle all modifiers in the current GC. Therefore, it is expedient to have different routines to quickly handle common special cases and reload the procedure pointers at validate time, as with the other output functions.

```
void pScreen->GetSpans(pDrawable, wMax, pPoints, pWidths, nSpans)
        DrawablePtr pDrawable;
        int wMax;
        DDXPointPtr pPoints;
        int *pWidths;
        int nSpans;
        char *pDst;
```

For each span, GetSpans gets bits from the drawable starting at pPoints and continuing for pWidths bits. Each scanline returned will be server-scanline padded. The routine can return NULL if memory cannot be allocated to hold the result.

GetSpans never translates -- for a window, the coordinates are already screen-relative. Consider the case of hardware that doesn't do translation: the mi code that calls ddX will translate each shape (rectangle, polygon,. etc.) before scan-converting it, which requires many fewer additions that having GetSpans translate each span does. Conversely, consider hardware that does translate: it can set its translation point to (0, 0) and get each span, and the only penalty is the small number of additions required to translate each shape being scan-converted by the calling code. Contrast the behavior of FillSpans and SetSpans (discussed above under miTranslate), which are expected to be used more often.

Thus, the penalty to hardware that does hardware translation is negligible, and code that wants to call GetSpans() is greatly simplified, both for extensions and the machine-independent core implementation.

5.6.1.1. Glyph Routines

The Glyph routines draw individual character glyphs for text drawing requests.

You have a choice in implementing these routines. You can use the mi versions; they depend ultimately upon the span routines. Although text drawing will work, it will be very slow.

```
void pGC->ops->PolyGlyphBlt(pDrawable, pGC, x, y, nglyph, ppci, pglyphBase)
	DrawablePtr pDrawable;
	GCPtr pGC;
	int x , y;
	unsigned int nglyph;
	CharInfoRec **ppci;		/* array of character info */
	pointer unused;			/* unused since R5 */
```

GC components: alu, clipOrg, clientClip, font, and fillStyle.

GC mode-dependent components: fgPixel (for fillStyle Solid); tile, patOrg (for fillStyle Tile); stipple, patOrg, fgPixel (for fillStyle Stipple); and stipple, patOrg, fgPixel and bgPixel (for fillStyle OpaqueStipple).

```
void pGC->ops->ImageGlyphBlt(pDrawable, pGC, x, y, nglyph, ppci, pglyphBase)
	DrawablePtr pDrawable;
	GCPtr pGC;
	int x , y;
	unsigned int nglyph;
	CharInfoRec **ppci;	/* array of character info */
	pointer unused;		/* unused since R5 */
```

GC components: clipOrg, clientClip, font, fgPixel, bgPixel

These routines must copy the glyphs defined by the bitmaps in pglyphBase and the font metrics in ppci to the DrawablePtr, pDrawable. The poly routine follows all fill, stipple, and tile rules. The image routine simply blasts the glyph onto the glyph's rectangle, in foreground and background colors.

More precisely, the Image routine fills the character rectangle with the background color, and then the glyph is applied in the foreground color. The glyph can extend outside of the character rectangle. ImageGlyph() is used for terminal emulators and informal text purposes such as button labels.

The exact specification for the Poly routine is that the glyph is painted with the current fill style. The character rectangle is irrelevant for this operation. PolyText, at a higher level, includes facilities for font changes within strings and such; it is to be used for WYSIWYG word processing and similar systems.

Both of these routines must clip themselves to the overall clipping region.

Example implementations in mi are miPolyGlyphBlt() and miImageGlyphBlt() in Xserver/mi/miglblt.c.

5.6.1.2. PushPixels routine

The PushPixels routine writes the current fill style onto the drawable in a certain shape defined by a bitmap. PushPixels is equivalent to using a second stipple. You can thing of it as pushing the fillStyle through a stencil. PushPixels is not used by any of the mi rendering code, but is used by the mi software cursor code.

Suppose the stencil is:	00111100
and the stipple is:	10101010
PushPixels result:	00101000

You have a choice in implementing this routine. You can use the mi version which depends ultimately upon FillSpans(). Although it will work, it will be slow.

```
void pGC->ops->PushPixels(pGC, pBitMap, pDrawable, dx, dy, xOrg, yOrg)
	GCPtr pGC;
	PixmapPtr pBitMap;
	DrawablePtr pDrawable;
```

```
int dx, dy, xOrg, yOrg;
```

GC components: alu, clipOrg, clientClip, and fillStyle.

GC mode-dependent components: fgPixel (for fillStyle Solid); tile, patOrg (for fillStyle Tile); stipple, patOrg, fgPixel (for fillStyle Stipple); and stipple, patOrg, fgPixel and bgPixel (for fillStyle OpaqueStipple).

PushPixels applys the foreground color, tile, or stipple from the pGC through a stencil onto pDrawable. pBitMap points to a stencil (of which we use an area dx wide by dy high), which is oriented over the drawable at xOrg, yOrg. Where there is a 1 bit in the bitmap, the destination is set according to the current fill style. Where there is a 0 bit in the bitmap, the destination is left the way it is.

This routine must clip to the overall clipping region.

An Example implementation is miPushPixels() in Xserver/mi/mipushpxl.c.

5.7. Shutdown Procedures

```
void AbortDDX()
void ddxGiveUp()
```

Some hardware may require special work to be done before the server exits so that it is not left in an intermediate state. As explained in the OS layer, FatalError() will call AbortDDX() just before terminating the server. In addition, ddxGiveUp() will be called just before terminating the server on a "clean" death. What AbortDDX() and ddxGiveUP do is left unspecified, only that stubs must exist in the ddx layer. It is up to local implementors as to what they should accomplish before termination.

5.7.1. Command Line Procedures

```
int ddxProcessArgument(argc, argv, i)
    int argc;
    char *argv[];
    int i;

void
ddxUseMsg()
```

You should write these routines to deal with device-dependent command line arguments. The routine ddxProcessArgument() is called with the command line, and the current index into argv; you should return zero if the argument is not a device-dependent one, and otherwise return a count of the number of elements of argv that are part of this one argument. For a typical option (e.g., "-realtime"), you should return the value one. This routine gets called before checks are made against device-independent arguments, so it is possible to peek at all arguments or to override device-independent argument processing. You can document the device-dependent arguments in ddxUseMsg(), which will be called from UseMsg() after printing out the device-independent arguments.

5.8. Wrappers and devPrivates

Two new extensibility concepts have been developed for release 4, Wrappers and devPrivates. These replace the R3 GCInterest queues, which were not a general enough mechanism for many extensions and only provided hooks into a single data structure.

5.8.1. devPrivates

devPrivates are arrays of values attached to various data structures (Screens, GCs, Windows, and Pixmaps currently). These arrays are sized dynamically at server startup (and reset) time as various modules allocate portions of them. They can be used for any purpose; each array entry is actually a union, DevUnion, of common useful types (pointer, long and unsigned long). devPrivates must be allocated on startup and whenever the server resets. To make this easier, the global variable "serverGeneration" is incremented each time devPrivates should be allocated, but before the initialization process begins, typical usage would be:
static int privateGeneration = 0;

```
	if (privateGeneration != serverGeneration)
	{
		allocate devPrivates here.

		privateGeneration = serverGeneration;
	}
```

5.8.1.1. Screen devPrivates

An index into every screen devPrivates array is allocated with

```
	int AllocateScreenPrivateIndex()
```

This call can occur at any time, each existing devPrivates array is resized to accommodate the new entry. This routine returns -1 indicating an allocation failure. Otherwise, the return value can be used to index the array of devPrivates on any screen:

```
	private = (PrivatePointer) pScreen->devPrivates[screenPrivateIndex].ptr;
```

The pointer in each screen is not initialized by AllocateScreenPrivateIndex().

5.8.1.2. Window devPrivates

An index into every window devPrivates array is allocated with

```
	int AllocateWindowPrivateIndex ()
```

AllocateWindowPrivateIndex() never returns an error. This call must be associated with a call which causes a chunk of memory to be automatically allocated and attached to the devPrivate entry on every screen which the module will need to use the index:

```
	Bool AllocateWindowPrivate (pScreen, index, amount)
		ScreenPtr pScreen;
		int index;
		unsigned amount;
```

If this space is not always needed for every object, use 0 as the amount. In this case, the pointer field of the entry in the devPrivates array is initialized to NULL. This call exists so that DIX may preallocate all of the space required for an object with one call; this reduces memory fragmentation considerably. AllocateWindowPrivate returns FALSE on allocation failure. Both of these calls must occur before any window structures are allocated; the server is careful to avoid window creation until all modules are initialized, but do not call this after initialization. A typical allocation sequence for WindowPrivates would be:

```
   privateInitialize (pScreen)
	ScreenPtr pScreen;
   {
	if (privateGeneration != serverGeneration)
	{
	   windowPrivateIndex = AllocateWindowPrivateIndex();
	   privateGeneration = serverGeneration;
```

```
    }

    return (AllocateWindowPrivate(pScreen, windowPrivateIndex,
                       sizeof(windowPrivateStructure)));
}
```

5.8.1.3. GC and Pixmap devPrivates

The calls for GCs and Pixmaps mirror the Window calls exactly; they have the same requirements and limitations:

```
int AllocateGCPrivateIndex ()

Bool AllocateGCPrivate (pScreen, index, amount)
    ScreenPtr pScreen;
    int index;
    unsigned amount;

int AllocatePixmapPrivateIndex ()

Bool AllocatePixmapPrivate (pScreen, index, amount)
    ScreenPtr pScreen;
    int index;
    unsigned amount;
```

5.8.2. Wrappers

Wrappers are not a body of code, nor an interface spec. They are, instead, a technique for hooking a new module into an existing calling sequence. There are limitations on other portions of the server implementation which make using wrappers possible; limits on when specific fields of data structures may be modified. They are intended as a replacement for GCInterest queues, which were not general enough to support existing modules; in particular software cursors and backing store both needed more control over the activity. The general mechanism for using wrappers is:

```
privateWrapperFunction (object, ...)
    ObjectPtr    object;
{
    pre-wrapped-function-stuff ...

    object->functionVector = (void *) object->devPrivates[privateIndex].ptr;
    (*object->functionVector) (object, ...);
    /*
     * this next line is occasionally required by the rules governing
     * wrapper functions.  Always using it will not cause problems.
     * Not using it when necessary can cause severe troubles.
     */
    object->devPrivates[privateIndex].ptr = (pointer) object->functionVector;
    object->functionVector = privateWrapperFunction;

    post-wrapped-function-stuff ...
}

privateInitialize (object)
    ObjectPtr    object;
{
    object->devPrivates[privateIndex].ptr = (pointer) object->functionVector;
    object->functionVector = privateWrapperFunction;
}
```

Thus the privateWrapperFunction provides hooks for performing work both before and after the wrapped function has been called; the process of resetting the functionVector is called "unwrapping" while the process of fetching the wrapped function and replacing it with the wrapping function is called "wrapping". It should be clear that GCInterest queues could be emulated using wrappers. In general, any function vectors contained in objects can be wrapped, but only vectors in GCs and Screens have been tested.

Wrapping screen functions is quite easy; each vector is individually wrapped. Screen functions are not supposed to change after initialization, so rewrapping is technically not necessary, but causes no problems.

Wrapping GC functions is a bit more complicated. GC's have two sets of function vectors, one hanging from gc->ops and the other from gc->funcs. Wrappers should modify only those values, not the internal values as they may be shared by more than one GC (and, in the case of funcs, are probably shared by all gcs). To wrap the ops, wrap the funcs and, in each func wrapper, unwrap the ops and funcs, call down, and re-wrap. In each op wrapper, unwrap both the funcs and ops, call down and rewrap afterwards. The rule is: if you wrap funcs+ops, you must always unwrap both before down calling. If you wrap ops, you must always pull the ops value out of the GC in the func wrapper and save it. If you wrap funcs, you needn't pull the funcs value out of the GC to rewrap as the func values are required to be constant. In this way, the wrapped validation routine can change the op vector and not have it lost when your wrapper routine rewraps the GC. This occurs when the wrapped op revalidates the GC with new entries (many mi routines do this for opaque stipples or double dashes). The corollary to this rule is: Never change the func vector after CreateGC.

5.9. Work Queue

To queue work for execution when all clients are in a stable state (i.e. just before calling select() in WaitForSomething), call:

```
Bool QueueWorkProc(function,client,closure)
        Bool        (*function)();
        ClientPtr   client;
        pointer         closure;
```

When the server is about to suspend itself, the given function will be executed:

```
(*function) (client, closure)
```

Neither client nor closure are actually used inside the work queue routines.

6. Extension Interfaces

This section describes the functions which exist in DDX for extension writers to use.

6.1. Extension initialization

This function should be called from your extensionInitProc which should be called by InitExtensions.

```
ExtensionEntry *AddExtension(name, NumEvents,NumErrors,
        MainProc, SwappedMainProc, CloseDownProc, MinorOpcodeProc)

        char *name;  /*Null terminate string; case matters*/
        int NumEvents;
        int NumErrors;
        int (* MainProc)(ClientPtr);/*Called if client matches server order*/
        int (* SwappedMainProc)(ClientPtr);/*Called if client differs from server*/
        void (* CloseDownProc)(ExtensionEntry *);
        unsigned short (*MinorOpcodeProc)(ClientPtr);
```

name is the name used by clients to refer to the extension. NumEvents is the number of event types used by the extension, NumErrors is the number of error codes needed by the extension. MainProc is called whenever a client accesses the major opcode assigned to the extension. SwappedMainProc is identical, except the client using the extension has reversed byte-sex. CloseDownProc is called at server reset time to deallocate any private storage used by the extension. MinorOpcodeProc is used by DIX to place the appropriate value into errors. The DIX routine StandardMinorOpcode can be used here which takes the minor opcode from the normal place in the request (i.e. just after the major opcode).

6.2. Resource type allocation.

These functions should also be called from your extensionInitProc to allocate all of the various resource classes and types required for the extension. Each time the server resets, these types must be reallocated as the old allocations will have been discarded. Resource types are integer values starting at 1. Get a resource type by calling

```
RESTYPE CreateNewResourceType(deleteFunc)
```

deleteFunc will be called to destroy all resources with this type.

Resource classes are masks starting at 1 << 31 which can be or'ed with any resource type to provide attributes for the type. To allocate a new class bit, call

```
RESTYPE CreateNewResourceClass()
```

There are two ways of looking up resources, by type or by class. Classes are non-exclusive subsets of the space of all resources, so you can lookup the union of multiple classes. (RC_ANY is the union of all classes).

Note that the appropriate class bits must be or'ed into the value returned by CreateNewResourceType when calling resource lookup functions.

If you need to create a ''private'' resource ID for internal use, you can call FakeClientID.

```
XID FakeClientID(client)
    int client;
```

This allocates from ID space reserved for the server.

To associate a resource value with an ID, use AddResource.

```
Bool AddResource(id, type, value)
    XID id;
    RESTYPE type;
    pointer value;
```

The type should be the full type of the resource, including any class bits. If AddResource fails to allocate memory to store the resource, it will call the deleteFunc for the type, and then return False.

To free a resource, use one of the following.

```
void FreeResource(id, skipDeleteFuncType)
    XID id;
    RESTYPE skipDeleteFuncType;

void FreeResourceByType(id, type, skipFree)
    XID id;
    RESTYPE type;
    Bool    skipFree;
```

FreeResource frees all resources matching the given id, regardless of type; the type's deleteFunc will be called on each matching resource, except that skipDeleteFuncType can be set to a single type for which the deleteFunc should not be called (otherwise pass RT_NONE). FreeResourceByType frees a specific resource matching a given id and type; if skipFree is true, then the deleteFunc is not called.

To look up a resource, use one of the following.

```
pointer LookupIDByType(id, rtype)
    XID id;
    RESTYPE rtype;

pointer LookupIDByClass(id, classes)
    XID id;
    RESTYPE classes;
```

LookupIDByType finds a resource with the given id and exact type. LookupIDByClass finds a resource with the given id whose type is included in any one of the specified classes.

6.3. Macros and Other Helpers

There are a number of macros in Xserver/include/dix.h which are useful to the extension writer. Ones of particular interest are: REQUEST, REQUEST_SIZE_MATCH, REQUEST_AT_LEAST_SIZE, REQUEST_FIXED_SIZE, LEGAL_NEW_RESOURCE, LOOKUP_DRAWABLE, VERIFY_GC, and VALIDATE_DRAWABLE_AND_GC. Useful byte swapping macros can be found in Xserver/include/misc.h: lswapl, lswaps, LengthRestB, LengthRestS, LengthRestL, SwapRestS, SwapRestL, swapl, swaps, cpswapl, and cpswaps.

7. Callback Manager

To satisfy a growing number of requests for the introduction of ad hoc notification style hooks in the server, a generic callback manager was introduced in R6. A callback list object can be introduced for each new hook that is desired, and other modules in the server can register interest in the new callback list. The following functions support these operations.

Before getting bogged down in the interface details, an typical usage example should establish the framework. Let's look at the ClientStateCallback in dix/dispatch.c. The purpose of this particular callback is to notify intereseted parties when a client's state (initial, running, gone) changes. The callback is "created" in this case by simply declaring a variable:

```
CallbackListPtr ClientStateCallback;
```

Whenever the client's state changes, the following code appears, which notifies all intereseted parties of the change:

```
if (ClientStateCallback) CallCallbacks(&ClientStateCallback, (pointer)client);
```

Interested parties subscribe to the ClientStateCallback list by saying:

```
AddCallback(&ClientStateCallback, func, data);
```

When CallCallbacks is invoked on the list, func will be called thusly:

```
(*func)(&ClientStateCallback, data, client)
```

Now for the details.

```
Bool CreateCallbackList(pcbl, cbfuncs)
    CallbackListPtr  *pcbl;
    CallbackFuncsPtr cbfuncs;
```

CreateCallbackList creates a callback list. We envision that this function will be rarely used because the callback list is created automatically (if it doesn't already exist) when the first call to AddCallback is made on the list. The only reason to explicitly create the callback list with this function is if you want to override the implementation of some of the other operations on the list by passing your own cbfuncs. You also lose something by explicit creation: you introduce an order dependency during server startup because the list must be created before any modules subscribe to it. Returns TRUE if successful.

```
Bool AddCallback(pcbl, callback, subscriber_data)
    CallbackListPtr *pcbl;
    CallbackProcPtr callback;
    pointer         subscriber_data;
```

Adds the (callback, subscriber_data) pair to the given callback list. Creates the callback list if it doesn't exist. Returns TRUE if successful.

```
Bool DeleteCallback(pcbl, callback, subscriber_data)
    CallbackListPtr *pcbl;
    CallbackProcPtr callback;
    pointer         subscriber_data;
```

Removes the (callback, data) pair to the given callback list if present. Returns TRUE if (callback, data) was found.

```
void CallCallbacks(pcbl, call_data)
   CallbackListPtr    *pcbl;
   pointer            call_data;
```

For each callback currently registered on the given callback list, call it as follows:

```
(*callback)(pcbl, subscriber_data, call_data);
```

```
void DeleteCallbackList(pcbl)
   CallbackListPtr    *pcbl;
```

Destroys the given callback list.

8. Summary of Routines

This is a summary of the routines discussed in this document. The procedure names are in alphabetical order. The Struct is the structure it is attached to; if blank, this procedure is not attached to a struct and must be named as shown. The sample server provides implementations in the following categories. Notice that many of the graphics routines have both mi and mfb implementations.

dix	portable to all systems; do not attempt to rewrite (Xserver/dix)
os	routine provided in Xserver/os or Xserver/include/os.h
ddx	frame buffer dependent (examples in Xserver/mfb,Xserver/cfb)
mi	routine provided in Xserver/mi
hd	hardware dependent (examples in many Xserver/hw directories)
none	not implemented in sample implementation

Procedure	Port	Struct
ALLOCATE_LOCAL	os	
AbortDDX	hd	
AddCallback	dix	
AddEnabledDevice	os	
AddInputDevice	dix	
AddScreen	dix	
Bell	hd	Device
ChangeClip	mi	GC func
ChangeGC		GC func
ChangeWindowAttributes	ddx	Screen
ClearToBackground	ddx	Window
ClientAuthorized	os	
ClientSignal	dix	
ClientSleep	dix	
ClientWakeup	dix	
ClipNotify	ddx	Screen
CloseScreen	hd	
ConstrainCursor	hd	Screen
CopyArea	mi	GC op
CopyGCDest	ddx	GC func
CopyGCSource	none	GC func
CopyPlane	mi	GC op
CopyWindow	ddx	Window
CreateGC	ddx	Screen
CreateCallbackList	dix	
CreatePixmap	ddx	Screen
CreateScreenResources	ddx	Screen
CreateWellKnowSockets	os	
CreateWindow	ddx	Screen
CursorLimits	hd	Screen
DEALLOCATE_LOCAL	os	
DeleteCallback	dix	
DeleteCallbackList	dix	
DestroyClip	ddx	GC func
DestroyGC	ddx	GC func
DestroyPixmap	ddx	Screen
DestroyWindow	ddx	Screen
DisplayCursor	hd	Screen
Error	os	
ErrorF	os	

Procedure	Port	Struct
FatalError	os	
FillPolygon	mi	GC op
FillSpans	ddx	GC op
FlushAllOutput	os	
FlushIfCriticalOutputPending	os	
FreeScratchPixmapHeader	dix	
GetImage	mi	Screen
GetMotionEvents	hd	Device
GetScratchPixmapHeader	dix	
GetSpans	ddx	Screen
GetStaticColormap	ddx	Screen
ImageGlyphBlt	mi	GC op
ImageText16	mi	GC op
ImageText8	mi	GC op
InitInput	hd	
InitKeyboardDeviceStruct	dix	
InitOutput	hd	
InitPointerDeviceStruct	dix	
InsertFakeRequest	os	
InstallColormap	ddx	Screen
Intersect	mi	Screen
Inverse	mi	Screen
LegalModifier	hd	
LineHelper	mi	GC op
ListInstalledColormaps	ddx	Screen
LookupKeyboardDevice	dix	
LookupPointerDevice	dix	
ModifyPixmapheader	mi	Screen
NextAvailableClient	dix	
OsInit	os	
PaintWindowBackground	mi	Window
PaintWindowBorder	mi	Window
PointerNonInterestBox	hd	Screen
PointInRegion	mi	Screen
PolyArc	mi	GC op
PolyFillArc	mi	GC op
PolyFillRect	mi	GC op
PolyGlyphBlt	mi	GC op
Polylines	mi	GC op
PolyPoint	mi	GC op
PolyRectangle	mi	GC op
PolySegment	mi	GC op
PolyText16	mi	GC op
PolyText8	mi	GC op
PositionWindow	ddx	Screen
ProcessInputEvents	hd	
PushPixels	mi	GC op
PutImage	mi	GC op
QueryBestSize	hd	Screen
ReadRequestFromClient	os	
RealizeCursor	hd	Screen

Procedure	Port	Struct
RealizeFont	ddx	Screen
RealizeWindow	ddx	Screen
RecolorCursor	hd	Screen
RectIn	mi	Screen
RegionCopy	mi	Screen
RegionCreate	mi	Screen
RegionDestroy	mi	Screen
RegionEmpty	mi	Screen
RegionExtents	mi	Screen
RegionNotEmpty	mi	Screen
RegionReset	mi	Screen
ResolveColor	ddx	Screen
RegisterKeyboardDevice	dix	
RegisterPointerDevice	dix	
RemoveEnabledDevice	os	
ResetCurrentRequest	os	
RestoreAreas	none	BackingStore
SaveDoomedAreas	none	BackingStore
SaveScreen	ddx	Screen
SetCriticalOutputPending	os	
SetCursorPosition	hd	Screen
SetInputCheck	dix	
SetSpans	ddx	GC op
StoreColors	ddx	Screen
Subtract	mi	Screen
TimeSinceLastInputEvent	hd	
TranslateBackingStore	none	BackingStore
TranslateRegion	mi	Screen
UninstallColormap	ddx	Screen
Union	mi	Screen
UnrealizeCursor	hd	Screen
UnrealizeFont	ddx	Screen
UnrealizeWindow	ddx	Screen
ValidateGC	ddx	GC func
ValidateTree	mi	Screen
WaitForSomething	os	
WindowExposures	mi	Window
WriteToClient	os	
Xalloc	os	
Xfree	os	
Xrealloc	os	

Table of Contents