CHAPTER 14 DDX Issues

14.1 Cursor Management

Both software and hardware cursors must be considered in the design of the MTX server, although our final preference will be to use hardware cursors. The hardware cursor design uses either the overlay-plane when drawing the cursor, or uses a hardware function to draw the cursor. So, erasing the cursor is not needed before rendering to a window as is required for software cursors. Because cursor drawing is hardware dependent, these functions are located in the DDX layer. Software cursor locking is enabled in MTX by setting the -DUSE_SOFTWARE_CURSOR flag in the build environment.

There are three ways that the cursor is affected:

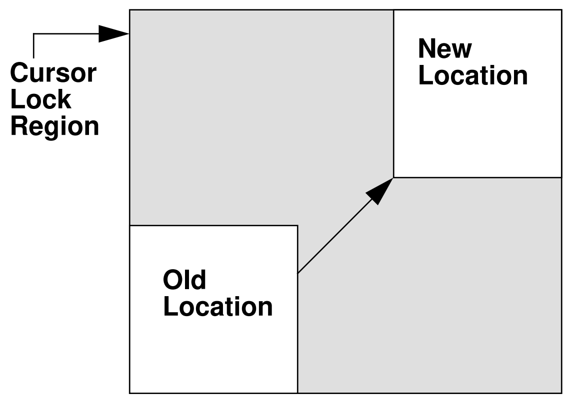
1. Move the location of the visible cursor in response to pointer movement.
2. When drawing into the window that contains the cursor, erase the cursor. This is known as *Cursor faulting* and is performed only in the software cursor case.
3. Change the cursor's shape, color or location (usually at the client's request).

In the MTX, the DIT handles the first case, and the CIT is responsible for the last two. Because multiple threads may modify the cursor data, exclusive access is required.This section describes how exclusive control of the cursor data is managed.

14.1.1 Software cursor

Software cursor conflicts are resolved using the POQ. Operations that modify the cursor need to set the CM_X_CURSOR bit in the POQ's conflict mask. These operations include the DIT changing the location of the cursor as well as requests issued by CITS to modify attributes of the cursor (such as changing the shape, color, or location). When using the software cursor mechanism, the CM_X_RENDER bit on the POQ will also conflict with a CM_X_CURSOR bit. This enables a render request to test for cursor overlap without having to acquire any additional locks. If a conflict is detected, it is resolved by determining whether the render region and the *cursor lock region* intersect.

The following diagram shows how the *cursor lock region* is defined. The *cursor lock region* is calculated only when a cursor conflict has been detected.



There are four possible cursor conflict cases, each is described in detail below:

- CM_X_CURSOR => CM_X_CURSOR
- CM_X_CURSOR => CM_X_RENDER
- CM_X_RENDER => CM_X_CURSOR
- CM_X_RENDER => CM_X_RENDER

In the CM_X_CURSOR => CM_X_CURSOR case, the conflict is absolute, therefore no *cursor lock region* needs to be generated. This occurs, for example, when the DIT is moving the cursor at the same time a CIT is warping the cursor. Only one thread may change the cursor location or attributes at a time. The other must wait. Only requests that alter a cursor's attributes or location need to set the CM_X_CURSOR bit. CM_X_CURSOR conflicts are detected prior to CM_X_RENDER conflicts on the POQ. This is done to avoid generating a *cursor lock region* unnecessarily.

In the case of a CM_X_CURSOR => CM_X_RENDER conflict, a *cursor lock region* is generated to determine whether the cursor request conflicts with the render request. If so, then the cursor request blocks while the render completes. Since the cursor request is suspended, the cursor location cannot be moved until the render is complete. The render request is free to test for cursor overlap and fault the cursor without fear of the cursor location changing. If the regions do not conflict, both operations may proceed simultaneously. In this case, there is a chance that the rendering thread may read a cursor location value that is being changed by another thread. On an architecture where writes to memory are atomic, this is not a problem because it will read either the new or old location of the cursor, and not some arbitrary location. This is acceptable because the *cursor lock region* has already determined if a cursor overlap exists and this region is known before the cursor is actually moved.

In the case of a CM_X_RENDER => CM_X_CURSOR conflict, the above is still true, except it is the render request that is suspended until the cursor request completes. Since the render request is suspended, the cursor request is free to change the location or attributes of the cursor without fear of another thread reading cursor data while it is being modified. If the regions do not conflict, both operations proceed in parallel.

In the case of a CM_X_RENDER => CM_X_RENDER conflict, the normal region conflict mechanism is used to determine if a conflict exists between the two operations.

When the cursor glyph is moved, changed, or faulted, an additional mutex is required to prevent multiple CITs from accessing the cursor glyph simultaneously. This mutex is acquired only when the cursor glyph is drawn or undrawn.

The advantages of this scheme are:

- All DIX changes are localized to the POQ.
- Requires no additional locking for render requests when testing for cursor overlap. This optimizes for the benchmark case (i.e. when the cursor is not in motion).
- It allows the DIT to move the cursor at the same time a CIT is rendering, provided their regions do not conflict.

Note:

- On non-cache coherent architectures, a render request will need to acquire the cursor mutex before testing for cursor overlap. Thus, concurrency is reduced in this case.

14.1.2 Hardware cursor

Before the hardware cursor is accessed, the cursor mutex will be locked. In the hardware cursor case, there is no need to erase the cursor before drawing into a window. Therefore, render requests do not test for cursor overlap or fault the cursor.

- The following algorithm is used when a hardware cursor is used.

1. Lock the cursor mutex.
2. Access the data for the hardware cursor.
3. Unlock the cursor mutex.

14.2 Reentrancy

The following sub-sections describe those areas of the R5 DDX that are not reentrant, and how MTX will solve the reentrancy problems.

14.2.1 miPolyArc

miPolyArc uses a global cache to save previously generated arc span data for circles and ellipses whose line width is non-zero and whose fill style is solid. Using a cache, the arc span data can be used without re-calculation whenever there is a cache hit. (This is especially efficient for x11perf.) In general, if the arc is large, the time to draw the arc is relatively long because drawing into the frame buffer is slow. This tends to negate the effect of the cache.

Since caching increases performance, MTX will use this type of cache mechanism as well. There are two cache implementation methods. The first method allows all CITs to share one cache through an exclusive lock. In the second method, each CIT has its own cache. The first method saves cache space while using an exclusive lock. The second method has no locking, but increases concurrency and speed at the cost of higher memory utilization.

The following describes the first of these methods since it is the one implemented by MTX. Every CIT shares a single cache. Access to the cache is through a single mutex, MiArcMutex.

Algorithm:

1. Lock miArcMutex.
2. Create the arc span data and transfer it to span data.
3. Unlock miArcMutex.
4. Execute *FillSpan* using span data.

14.2.2 miPaintWindow

When drawing the background of the root window, once the GC is created, the pointer to the GC is saved in a static variable to reduce drawing time. A new resource is added using CreateNewResourceType() and AddResource() to free the GC created when the server is reset.

In order to eliminate this global variable and make miPaintwindow reentrant, MTX will create and free the GC every time. Even though drawing speed is decreased, this is acceptable because miPaintwindow is rarely (if ever) used.

14.2.3 cfb Stipple (only when PPW == 4)

The following DDX functions are not reentrant because they share global variables.

- cfbCopyPlane
- cfbUnnaturalStippleFS
  cfb8OpaqueStipple32FS
  cfb8Stipple32FS
- cfb8FillRectOpaqueStipple32FS
  cfb8FillRectTransparentStipple32FS
  cfb8FillRectStipple32FS
- cfbPolyGlyphRop8
  cfbPolyGlyphRop8Clipped
- cfbTEGlyphBlt8

The previous functions convert stipple bits to pixel-fill bits and draw to a drawable. This is called *stipple pattern unfolding*. These functions create a pattern-unfolding array to make unfolding faster. The data depends on the foreground pixel, the background pixel, the plane-mask, and the drawing function. To reduce time spent re-creating the array, it is saved in a global storage area, and re-used the next time a stipple requests is issued if the data has not changed.

In the R5 implementation, the following global variables are used to hold the latest contents of the foreground pixel, the background pixel, the plane-mask, the drawing-function, and the pattern-unfolding array.

- cfb8StippleAnd[16]
  cfb8StippleXor[16]
- cfb8StippleAlu
  cfb8StippleBg
  cfb8StippleFg
  cfb8StippleMode
  cfb8StipplePm
  cfb8StippleRRop

The MTX implementation of cfb stipple code will not store these as global variables. Instead, they will be stored in the cfbPrivGC. Since the GC is exclusively protected, no additional locking is required to protect the stipple data.

14.2.3.1 Implementation

The pattern-unfolding array and data are stored in the private area of the GC. The pattern-unfolding array is created when validating the GC. When the special functions *cfbCopyPlane* and *cfbTEGlyphBlt8* are called, the array data is re-created.

NOTE: *cfbCopyPlane* draws the bitmap data using OpaqueStipple unfolding, regardless of the GC's FillStyle, hence the data and pattern-unfolding array created during GC validation cannot be used. There are also cases when data isn't stored. (Maybe we should consider creating this data when rendering rather than at GC validation time.)

The pattern-unfolding data to be saved is as follows:

```
typedef struct _Stipple {
        unsigned int      change;
        int               cfb8StippleMode;
        int               cfb8StippleAlu;
        int               cfb8StippleRRop;
        unsigned long     cfb8StippleFg;
        unsigned long     cfb8StippleBg;
        unsigned long     cfb8StipplePm;
        unsigned long     cfb8StippleXor[16];
        unsigned long     cfb8StippleAnd[16];
} StippleRec;
```

The GC private information structure:

```
typedef struct {
                          .
                          .
                          .
        struct _Stipple   *stipple;
} cfbPrivGC;
```

Pattern functions are cfb8SetStipple(), cfb8SetOpaqueStipple().

Algorithm:

1. cfbCreateGC() sets cfbPrivGC.stipple to NULL
2. build pattern unfolding array at GC validation for each of the following cases:
   - FillStyle is changed to Stipple/OpaqueStipple.
   - Either alu/fg/planemask is changed while FillStyle is Stipple.
   - alu/fg/bg/planemask is changed while FillStyle is OpaqueStipple.
   If the stipple of cfbPrivGC is NULL, create new data using the pattern function. If not NULL, update the data using the pattern function.
   Set StippleRec.change to FALSE.
3. Do the following whenever the drawing function that uses the pattern-unfolding array is called:
   For normal drawing functions - Draw using StippleRec information when the StippleRec.change is FALSE. If StippleRec.change is TRUE, compare the drawing attributes of the GC and StippleRec. If they are equal set StippleRec.change to FALSE, and draw using StippleRec. If not equal, set StippleRec.change to FALSE, and draw after updating with the pattern function.
   Special drawing functions (*cfbCopyPlane* and *cfbTEGlyphBlt8*)- If stipple of cfbPrivGC is NULL, create a new StippleRec using the pattern function. Set StippleRec.change to TRUE, and draw. If stipple of cfbPrivGC is not NULL, compare the drawing attributes with the StippleRec. If they are equal, draw using StippleRec. If they are not equal, set StippleRec.change to TRUE, and draw after updating with the pattern function.
4. In cfbDestroyGC(), if cfbPrivGC stipple is not NULL, free the stipple area.

14.2.4 NEXT_SERIAL_NUMBER

NEXT_SERIAL_NUMBER is used when validating a GC with a drawable. The value of the global serial number is changed in the NEXT_SERIAL_NUMBER macro. This implies that the following functions that use the NEXT_SERIAL_NUMBER macro are not reentrant:

- mi
  - miResizeBackingStore
  - miDestroyBSPixmap
  - miPaintWindow
  - miValidateTree
- mfb
  - mfbPutImage
  - mfbGetImage
  - mfbCreatePixmap
  - mfbCopyPixmap
  - mfbCopyRotatePixmap
  - mfbPositionWindow
- cfb
  - cfbPutImage
  - cfbGetImage
  - cfbCreatePixmap
  - cfbCopyPixmap
  - cfbCopyRotatePixmap

There are also DIX routines that use the NEXT_SERIAL_NUMBER macro.

In the current R5 design, the global serial number is updated by any function that changes the relationship between a GC and a drawable. In general, a GC is validated only when the following is true:

- if (the serial number of the GC is different from the serial number of the drawable)

```
{
        ValidateGC()
}
```

MTX will still use a serial number, but it will be one serial number per thread instead of the global serial number used in R5.

thread id	thread serial number
31 24	23 0

In MTX, a serial number is associated with each thread rather than a single global serial number. This will allow a thread that needs to reference a serial number to avoid acquiring an additional lock to do so. The per thread serial number is composed of two parts. The first part contains the thread id. This will insure that the per thread serial number is unique across the server. The second part is an increasing number that is unique to that thread, and may be the same among different threads.

The NEXT_SERIAL_NUMBER will be implemented using a thread specific key. Serial number initialization will be performed in the MST and CIT.

14.2.5 Must_have_memory

Xalloc(), Xrealloc(), Xcalloc() are not reentrant because they use the global variable *Must_have_memory*. This indicates how the server will respond if these functions cannot allocate memory. *Must_have_memory* is normally set to FALSE. The following responses are defined in R5:

- Call FatalError and terminate if Must_have_memory == TRUE.
- Return NULL if Must_have_memory == FALSE.

In the DDX layer, the following functions change Must_have_memory to TRUE:

- mi
  - miRecolorCursor
  - miRegionCreate
  - miRectAlloc
  - miRegionCopy
  - miRegionValidate
  - mRectsToRegion
- mfb
  - mfbCreateOps
- cfb
  - cfbCreateOps

MTX will eliminate the global variable *Must_have_memory*. Instead, should require all requests for memory that fail to return a BADALLOC error to the client. This implies that the server will never abort if memory cannot be obtained.

14.3 Render Locking

Locking of the frame buffer is required when *PixelPerWord* is not equal to 1. CHAPTER 10 describes how locking of the frame buffer is implemented using the POQ. The following function is provided to tell the POQ what type of hardware is available on each screen. This will allow the POQ to adjust region calculations accordingly when detecting region conflicts. This function should be called from the DDX function *InitOutput()* after a new screen has been added.

POQSelectRegionConflictType (screenNum, hardwareType)

screenNum is the id number of the screen;
hardwareType is one of the following:

POQ_1_BIT_PER_PIXEL
POQ_8_BITS_PER_PIXEL
POQ_16_BITS_PER_PIXEL
POQ_24_BITS_PER_PIXEL
POQ_32_BITS_PER_PIXEL
POQ_USE_GRAPHICS_ACCEL

See CHAPTER 10 for more details.

CHAPTER 15 Other DIX Issues

15.1 ScreenSaver

There are three ways that screensaver can be affected:

1. If no device input is received during a predetermined interval, the screensaver is turned on. If the screensaver option is BLANK, then the screen is blanked. If the screensaver option is NO-BLANK, a user specified background pattern is displayed.
2. If device input is received, the screensaver is turned off and the timer reset.
3. The screensaver attributes can be modified via the SetScreenSaver and ForceScreenSaver protocol requests.

The CIT will ask the DIT to change the state of the screensaver as the result of a SetScreenSaver or ForceScreenSaver protocol request. The SIGALRM signal handler will be used to tell the DIT whenever the screensaver must be turned on. But, the DIT will actually change the screensaver state (on/off, interval time) at the direction of these two outside tasks. The following screensaver global variables will require exclusive locking (via the POQ CM_R_SCREENSAVER and CM_W_SCREENSAVER bits) since each of these threads can execute concurrently:

ScreenSaverTime
ScreenSaverInterval
ScreenSaverAllowExposures
ScreenSaverBlanking
screenIsSaved
savedScreenInfo

Read lock	- when CIT processes GetScreenSaver
Write lock	- when CIT processes SetScreenSaver
	- when CIT processes ForceScreenSaver
	- when DIT resets screensaver
	- when DIT activates screensaver

lastEventTimeThis
this variable records the time the last input event occurred.

Screensaver blanking is controlled by the blanking/non-blanking switch.

- Non video blanking:
  Lock the screen to prevent other threads from drawing.
  Realize the screen lock using the Pending Operation Queue.
- Video blanking:
  Implementation is machine dependent.
  If the hardware processes screen blanking, then exclusive control is not needed.
  If the server controls the color pallet directly, exclusive control is required.

15.2 Block and Wakeup Handler

In the R5 server, WaitForSomething() is used to wait for any of three occurrences:
a device input event is received
a client request is received
a new connection request is received.

These different occurrences are implemented using the select() mechanism.

In R5, WaitForSomething() calls the BlockHandler before it calls select(). After the select() returns, it calls WakeupHandler(). The block and wakeup handlers are dependent on the machine or operating system.

In MTX, WaitForSomething() does not exist as each of the above three functional occurrences has been implemented via threads:
DIT - a device input event is received
CIT - a client request is received
CCT - a new connection request is received.

The Block and Wakeup Handlers are used to poll device input data, or to redraw the cursor in some R5 DDX implementations. Since MTX can better implement these functions in the existing threads, these handlers are no longer required. The DDX implementor can put machine and operating system dependencies directly into the threads.

15.3 Server Extensions

Most extensions will be initialized in the MST, but dynamically loadable extensions should be handled in the appropriate CIT. See CHAPTER 16 for a more complete discussion.

15.4 PEX

The PEX 5.1 protocol has relaxed request atomicity, in order to allow future threaded PEX implementations. The new words say that

> PEXRenderOutputCommands and PEXRenderNetwork are not guaranteed to be atomic with respect to concurrent rendering to the same destination drawable, except that atomicity of execution is guaranteed for each individual OutputCommand primitive (excluding Execute Structure, GSE, GDP 3D, and GDP 2D)

PEX protocol requests must be re-implemented to mimic the locking model used by core MTX (see CHAPTER 16 for an example).

15.5 Security

Such as Kerberos; Trusted X.

These issues will not be addressed in this version of MTX.

15.6 Priority Threads

Real Time applications may require the ability to set priorities on server threads, or to change priorities of server threads dynamically based on changing conditions in the server.

These issues will not be addressed in this version of MTX.

CHAPTER 16 Server Extension Writers Guidelines

16.1 Introduction

This chapter will describe some of the issues related to writing server extensions in the MTX environment. This chapter may duplicate material in previous chapters, but it was felt beneficial to consolidate these guidelines in one place for extension writers porting to MTX.

The server distinguishes very little between extension requests and the core protocol requests. Although extensions must be explicitly initialized by the server, once the extension is registered, the server receives and processes protocol requests for the extension in exactly the same way that it receives and processes the core protocol requests.

The next figure outlines the steps that an extension writer should follow in building an extension that is safe to operate within MTX. Each step generates information that is used by the next step in the process.

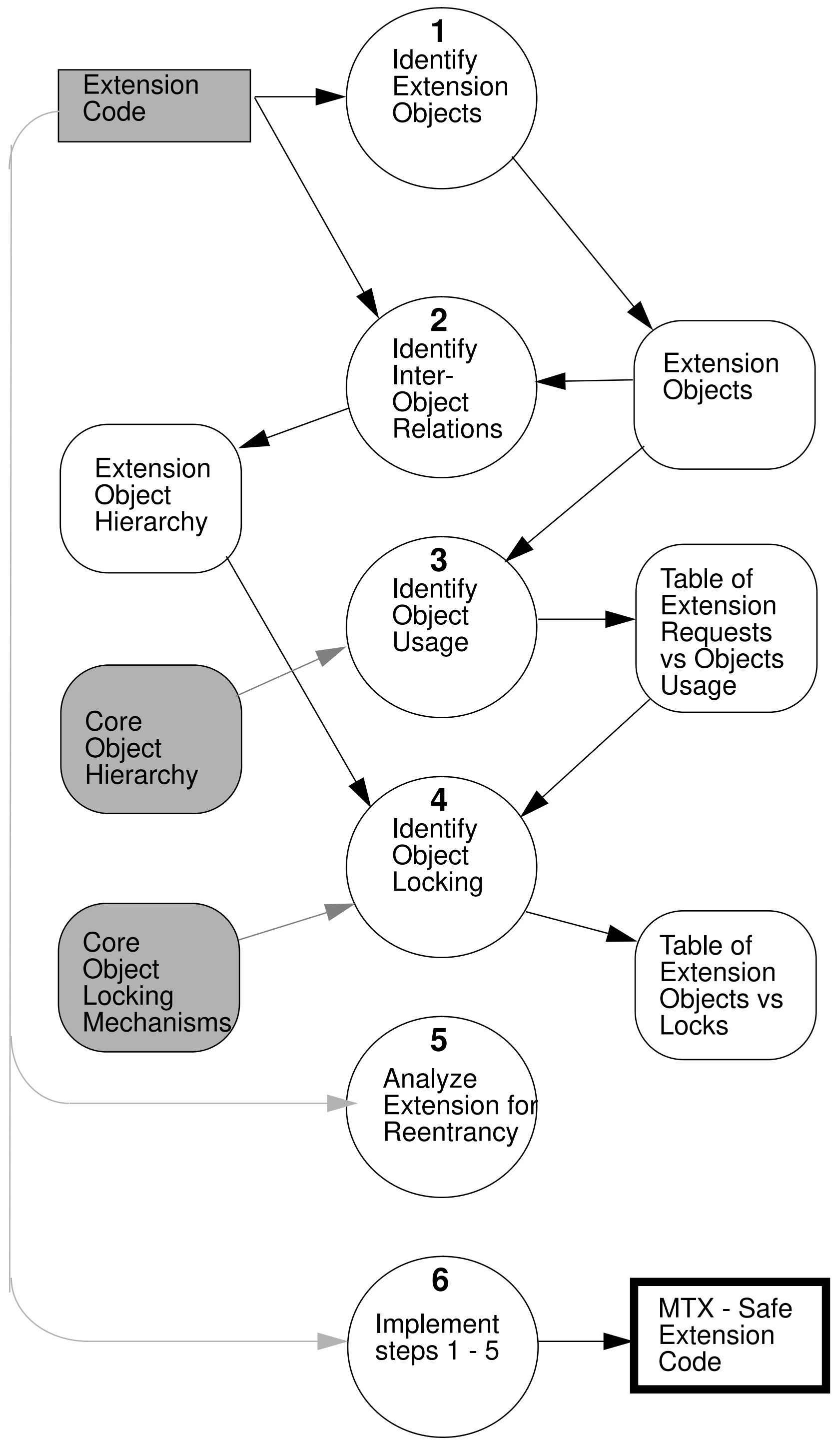


FIGURE 51 Building an MTX-safe extension

The legend for FIGURE 51 is as follows:

- circles indicate the steps to be followed
- round-edged boxes indicate results that are compiled in each step
- the grayed extension code rectangle indicates the initial input to the process
- the grayed round-edged boxes indicate input from the core MTX server

Some terminology:

Reentrant: Code that may be safely executed concurrently by more than one process or thread.

Thread-safe: Code that is reentrant within a threaded environment. Locks and other synchronization mechanisms may be used to enforce reentrancy. References to global and static variables must be removed or protected from mutual access.

MTX-safe: An extension is termed safe for MTX if it is thread-safe and conforms to the MTX locking strategy defined in the previous chapters of this document.

The steps needed to generate an MTX-safe extension are as follows:

1. Identify extension objects from an examination of the extension code.
2. Identify the inter-relationships of the extension objects, and generate an object hierarchy.
3. Identify core and extension objects used by the extension requests. Create a table of extension requests versus objects.
4. Determine object locking requirements in the extension by examining the extension object hierarchy and how extension objects are used in the extension requests. Create a table of objects versus locks that describes how the locking strategy will be implemented.
5. Determine where extension code must be made re-entrant.
6. Upon completion of the analysis of the extension, and the locking specification, generate an MTX-safe extension.

The following sections describe these steps in more detail. Each step will use the PEX extension as an example. Hopefully, this will give you a better idea of how this whole process works if we can apply it to a "real" extension. Note: It is assumed that the reader has some knowledge of the server side PEX-SI (version 5.0).

16.2 Identify Extension Objects

First, the extension writer must identify what new objects the externsion will create that differ from the set of core objects. Although the extension may use core objects, such as windows, GCs, fonts, core object usage by the extension has been defined in the previous locking chapters. By using the existing external interface to the core objects, the extension will adhere to the requirement that it be MTX-safe.

The extension may create new server resource types using *CreateResourceType* and have objects of that type managed by the RDB Monitor. Since the extension can execute in multiple threads, these objects must be listed so that later the correct locking granularity can be determined.

16.2.1 Identify PEX Objects

Resources unique to PEX are:

- **lookup tables** provide a level of indirection for various output primitive attributes. Examples of tables include color tables, line bundle tables, view tables, and light tables.
- **pipeline context** resource is a structure that contains all of the attributes associated with the renderer's pipeline state.
- **renderer** encompasses the process of converting three-dimensional geometric objects into a two-dimensional raster image. Renderers are used in PEX to support immediate mode rendering.
- **structures** stores output commands for later execution. The output commands stored in structures are called structure elements. Structures can reference other structures.
- **name sets** is a list of names that can be used to specify which primitives are eligible for various operations such as highlighting, invisibility, searching, and picking.
- **search contexts** allows a PEX client to designate search parameters and then searcha structure network for the first primitive that fulfills the search attributes.
- **PHIGS workstation** contains all the functionality to support the PHIGS model of a workstation.
- **pick measures** allows a client to specify parameters for selecting output primitives usinga pointing device.
- **PEX fonts** defines an array of characters for a vector font. Unlike bitmapped fonts, PEX fonts can be arbitrarily scaled and rotated.

16.3 Identify Inter-Object Relations

The second step is to "know your objects". Without this knowledge, you can not effectively determine where locking should be applied. The best method for understanding the extension object database is to create a diagram of the objects and their inter-relationships. In APPENDIX A, there is a description of how this was done for the core objects. Having a good Entity-Relationship diagram such as FIGURE 60 will help you in the subsequent steps. Your diagram may not be as complicated, but at least you'll be able to put an envelope around the object space.

16.3.1 Identify PEX Inter-Object Relations

The nine PEX specific resources are inter-related as described in the enity-relationship diagram of FIGURE 53. The two primary objects are the PHIGS Workstation and the Renderer. Most objects support the activities centered around these two resources.

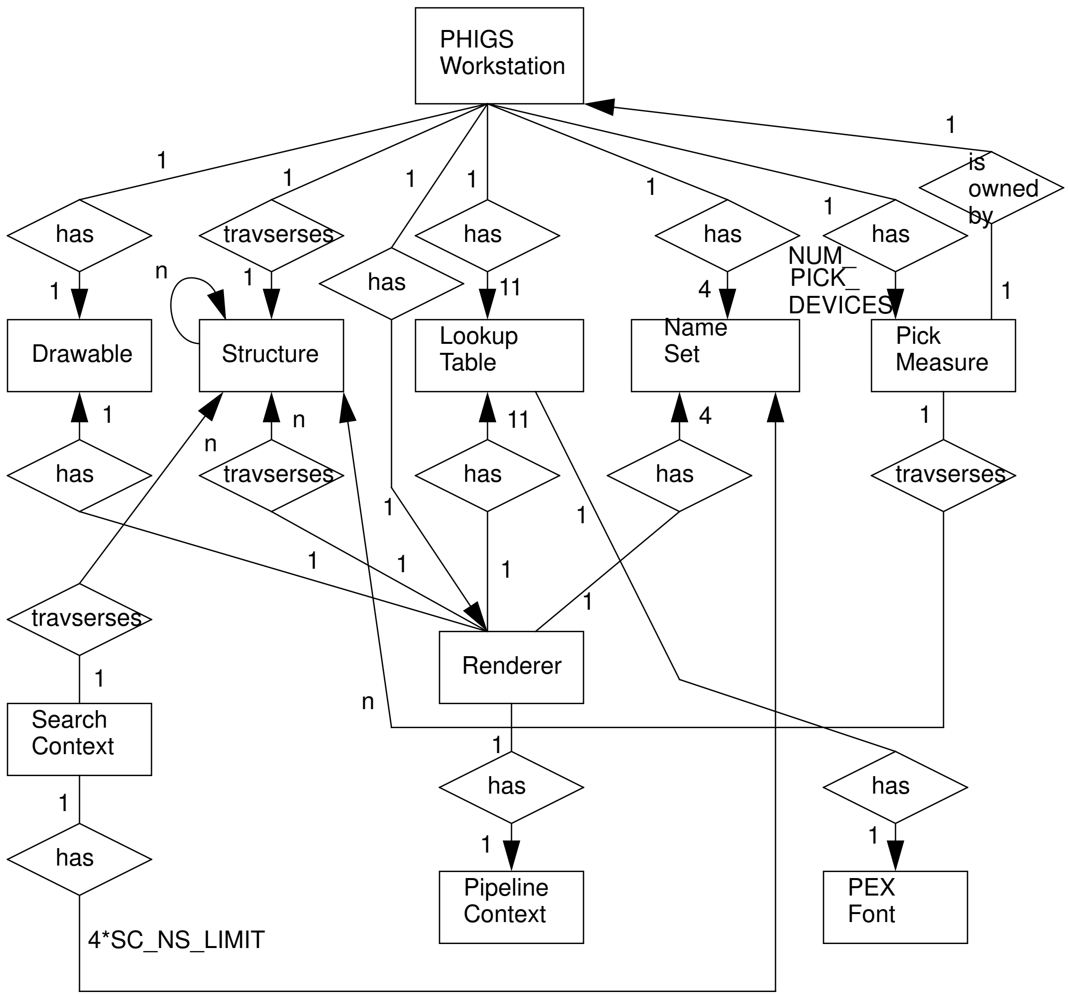


FIGURE 52 PEX Entity-Relationship Diagram

Although the entity-relationship diagram shows the implementation independent relationships of the PEX objects, we are constrained by the existing PEX-SI when making PEX MTX-safe. FIGURE 53 shows how the PEX object inter-relationships are currently implemented by the code modules.

There are three basic ways that the PEX-SI preserves the dependency of object A on object B:

- object A has a handle to object B (indicated by a pointed line)
- In addtion to the pointer, object B also maintains a count of the number of times it is referenced (indicated by a greyed, pointed line)
- In addition to the pointer, object B maintains a cross-reference list of handles to the objects that reference it (such as object A's handle)

For example, a reference to the PHIGS Workstation can generate a reference to a Lookup Table, a Name Set, a PEX Font, and/or a Structure. In fact, a Lookup Table object that is referenced by a PHIGS Workstation cannot be deleted until the workstation de-registers the lookup table. Internally, the lookup table keeps a list of renderer objects and a list of PHIGS Workstation objects that reflect these resource dependencies. In turn, a PHIGS workstation object keeps a reference count of the number of pick measures that are using the workstation.

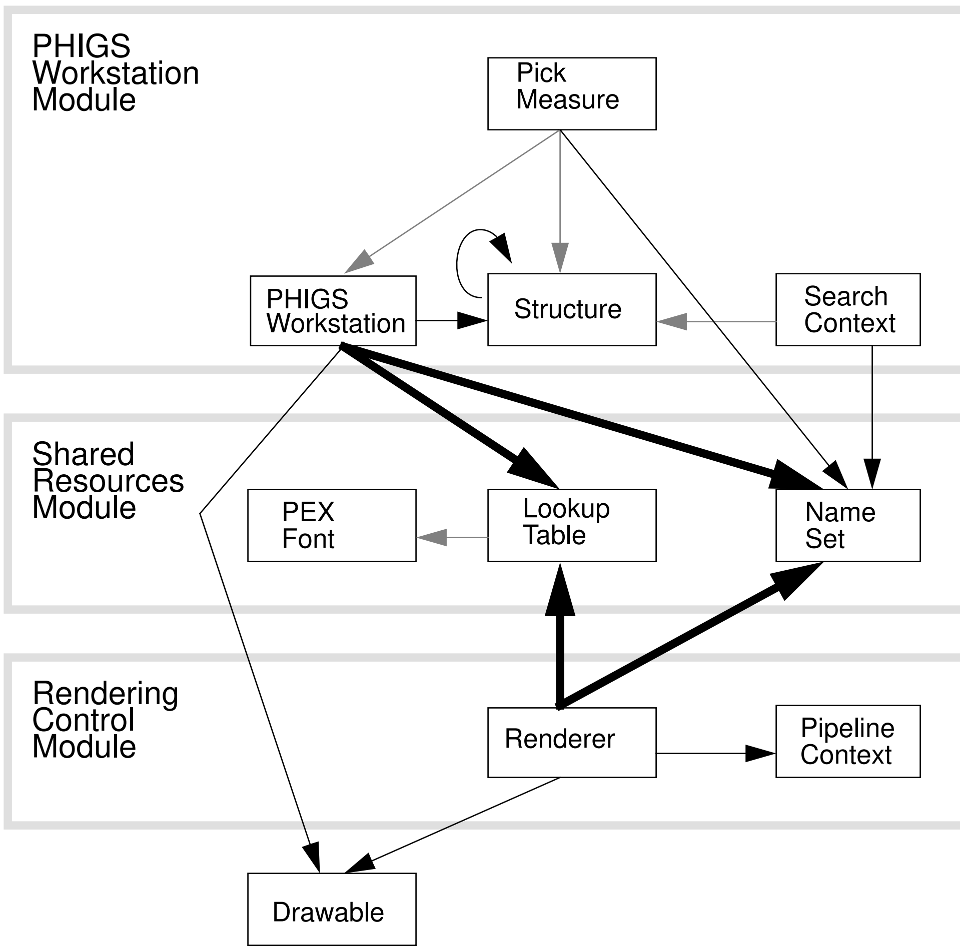


FIGURE 53 PEX Entity-Relationship Diagram

16.4 Identify Object Usage

In the core MTX design, the hardest task was deciding how the core protocol requests should be redesigned so that integrity of data access was insured. The core requests use the RDB Monitor and the POQ Monitor to control access to objects for which the server and X client have a common handle (resource id). Other server specific objects, such as devices, may be locked as needed and use the monitors described in the previous chapters. The usage of objects determines the locking controls needed to insure an adequate trade-off between performance and concurrency. The result of this analysis for the core requests is reflected in FIGURE 9 of CHAPTER 4. The locking requirements are described in earlier chapters of this document.

Extension writers must also decide on which objects to lock, the granularity of the locks, and the mechanisms to employ while locking. It must be decided what objects are to be locked and how the locking is to take place. Hence, this step requires the extension writer to create a matrix similar to FIGURE 9. The matrix should include all objects expected to be accessed by the extension requests. This includes core objects as well as extension objects.

16.4.1 Identify PEX Object Usage

In FIGURE 54, all PEX protocol requests have been categorized by the type of object they manipulate. It is apparent from the diagram that most requests manipulate their own objects, and only read other objects. In addition, the only core resource referenced by PEX is the *drawable*.

Extension Protocol Request Category	Data Object									
	Drawable	Lookup Table	Pipeline Context	Renderer	Structure	Name Set	Search Context	PHIGS Workstation	Pick	PEX Font
Extension Info	R									
Lookup Table	R	*								
Pipeline Context			*							
Renderer	R		R	*	R					
Structure					*		R	W	R	
Name Set						*				
Search Context					R	R	*			
PHIGS Workstation	W	W			R	R		*		
Picks						W			*	
PEX Font		R			R			R		*

FIGURE 54 PEX Object Usage

16.5 Identify Object Locking

The analysis reflected in the Object Usage Matrix of the previous step allows us to now determine how objects should be locked. Part of the process of building the list of objects is discovering the locking requirements on those objects. Objects may theoretically be accessed in any of the read and write mode combinations, but practically, we want to impose reasonable resource locks to insure mutual exclusion while ensuring maximum concurrency.

Synchronization is enforced by requiring each thread to lock the shared object it intends to access. The lock mode may be read only, read/write, or exclusive. Mutexes and condition variables adhering to the POSIX 1003.4a standard are used to implement this synchronization in the MTX server.

Once we have decided how to lock an object or block of objects, we should consider the granularity of the locks. Lock granularity can be defined in terms of the size of the resource to be locked and the length of time that a resource is protected from mutual access (see CHAPTER 8). Granularity can range from fine to coarse grained. An example of fine grained locking is that specified for the pixmap. An individual pixmap is locked for a short period of time. An example of coarse grained locking is the Device/ Event Object. The entire device database including grabs and events are locked for longer periods. Determining the lock granularity of each object will depend on the expected use of that object and the read/write access level required.

Choosing the correct level of lock granularity for each resource will be important in maximizing performance and interactivity. For instance, if multiple threads reading trackball input were to lock around each read while updating the device record, then more time would be spent in locking/unlocking than if the thread were to lock, execute multiple reads, and then unlock. Since there is overhead in locking and unlocking, fine grained locking will consume more system resources but provide quick access to objects. Generally, coarse grained locking will consume fewer system resources but result in a higher probability of contention by threads. Fine grained locking increases interactivity at the expense of performance and memory usage while coarse grained locking increases performance at the expense of interactivity and complexity.

Complementing the choice of lock granularity is the decision on how the locks will be organized. A lock hierarchy must be created that defines the order in which locks are acquired. Lock precedence allows us to avoid deadlock situations. FIGURE 20 in CHAPTER 8 shows the lock precedence for the core objects.

In order to increase the interactivity of the server but not degrade performance, the lock granularity for each object and the lock precedence must be carefully considered with the above trade-offs in mind.

The extension writer must also decide on the mechanism for implementing lock protection. In core MTX, there are a several monitors that enforce this protection. The extension writer should use the exeternal interface to these core monitors when accessing core objects. The extension writer should define which objects, whether core or extension, are protected by which monitor.

The extension writer must use the core external interface for the RDB and the POQ to access core objects, while access to the device database requires the DE Monitor. When sending messages, such as replies, events, and errors, the extension request must use the MO Monitor. For extension objects that are managed by the RDB, the extension writer may have to allocate new resource types and classes before resource lookup and locking of the extension object.

For locking of non-RDB objects, the extension writer should create standard interfaces so that other threads executing requests from the extension will not collide. This is best implemented using methods similar to the core monitors. The monitor can be designed to give read only, read/write, or exclusive access to the object - it depends largely on the results of the previous steps. See the discussion in CHAPTER 8 for further clarification of the trade-offs between performance and concurrency.

16.5.1 Identify PEX Object Locking

The PEX-SI stores the nine types of PEX objects in the RDB. Since the core server protects access to the RDB with the RDB Monitor, the PEX-SI must be modified to use the RDB Monitor interface (see CHAPTER 9) in order to make the extension MTX-safe. Although the RDB Monitor protects access to PEX objects, the PEX implementor must also decide on the granularity of the object locks.

The following subsections describe how PEX object locks can be implemented with various levels of granularity.

16.5.1.1 Coarse Grained PEX Locks

Monitor	RDB		POQ													MO			
Lock / Object/ Function	RDBMutex[i]	Lockbits in resource	POQMutex	* CM_R/W_Hierarchy	* CM_R/W_Geometry	CM_R/W_Colormap	CM_R/W_EventProp	CM_R/W_ScreenSaver	CM_X_GrabServer	CM_X_Cursor	* CM_X_Render	CM_X_ICCCM	CM_X_Server	CM_Extension_BIT	PEX _CM_X_Other	GlobalMessageBufferMutex	MessagePoolMutex	MessageDeliveryMutex[i]	Thread Specific (Implied)
OSCommOutput																			X
LocalMessageBuffer																			X
LocalPOQElement			X																X
ClientRDB	X																		X
window	X	X		X	X	X	X			X	X	X							
pixmap	X	X																	
Lookup Table	X														X				Coarse
Pipeline Context	X														X				
Renderer	X														X				
Structure	X														X				
Name Set	X														X				
Search Context	X														X				
PHIGS Workstation	X														X				
Pick Measure	X														X				
xferGlobalToSocket																X		X	
Grab/UngrabServer									X										
POQ			X																
GlobalMessageBuffer																X			
MessagePool																	X		

FIGURE 55 PEX Lock Summary with Coarse Grained Locks

We can decide to perform coarse grained locking in which all PEX objects are locked with a single lock. This would be implemented with the CM_EXTENSION_BIT (see CHAPTER 10). The PEX protocol request would set this bit to indicate that there is another set of PEX POQ bits to check. PEX could allocate one bit, PEX_CM_X_Other, in the PEX extension conflict mask to indicate that PEX requests lock all PEX objects together, regardless of type (see FIGURE 55).

The disadvantage is that only one PEX request can run at a time. However, non-colliding core requests could still continue to execute concurrently. The advatange of this lock strategy is that it is easy to implement.

As a second pass, we can create finer grained locks that partition the PEX objects into smaller groups. As an example, we could create a separate lock for each of the nine PEX resource types. After looking-up an object of type

16.5.1.2 Medium Grained PEX Locks

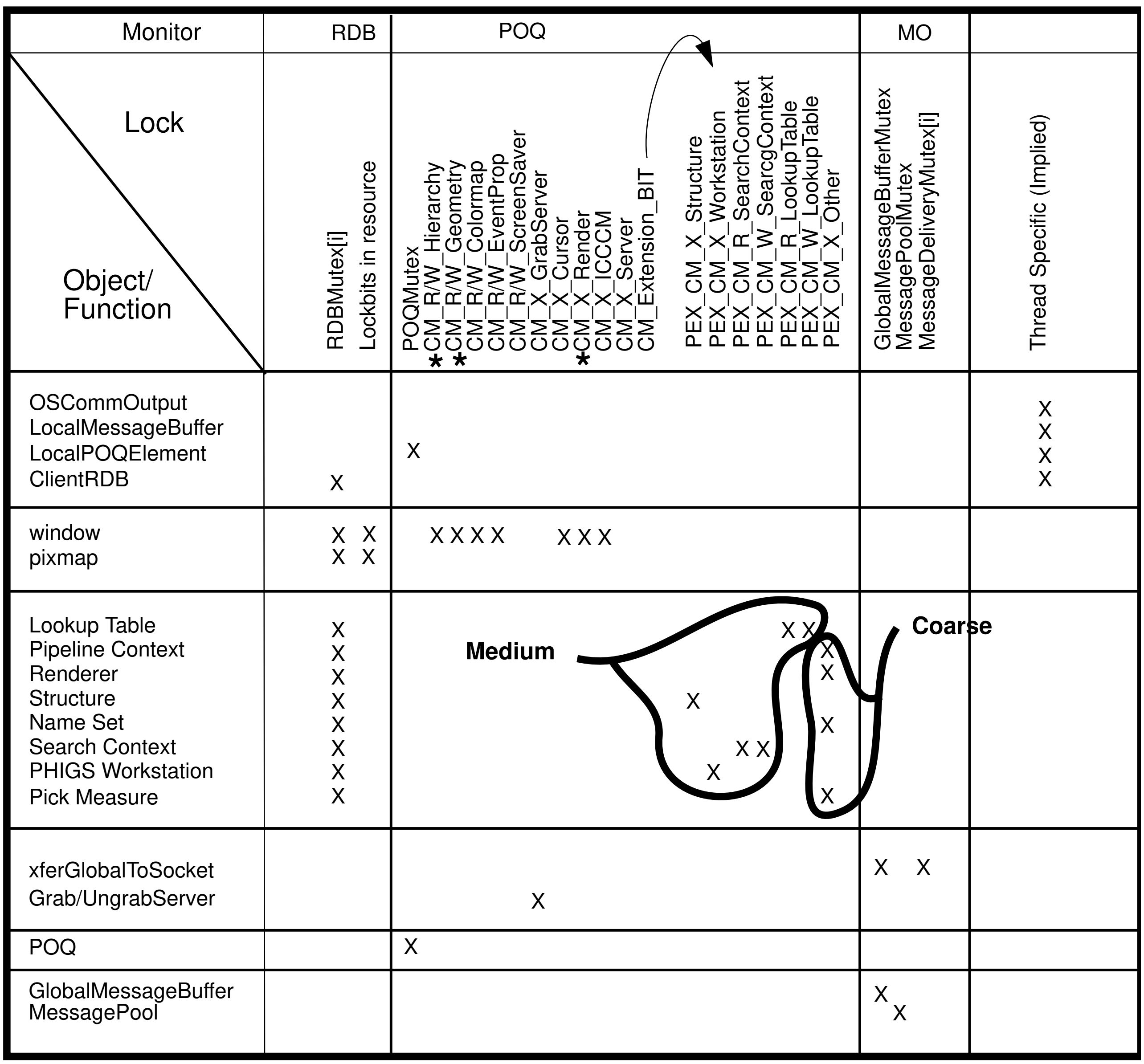


FIGURE 56 PEX Lock Summary with Medium Grained Locks

We can do a little more work in the extension to increase concurrency by implementing a medium grained locking strategy. The CM_EXTENSION_BIT is still used to indicate that there is another set of PEX POQ bits to check, but instead of locking all objects together, we can lock objects of the same type. For example, FIGURE 55 shows that any PEX protocol requests that use *structures* would set the PEX_CM_X_Structure bit. FIGURE 55 also shows that *workstations* are exclusively locked, while *seach contexts* and *lookup tables* are read/write accessable. The PEX_CM_X_Other will exclusively protect the remaining PEX resources of type *renderer*, *name set*, *pick measure*, and *PEX fonts*.

The disadvantage of this lock strategy is that we still have coarse grained locking for any object of type *renderer*, *name set*, *pick measure*, and *PEX fonts*. Also, we have more work to do when implementing the medium grained locking for *structures*, *workstations*, *seach contexts*, and *lookup tables*. The advatange of this mix of medium and coarse grained locking is that PEX requests can execute concurrently if they are operating on non-conflicting resources. For example, if a PEXGetTableInfo request sets CM_R_Window and PEX_CM_R_Structure, while a PEXCreateStructure sets PEX_CM_X_Structure, then the two PEX requests can execute without collision. However, a PEXCreateRenderer request and a PEXFreeNameSet request would not be able to execute concurrently because each must be able to set the PEX_CM_X_Other bit. One request would have to wait for the other request to finish.

16.5.1.3 Fine Grained PEX Locks

We can do still more work in the extension to increase concurrency by implementing a fine grained locking strategy. The CM_EXTENSION_BIT is still used to indicate that there is another set of PEX POQ bits to check, but instead of locking all objects together or locking objects by type, we can lock individual objects in the RDB. For example, FIGURE 55 shows that any PEX protocol requests that use *workstations* would be able to lock individual workstation resources. Note that no PEX_CM_X_Workstation extension bit is needed. If all other medium and coarse grained locking were to remain as in FIGURE 55, we would have a mix of locking granularity.

The disadvantage of this mix of lock strategies is that we still have coarse grained locking for any object of type *renderer*, *name set*, *pick measure*, and *PEX fonts*. Also, we have even more work to implement the fine grained locking for *workstations*. The advatange of this mix of fine, medium, and coarse grained locking is that PEX requests can still execute concurrently if they are operating on non-conflicting resources. In this case, if a PEXFreePhigsWKS and a PEXGetDynamics are operating on different workstation objects, they can execute concurrently. In the previous discussion of medium grained locking, these two PEX requests would have blocked each other.

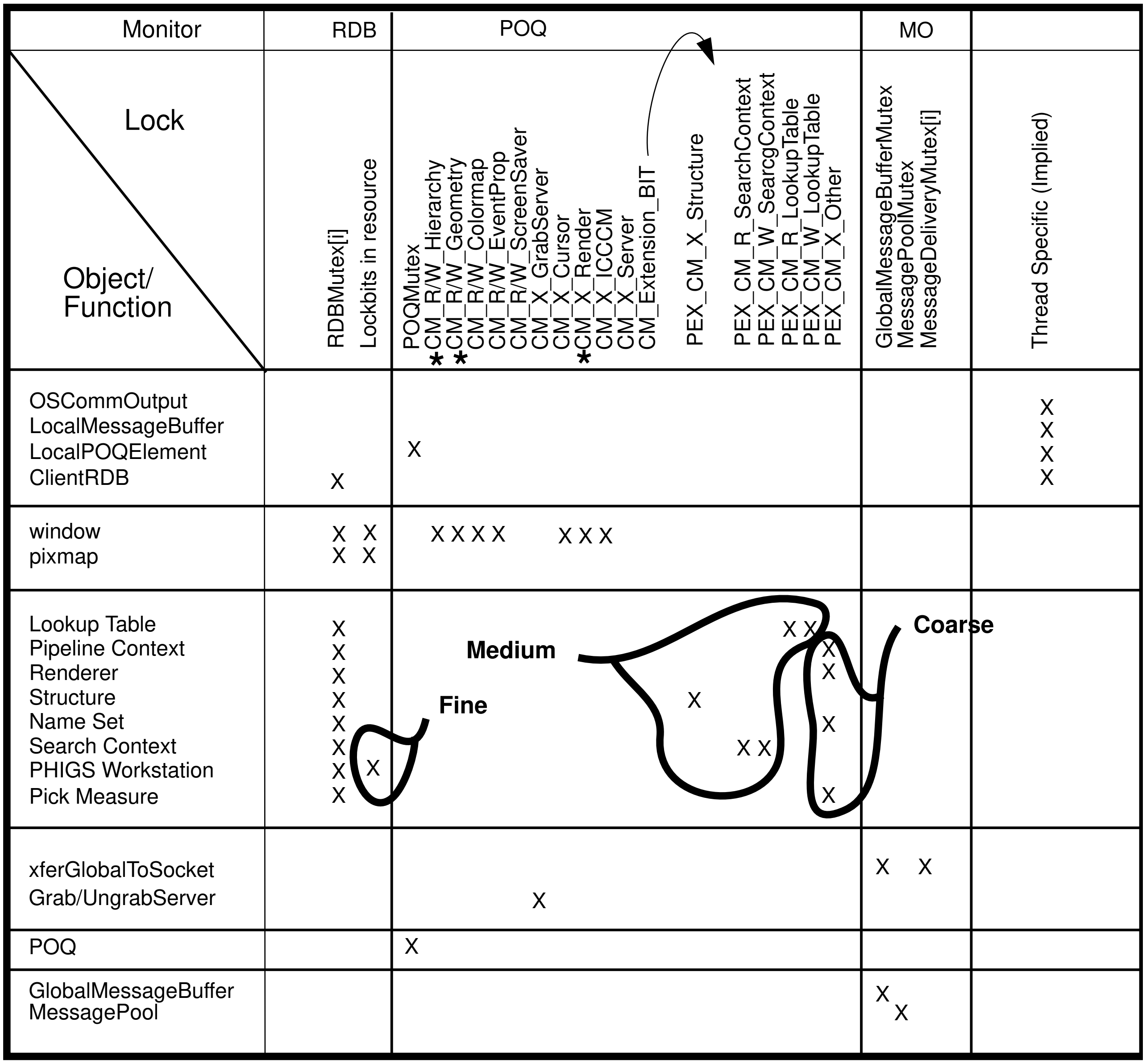


FIGURE 57 PEX Lock Summary with Fine Grained Locks

16.6 Analyse Extension for Reentrancy

In the previous sections, we have shown how locking strategies can be used to insure that the extension is thread-safe. The other facet of being thread-safe requires that the extension be made reentrant. This involves identifying and removing references to global and static variables where possible. If it is not practical to remove global and static variables, locks must be used to serialize access.

16.6.1 Analyse PEX for Reentrancy

The current implementation of PEX makes frequent use of global state and static variable declarations in both the dipex and ddpex layers. The following is a partial list of areas in PEX that have reentrancy problems:

1. ospex
   - PEX Font - defaultPEXFont, already_determined, font_dir_path
2. dipex/dispatch
   - Extension Init - PEXRequest array, set table array
   - Utils - obj_struct_sizes array, obj_array_sizes array
3. dipex/objects
   - PEX buffer management
   - Error recovery for pipeline context and search context
   - fakeRenderNetwork froc array
4. dipex/swap
   - check floating point - PEXOutputCmd arrays, PEXRequest arrays, PEXReply array, lastfp arrays, error recovery of check
   - convert request - PEXRequest array
5. ddpex/mi/include
   - Nurbs
   - render - identity transformation matrix
   - pex error base
   - Many LineDash and Marker static variables
6. ddpex/mi/level2
   - device dependent context - several global arrays, pipeline context flag and attributes
   - InitExecuteOCTable array
   - vector transform matrix for Output Commands
   - Pick and Text directions vectors
7. ddpex/mi/level3
   - renderer - OCTable array management, render and pick primitive table arrays
8. ddpex/mi/level4
   - Client Side Structure - OCTable array management
   - structure traversal - CSSelement array management, OCTable array management
   - Workstation - miHlhsrModeET array, miDisplayUpdateModeET array, NPC and viewport init, error recovery
9. ddpex/mi/shared

- Lookup Tables - various LUT related arrays

In most cases, PEX global arrays can be protected by global locks. For example, the LUT arrays could be protected by a LUTMutex.

16.7 Implement the MTX-safe design

Several examples of lock granularity were cited in the previous sections. The extension can implement coarse grained locking first just to get the extension integrated with MTX. Later, the extension can be modified to use medium or fine grained locks as the implementation matures. In this way, the extension can be tuned and optimized in stages as areas of lock granularity are tested.

16.7.1 Implement the PEX MTX-safe design

The nine PEX object types use the RDB Monitor for lookup of the objects. If access to any of these objects is to be coarse or medium grained, then the POQ Monitor can be used to protect them, If fine grained locking is needed, then the RDB Monitor can be used.

Each of the top level routines that implement the PEX protocol requests must be reworked to incorporate the external interface of RDB and POQ locking. If the device database is accessed by any PEX requests, the DE Monitor external interface must be used. All message delivery is managed through the MO Monitor interface.

We complete our PEX discussion by presenting some examples of how PEX routines can be made MTX-safe. The next two figures use the following notation:

~~Strikethrough text~~	Lines of text to be deleted.
Bold text	New code added to function.
Italic text	Comments explaining code changes.

FIGURE 58 an example of how the ***ProcPEXDispatch()*** routine can be made MTX-safe. It shows changes that can be made to the function **ProcPEXDispatch** in order to integrate it with MTX Server. First, the function call to **LookupIDByType** will be replaced with the macro **LOCK_AND_VERIFY_PEXCONTEXT**. The macro will lock the client's context structure in the RDB and update **cntxtPtr** to point to the context structure.

Before **ProcPEXDispatch** returns, the call to **UNLOCK_PEXCONTEXT** will be added. This macro will unlock the PEX context structure, thus allowing other threads to access it.

```
ProcPEXDispatch(client)
ClientPtr client;
{
  XID pexId;
  pexContext *cntxtPtr;
  CARD8 op;
  ErrorCode err = Success;

  REQUEST( xReq );
  pexId = PEXID( client, PEXCONTEXTTABLE );
```

The call to LookupIDByType() will be replaced with the macro LOCK_AND_VERIFY_PEX_CONTEXT(). LOCK_AND_VERIFY_PEX_CONTEXT() will properly lock and access the Resource Data Base. It will also update the cntxtPtr.

```
  cntxtPtr = (pexContext *)LookupIDByType(pexId, PEXContextType);
  LOCK_AND_VERIFY_PEXCONTEXT(pexId, &cntxtPtr, PEXContextType);

  if( !cntxtPtr ) {
      if (!(cntxtPtr = InitPexClient(client))) return (BadAlloc);
  }

  op = ((pexReq *)stuff)->opcode;

  if ((op >= PEX_GetExtensionInfo) && (op <= PEXMaxRequest)) {
      if (!(err = set_tables[op](cntxtPtr, stuff))) {
      cntxtPtr->current_req = (pexReq *)stuff;
      err = cntxtPtr->pexRequest[ op ]( cntxtPtr, stuff ); }
  } else {
      err = BadRequest;
  }
```

Since a resource was locked earlier, now it must be unlocked.

```
  UNLOCK_PEXCONTEXT(cntxtPtr);

  return( err );
}
```

FIGURE 58 MTX-safe *ProcPEXDispatch()*

Changes that should be made to PEX protocol requests in order to make them MTX-safe are in general typical of the waythat core requests were made MTX-safe. As an example, FIGURE 59 shows the changes that should be made to the routine ***PEXCreateLookupTable()***. First the call to the macro **INIT_POQ** has been added. This macros defines the POQ structure.

Next, the calls to **LookupIDByType** and **LU_DRAWABLE** have been replaced with the macro **LOCK_AND_VERIFY_LUT_AND_DRAWABLE**. **LOCK_AND_VERI-**

FY_LUT_AND_DRAWABLE performs three operations. First, if fine grain locking is being used, it will lock PEX resources, in this case a LUT. Second, it will lock the drawable, which is an X resource. Third, it will add an entry on the POQ for this protocol request.

Finally, the call to **UNLOCK_LUT_AND_DRAWABLE** was added before **PEXCreateLookupTable** returns. **UNLOCK_LUT_AND_DRAWABLE** also performs three functions. First, it removes the entry from the POQ for this protocol request. Second, it unlocks the drawable. Third, for fine grain locking, it unlocks the LUT.

```
ErrorCode PEXCreateLookupTable (cntxtPtr, strmPtr)
pexContext *cntxtPtr; /* context pointer */
pexCreateLookupTableReq *strmPtr; /* stream pointer */
{
 ErrorCode freeLUT ();
ErrorCode err = Success;
DrawablePtr pdraw = 0;
diLUTHandle lutptr = 0;
```

Define the Pending Operation Queue.

```
INIT_POQ_LOCK;
```

PEX_ERR_EXIT will be relinked with the new version of SendErrorToClient which operates in the MTX environment.

```
if (!VALID_TABLETYPE(strmPtr->tableType))
PEX_ERR_EXIT(BadValue,strmPtr->tableType,cntxtPtr);
```

The calls to LookupIDByType() and LU_DRAWABLE will be replaced with the resource locking macro LOCK_AND_VERIFY_LUT_AND_DRAWABLE. LOCK_AND_VERIFY_LUT_AND_DRAWABLE locks the drawable for read access. LOCK_AND_VERIFY_LUT_AND_DRAWABLE implicitly locks all PEX resources by placing an entry on the POQ.

~~lutptr = (diLUTHandle) LookupIDByType (strmPtr->lut, PEXLutType);~~

```
LOCK_AND_VERIFY_LUT_AND_DRAWABLE(strmPtr->lut, PEXLutType,
&lutptr, strmPtr->drawableExample, pdraw);

if (lutptr) PEX_ERR_EXIT(BadIDChoice,strmPtr->lut, cntxtPtr);
```

Delete call to LU_DRAWABLE, the drawable will be locked in. LOCK_AND_VERIFY_LUT_AND_DRAWABLE.

~~LU_DRAWABLE(strmPtr->drawableExample, pdraw);~~

```
lutptr = (diLUTHandle) Xalloc ((unsigned long)sizeof(ddLUTResource));
if (!lutptr) PEX_ERR_EXIT (BadAlloc,0,cntxtPtr);
lutptr->id = strmPtr->lut;
lutptr->lutType = strmPtr->tableType;

err = CreateLUT( pdraw, lutptr);
if (err) {
Xfree((pointer)lutptr);
PEX_ERR_EXIT(err,0,cntxtPtr);
}
```

ADDRESOURCE will be relinked with the MTX version of AddResource.

```
ADDRESOURCE (strmPtr->lut, PEXLutType, lutptr);
```

Now unlock the resources.

```
UNLOCK_DRAWABLE_AND_LUT(lutptr, pDrawable);
return( err );
} /* end-PEXCreateLookupTable() */
```

FIGURE 59 MTX-safe ***PEXCreateLookupTable()***

One last note - if PEX uses the MBX multibuffering extension to handle its double-buffering requirements, then MBX must also be made MTX-safe.

16.8 Extension Threads

In the current design, the extension request executes within the context of a Client Input Thread. Since there is one CIT per client connection, this design insures that a client will see atomic and serial execution of its requests. If the CIT executing the extension request were to create another thread, it is possible that seriality could break if thread synchronization is not handled correctly.

We can envision that there are extensions which may want to increase interactivity and concurrency within the extension itself. For example, multi-threaded PEX may create a separate thread for each stage of the transformation pipeline. This would allow greater rendering throughput. The advantage gained by increasing PEX performance, however, should be balanced against the increased complexity required to synchronize the PEX threads and other core threads executing core server operations.

In general, because of the danger of introducing hard-to-find thread synchronization problems, we do not recommend that you create any threads. If you absolutely need to create more threads, however, make sure you know what you are doing. Otherwise, subtle errors may be introduced that manifest in unexpected parts of the core server.

16.9 Extension Initialization

As in the R5 server, *extensionInitProc* is added to *InitExtensions*. The initialization procedure for the extension will add the extension to the list of existing extensions. It may also need to call *CreateNewResourceType* and/or *CreateNewResourceClass* to register resources in the RDB in addition to the core resources. In addition, the initialization process will also determine the range of event and error codes used by the extension.

When the MTX server is initialized by the MST, all extensions are initialized and registered for use.

16.10 Final Thoughts on Extension Design

The previous sections have talked about how knowledge of the core and extension objects determines locking strategies. These locking strategies are then enforced by the monitor interface.

In general, the process of designing extensions for the MTX environment parallels that taken with the R5 server. The extension has a specific set of functions to perform. These functions are implemented via the extension requests. To insure that requests execute without colliding with other MTX threads, one must make sure that the request locks the shared objects it needs without deadlock or significant loss of performance.

The extension writer should look at a couple of core requests as examples to emulate. In general, the extension request and associated monitors should follow the object access algorithm described in CHAPTER 8.

Note that existing X extensions will have to be "fixed" to at least be made re-entrant, and more probably made compatible with the new message delivery system. On top of these changes, there is a good chance that extension specific monitors will be designed. MTX changes should be made to the base extensions (such as MISC and SHAPE) as well as XIE, the Input Extension, PEX, DPS, MBX, and others.

APPENDIX A Data Objects

This appendix shows the functional relationships of the key data objects in the MTX.

1.1 Entity-Relationship Diagram

The Entity-Relationship (ER) diagram summarizes the information in a data object design. This information may include both data objects and real objects. These objects have associated attributes that distinguish the object from all others.

FIGURE 60 describes the MTX ER diagram. Read the ER diagram from top to bottom. The **1**,**n**, or **m** on the linked edges indicate the degree of object mapping. For example, one window grabs many devices, and one window is a child of only one parent window. Rectangles indicate objects, diamonds indicate interobject relationships, and circles indicate major attributes of objects or relationships.

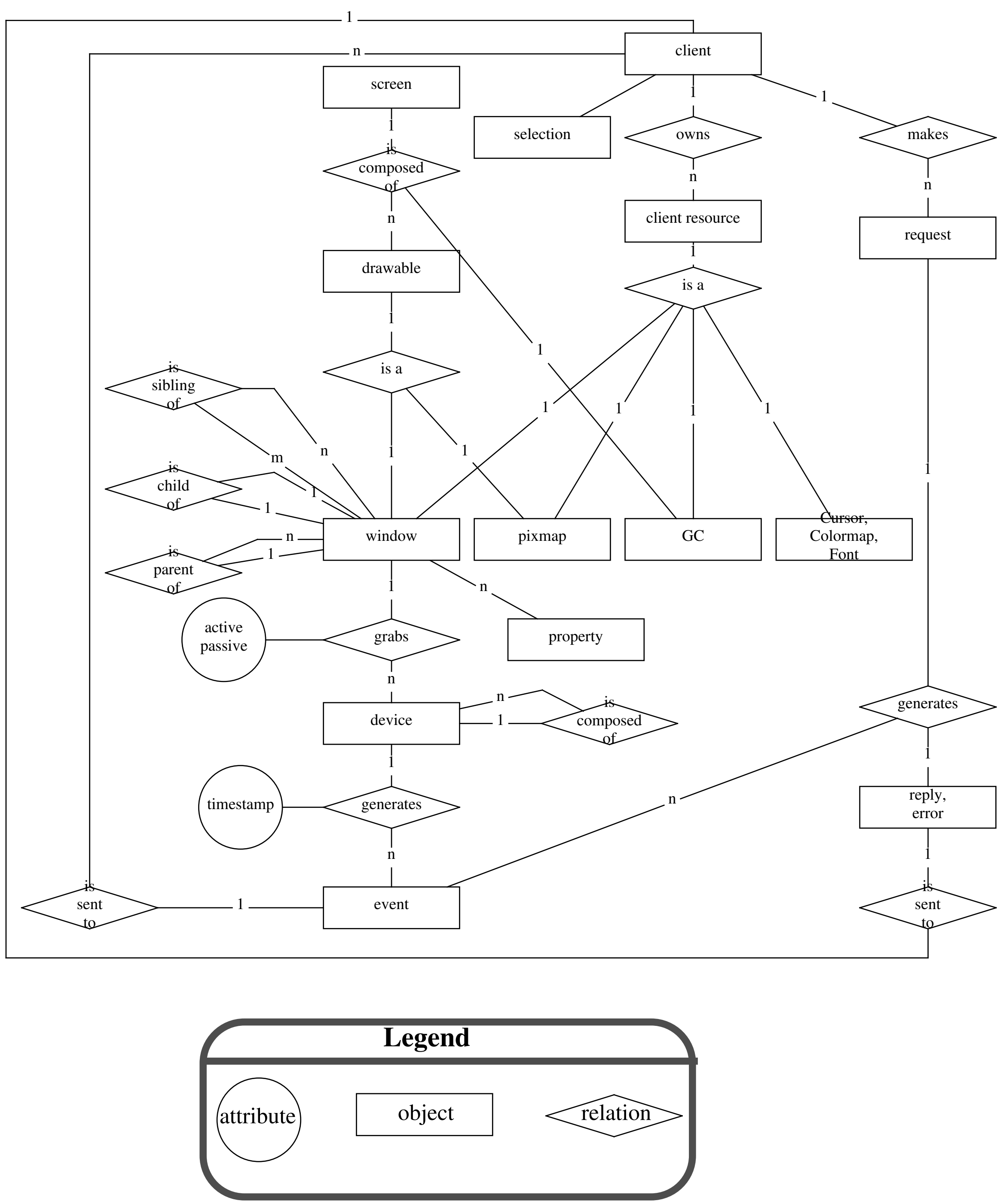


FIGURE 60 Entity-Relationship Diagram

1.2 Object Categories

The key objects defined in the entity-relationship diagram can be grouped into functional categories. Each category is an encapsulation of those objects which have similar capabilities and functions. By defining categories of objects, we can determine what functions in a thread are most likely to access those objects. The following sections describe those categories.

1.2.1 Client

The Client category includes all data objects that allow the MTX to understand the state of the X client.

- ClientResourceRec
- ResourceRec
- ClientRec

1.2.2 Color

The Color category includes the color tables and colormaps.

- ColorMapRec
- CMEntry
- RGB

1.2.3 Cursor

The Cursor category includes the cursor, glyph, and sprite objects.

- CursorRec
- CursorBits
- DevUnion
- CursorMetricRec
- GlyphShareRec
- Sprite
- Hotspot

1.2.4 Device

The Device category includes all information about input and output devices. These objects describe device state, device setup, and device grabs.

- SyncEvents
- QdEventRec
- InputInfo
- DeviceIntRec
- DeviceRec
- KbdFeedbackRec - KeybdCtrl

- PtrFeedbackRec - PtrCtrl
- IntegerFeedbackRec - IntegerCtrl
- StringFeedbackRec - StringCtrl
- BellFeedbackRec - BellCtrl
- LedFeedbackRec - LedCtrl
- KeyClassRec - KeySymsRec
- ValuatorClassRec
- XAxisInfo
- ButtonClassRec
- FocusClassRec
- ProximityClassRec
- GrabRec
- DetailRec

1.2.5 DIXFontInfo

The DIXFontInfo category describes those font objects that are device independent.

- FontRec
- FontIntRec
- DixFontProp
- CharInfoRec
- XCharInfo
- ExtentInfoRec

1.2.6 Extension

Extensions interface to MTX by the setup of these objects.

- ExtensionEntry
- ScreenProcEntry
- ProcEntryProc

1.2.7 Event Masks

Window objects may mask events, may specify which ancestor from which masking will be inherited, and which other clients are interested in masking. These functions are delivered via the Event Mask category objects.

- OtherClients
- OtherInputMasks
- InputClients

1.2.8 GC

The Graphics Context objects describe how output to the graphics device or frame buffer will occur. It also gives the device-dependent functions that govern how the GC is handled by the machine independent (mi), mono frame buffer (mfb), and color frame buffer (cfb) layers.

- GC
- GCFuncs
- GCOps

1.2.9 OSComm

The OSComm category contains objects that interface the server to the client via the network.

- OSCommRec
- ConnectionInput
- ConnectionOutput
- FamilyMap
- HOST

1.2.10 OSFontInfo

This category of objects provides lookup of fonts and font path names. It is superceded by the font server.

- FontTableRec
- NameEntry
- FontNameRec
- FileEntry
- FontFileRec
- FontPathRec
- FontPropRec
- FontFileReaderRec

1.2.11 Pixmap

Pixmap objects describe their namesake.

- PixUnion
- PixmapRec
- DrawableRec

1.2.12 Property

The Property category contains property objects.

- PropertyRec

1.2.13 Screen

Screen related objects describe the environment in which objects, such as windows, exist.

- ScreenInfo
- PixmapFormatRec
- ScreenIntRec
- ScreenRec
- DepthRec
- VisualRec

1.2.14 Selection

Selections are described in this category.

- Selection

1.2.15 Window

The Window category describes how windows are connected and function.

- WindowRec
- WindowOptRec
- ValidateRec
- RegionRec
- RegionDataRec
- BoxRec

1.3 Interrelationships of Object Categories

Objects are grouped into functional categories. Each of the objects in one category can have multiple dependencies on objects in other categories. These interrelationships define a network in which data may flow from one category to another. This data flow can be useful when locking strategies are later defined during the detailed design phase of the project.

FIGURE 61 describes this interrelationship of object categories. The reference arrows indicate the direction of dependency. For example, there are objects in the window category that refer to objects in the device category and vice versa. Therefore, the window category is dependent on objects in the device category. We need to consider these dependencies when the granularity of locking is determined.

Note that although there is a network of interconnections, there are hierarchies that are localized to just a few categories. This localization also defines functionality just as the interrelationship of some objects led to the definition of the object category. This concept will be useful in determining the usage of objects by functions and threads. In turn, the usage information will be a tool in refining the granularity of the object and category locks as defined later during detailed design.

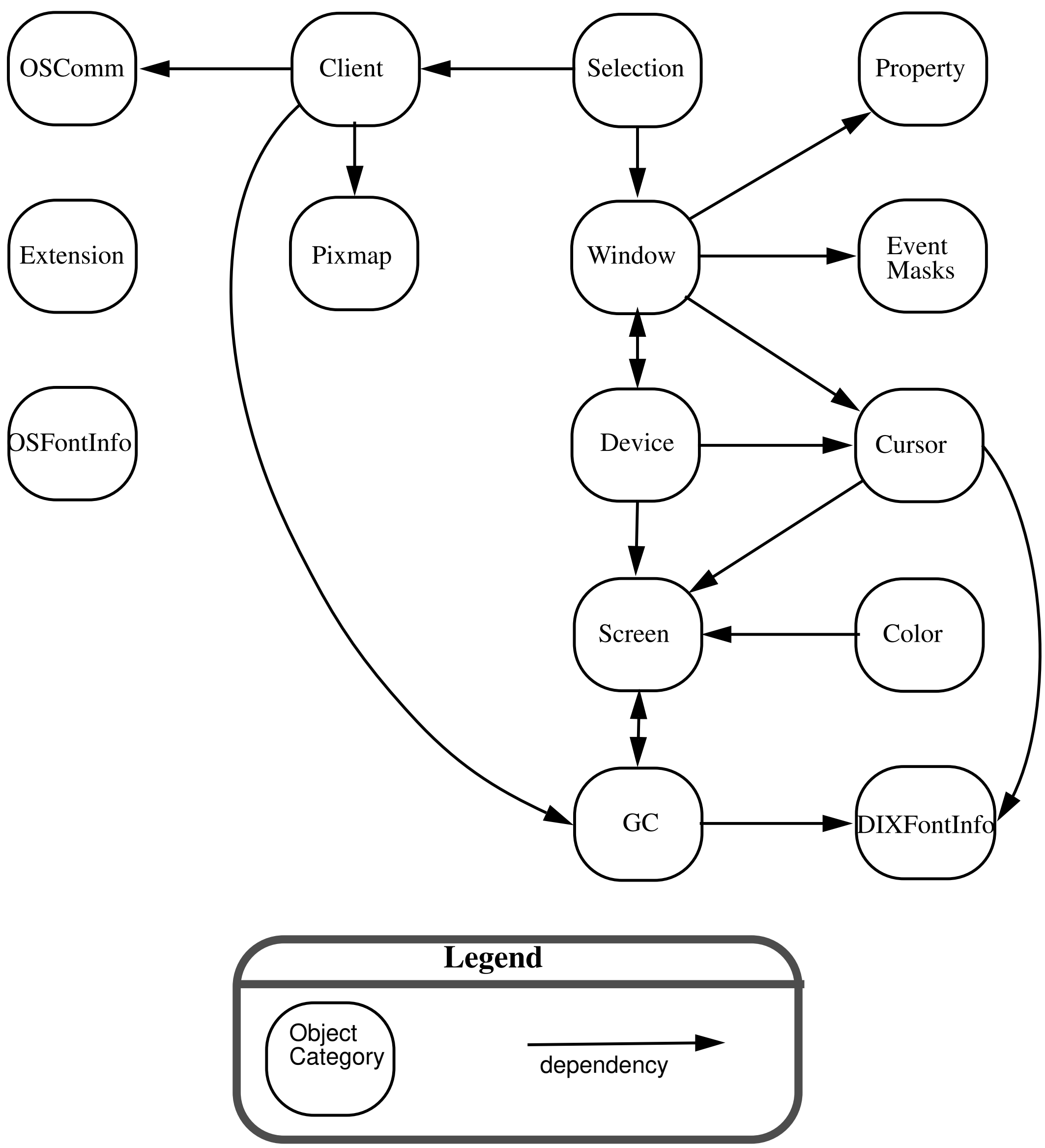


FIGURE 61 Object Category Interrelationships

APPENDIX B

Functionality Mapped to Design

This appendix contains a cross reference map of MTX server functionality versus server component design. For each major piece of MTX functionality described in the Functional Design Specification, the following table lists which MTX threads exhibit aspects of that functionality.

Also, the table shows which object groups (see APPENDIX A) implement the stated functionality in the design of the group's objects.

For example, the function of Event Delivery is exhibited in three types of threads: CITs, COTs, and the DIT. In addition, objects in the Connection and OutCommunication object groups are used to when mapping the functionality to the design.

Functionality	Threads				
	CIT	COT	CCT	DIT	MST
Request Processing	X				
Client Output Delivery	X	X		X	
Event Processing	X			X	
Client Connections			X		
Server Init & Control	X		X		X
DDX Software Cursor	X			X	
ScreenSaver	X			X	
BackingStore	X				
XDM Interface	X		X		X
FontServer Interface	X				
Extensions	X				X

FIGURE 62 Functionality Mapped to Threads

	Object Group							
Functionality	Render	Connection	Device	OutComm	InComm	OSFont	ICCCM	Extension
Request Processing	X	X	X		X	X	X	X
Client Output Delivery		X		X				
Event Processing	X		X					
Client Connections		X		X	X			
Server Init & Control	X	X	X					X
DDX Software Cursor	X		X					
ScreenSaver	X							
BackingStore	X							
XDM Interface		X						
FontServer Interface	X	X		X	X	X		
Extensions	X	X	X		X	X	X	X

FIGURE 63 Functionality Mapped to Object Groups

APPENDIX C

References

[Int90] Interactive Development Environments, San Francisco, CA 94105. *Software through Pictures User Manual*, March 1990

[MPJ88] Meilir Page-Jones, *The Practical Guide to Structured Systems Design*, Yourdon Press, 1988

[P1003.4a] Technical Committee on Operating Systems of the IEEE Computer Society, *Threads Extension for Portable Operating Systems*, P1003.4a/D6, February 26, 1992

[SG90] R. Scheifler and J.Gettys. *X Winodw System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFD*. Digital Press, 1990

[Smi91] John A. Smith. *Engineering a Multi-Threaded X Server*. In Xhibition Technical Conference, pages 17-27, 1991

[Y84] *Structured Design for Real-Time Systems*, Yourdon, Inc. 1984

[Y85] Structured Analysis Workshop - *Defining Requirements for Complex Systems*, Yourdon, Inc. 1985

END OF DOCUMENT