DATE:

May 1, 1989

Design Specification

for

Design of a Machine Independent X Image Extension Server System

Written by: Richard Piccolo

Issued by: IPG Engineering

Approved by:

Joe Mauro, IPG Architect

John Weber, Project Leader

Digital Equipment Corporation

Contents

PREFACE vi

CHAPTER 1 OVERVIEW 1–1

1.1 INTRODUCTION 1–1

1.2 XIE ORGANIZATION 1–2

1.2.1 XIELIB 1–3

1.2.2 DIX/XIE Component 1–4

1.2.3 DDX/XIE Component 1–4

1.2.3.1 DDX/Machine Independent Component • 1–4

1.2.3.2 DDX/Machine Dependent Component • 1–4

1.2.4 XIE Services 1–4

1.2.4.1 Session Services • 1–5

1.2.4.2 Resource Services • 1–5

1.2.4.2.1 IOB • 1–5

1.2.4.2.2 IDC • 1–5

1.2.4.2.3 IPC • 1–5

1.2.4.3 Server/Client Image/Export Services • 1–5

1.2.4.4 Transport Services • 1–6

1.2.4.5 Process Services • 1–6

1.2.4.6 Statistic Services • 1–6

1.2.4.7 Process Control Services • 1–6

1.2.4.8 Display Service • 1–6

1.2.4.9 Event Services • 1–6

1.2.5 Relationship to CORE X 1–6

1.2.5.1 CORE X Resources • 1–7

1.2.5.1.1 Graphics Context • 1–7

1.2.5.1.2 Drawables • 1–7

1.2.5.2 Processing Modes • 1–7

1.2.5.2.1 DIX Processing • 1–7

1.2.5.2.2 DIX/DDX Processing • 1–7

CHAPTER 2 DIX/XIE COMPONENT 2–1

2.1 INITIALIZATION PROCEDURES 2–3

2.1.1 XieInit 2–3

2.1.1.1 Making XIE Known to the CORE X • 2–3

2.1.1.2 Defining Errors and Events • 2–4

2.1.1.3 Defining XIE Resource Class and Types • 2–4

2.1.2 XieDeviceInit 2–5

2.2 DIX/XIE REQUEST PROCESSING 2–6

2.2.1 XieDispatcher 2–6

2.2.1.1 Error Notification to Client • 2–7

2.2.1.2 Swapping Requests • 2–7

Contents

2.2.2 DIX Request Procedures 2–8
2.2.2.1 Processing of Requests - Phase I • 2–8
2.2.2.1.1 Variable Length Requests • 2–8
2.2.2.2 Processing of Requests - Phase II • 2–9
2.2.2.2.1 XIE Replies • 2–9
2.2.2.2.2 Session Services • 2–10
2.2.2.2.2.1 XIEP_INIT_SESSION • 2–10
2.2.2.2.2.2 XIEP_TERM_SESSION • 2–10
2.2.2.2.3 Resource Services • 2–11
2.2.2.2.3.1 Creating XIE Resources • 2–11
2.2.2.2.3.2 Validating XIE Resources • 2–12
2.2.2.2.3.3 Freeing XIE Resources • 2–12
2.2.2.2.3.4 Querying XIE Resources • 2–12
2.2.2.2.4 Import/Export Services • 2–12
2.2.2.2.4.1 XIEP_IMPORT_IMAGE • 2–12
2.2.2.2.4.2 XIEP_EXPORT_IMAGE • 2–13
2.2.2.2.5 Transport Services • 2–14
2.2.2.2.5.1 IOB Data Formats and Organization • 2–14
2.2.2.2.5.2 XIEP_SET_TRANSPORT • 2–15
2.2.2.2.5.3 XIEP_PUT_TILE • 2–15
2.2.2.2.5.4 XIEP_GET_TILE • 2–16
2.2.2.2.5.5 XIEP_PUT_STREAM • 2–16
2.2.2.2.5.6 XIEP_GET_STREAM • 2–17
2.2.2.2.6 Process Services • 2–18
2.2.2.2.6.1 XIE's Handling of IOB Data Formats • 2–18
2.2.2.2.7 Statistics Services • 2–19
2.2.2.2.8 Process Control Services - TBS • 2–19
2.2.2.2.9 Display Service • 2–19
2.2.2.2.9.1 XIEP_DISPLAY_SEQUENCE • 2–19
2.2.2.2.10 Event Services • 2–20
2.2.2.2.10.1 Event Notification to Client • 2–20
2.2.2.2.10.2 XIEP_SELECT_EVENTS • 2–21
2.2.2.2.10.3 XIEP_QUERY_EVENTS • 2–21

CHAPTER 3 DDX COMPONENT 3–1

3.1 PROCEDURE VECTOR TABLE 3–1
3.1.1 Lower Level Image Processing Routines 3–4

3.2 DIX/DDX INTERFACE 3–4

CHAPTER 4 OPERATING SYSTEM CONSIDERATIONS 4–1

4.1 MEMORY MANAGEMENT 4–1

4.2 LINKING AND LOADING XIE 4–1

4.3 HARDWARE CONSIDERATIONS 4–1
4.3.1 Hardware Recognition 4–2
4.3.2 Multiple Function Request 4–2

Contents

APPENDIX A XIE PROTOCOL REQUESTS A–1

APPENDIX B MODULES AND PROCEDURES B–1

APPENDIX C XIE APPLICATION AND WIRE DEFINITIONS AND STRUCTURES C–1

APPENDIX D SERVER DEFINITIONS AND STRUCTURES D–1

APPENDIX E CORE X STRUCTURES E–1

APPENDIX F MACROS F–1

F.1 CORE X MACROS F–1

F.2 XIE MACROS F–2

APPENDIX G DIX/DDX INTERFACE G–1

FIGURES

1–1 Client/Server Relationship 1–2

1–2 Client/CORE X/Extension Relationship 1–3

2–1 DIX Procedures 2–2

3–1 DDX Subcomponents 3–2

TABLES

3–1 DDX Procedure Vector Table 3–2

A–1 Image Extension Functions A–1

A–2 Service Classes A–3

Preface

Associated Documents

- *Image Extension Concepts*
- *XIE_Protocol*

Chapter 1

Overview

This document defines the concepts, components, and relationships needed to extend the CORE X server to accommodate the proposed X Image Extension.

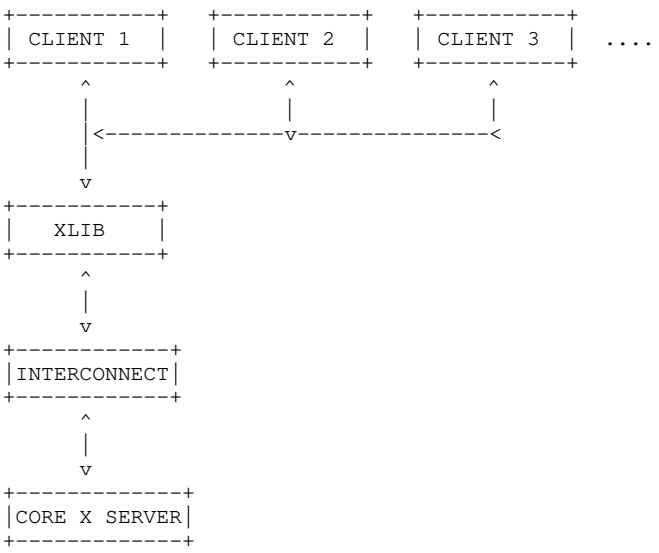
This design specification, which is based on the third release of X, Version 11 from MIT, addresses the extension's relationship to the CORE X, its components, its functionality and its operation.

1.1 Introduction

The CORE X window system or X is a network-transparent window system which allows users to run multiple applications simultaneously in windows, generating text and graphics in monochrome or color on a bitmap display. CORE X provides facilities for generating multifont text and two-dimensional graphics (such as points, lines, arcs, and polygons) in a hierarchy of rectangular windows.

An application program (client) connects to the window system (server) by some interprocess communication path. A client(s) communicates via calls to the CORE X library called XLIB. See Figure 1–1.

Figure 1–1: Client/Server Relationship

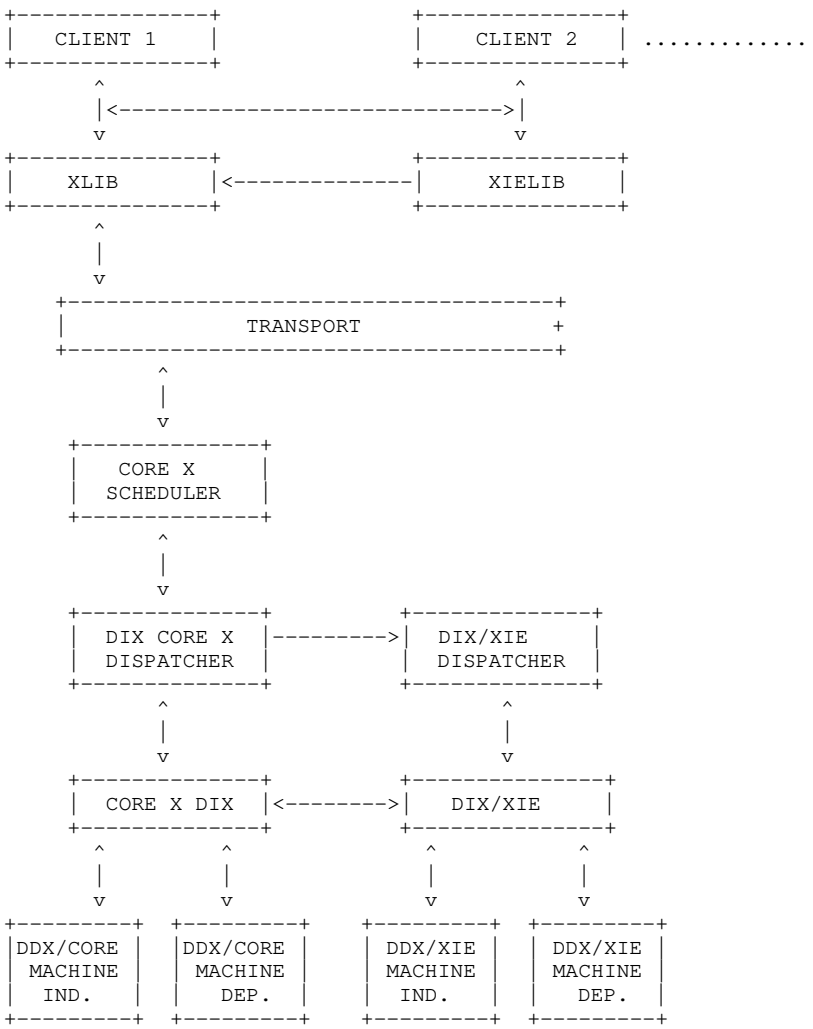


The CORE X protocol has minimally sufficient capabilities. There exists a mechanism within the protocol to allow "extending" the protocol to embrace new functionality. In order for the CORE X to accommodate image enhancement and rendition, a CORE X image extension (XIE) was conceived.

1.2 XIE Organization

The extension has three major components: an extended protocol interface library (XIE LIB), a device independent server component (DIX), and a device dependent server component (DDX). This structure is similar in concept to that of the CORE X. Figure 1–2 represents a block diagram of XIE' s components and their relationship to each other and to the CORE X.

Figure 1–2: Client/CORE X/Extension Relationship



1.2.1 XIELIB

This component resides on the client side of the wire and represents the extended protocol interface library (analogous to XLIB) which is used to pass requests to the server extension. Since this document represents the server side of the extension, it is stated here for completeness and because of its relevance to the CORE X protocol.

1.2.2 DIX/XIE Component

The functional responsibility of the DIX/XIE component is to:

- o Initialize the extension server.
- o Dispatch execution of extension server requests.
- o Return extension server events, errors and replies.
- o Manage extension server resources.
- o Interface with the CORE X services.
- o Interface with the server DDX component.

1.2.3 DDX/XIE Component

The DDX/XIE component consists of two subcomponents: machine independent DDX and machine dependent DDX. The former represents a software implementation of a DDX function, independent of hardware platform, while the latter represents a combined software/hardware implementation specific to a specific system platform or class of system platforms.

1.2.3.1 DDX/Machine Independent Component

This component includes the software algorithms to perform the various image processing tasks called upon by the client. This document will address this implementation.

1.2.3.2 DDX/Machine Dependent Component

This component performs all hardware related tasks in performing client requests. Hardware accelerators such as array processors and other specialized display components can achieve high efficiency in carrying out the various image rendition algorithms. To that end the DDX component will be called upon to make calls to the drivers of these devices when they exist.

1.2.4 XIE Services

A client XIE request can be classified by its type of service. The following outlines the various classes. For further details regarding the XIE services and resources, readers should read the documents referenced in the preface.

1.2.4.1 Session Services

These services allow clients to initialize and terminate a session with the extension.

1.2.4.2 Resource Services

These services allow clients to create, query, copy, and delete XIE resources. Resources include the following:

- IOB - Image Objects
- IDC - Image Domain Context
- IPC - Image Process Context

1.2.4.2.1 IOB

An Image Object (IOB) is a structure which contains the image data and closely associated attributes within a server.

1.2.4.2.2 IDC

An Image Domain Context (IDC) is a structure which is used for one of three purposes:

- define the subset or region of an image to process
- act as an area operand in area image operations
- both of the above (concurrently)

1.2.4.2.3 IPC

An Image Process Context (IPC) is a structure which describes the relationship between the various transfer, enhancement, and rendition processes that make up an image processing pipeline.

1.2.4.3 Server/Client Image/Export Services

These services allow clients to read/write image data within an XIE server resource from/to CORE X server resources such as pixmaps. The XIE resource is called an image object (IOB).

1.2.4.4 Transport Services

These services allow the transfer of image data to/from client/server. The server keeps images in server resources called image objects (IOB).

1.2.4.5 Process Services

These services provide for the application of low level image operators to image data primarily in support of preparation for or improvement of its display.

1.2.4.6 Statistic Services

These services provide clients procedures to generate statistical information about an image. Such statistics include the mean, variance, standard deviation, maximum and minimum of data within an area of an image.

1.2.4.7 Process Control Services

These services provide clients the ability to define and apply pipelines containing transport, process, and display operations.

1.2.4.8 Display Service

This service allows the client to display a sequence of prerendered image frames which are stored as pixmaps.

1.2.4.9 Event Services

These services allows the client to select or query XIE specific events.

1.2.5 Relationship to CORE X

There exist a resource and/or process relationship between XIE and CORE X which depends on the function the extension is carrying out. CORE X will provide the resource management of XIE resources as well as access to CORE X resources that XIE might need (such as the Graphics Context (GC) or drawables). In addition the extension may want to use a CORE X process rather than duplicating the process within the extension.

1.2.5.1 CORE X Resources

CORE X provides the extension with resource management for extension and CORE resources. Key CORE X resources that the extension will have interest in are the Graphics Context and drawables.

1.2.5.1.1 Graphics Context

The Graphics Context is used when the CORE X performs graphic operations. This resource consists of information relating to performing such operations: putImage (from a server data buffer to a CORE X pixmap), line style and so on.

1.2.5.1.2 Drawables

Graphics operations can be performed to either windows or pixmaps which collectively are called drawables. XIE will use these resources for services such as import/export.

1.2.5.2 Processing Modes

To better delineate which of the aforementioned components are used for a particular client XIE request, we define several processing modes. Appendix A maps each XIE protocol request to a service class and processing mode.

1.2.5.2.1 DIX Processing

This mode refers to those requests which do not need processes from the DDX component of XIE.

1.2.5.2.2 DIX/DDX Processing

This processing mode refers to those requests which rely on the DDX/XIE component to perform its task.

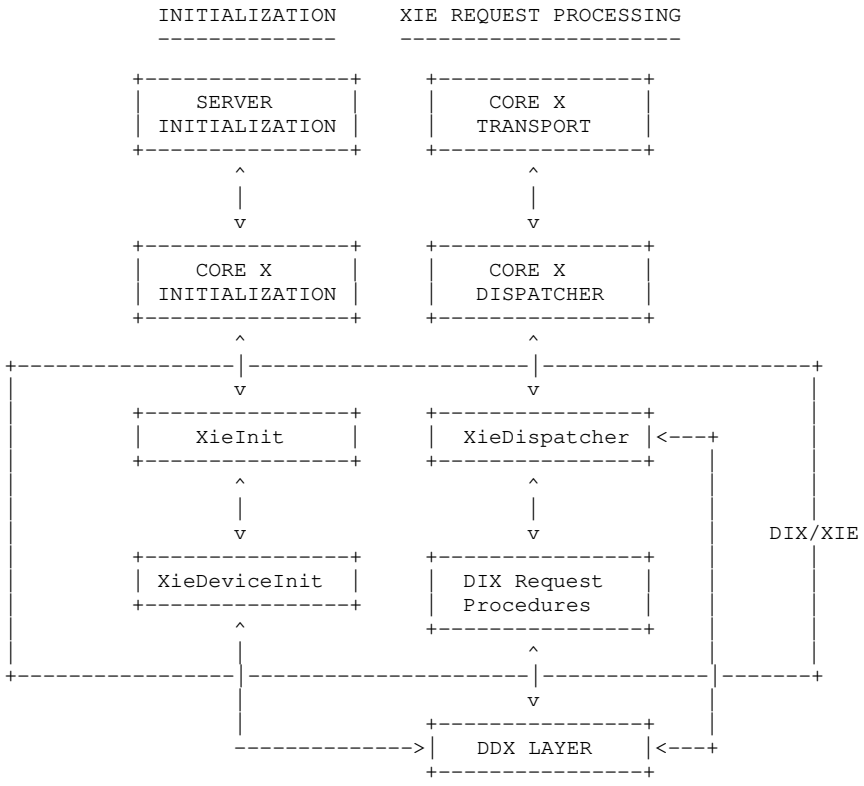
Chapter 2

DIX/XIE Component

This chapter describes the procedures that make up the DIX/XIE Server Component. All XIE client requests to some degree use procedures within this component. Figure 2–1 represents a block diagram of the procedures of the DIX/XIE Component and its relationship to CORE X and DDX/XIE functional components.

Note: See Appendix B for a list of modules and procedures within the DIX/XIE Component.

Figure 2–1: DIX Procedures



2.1 Initialization Procedures

Upon server initialization, the CORE X initializes all extensions by calling a CORE X procedure - InitExtensions. This procedure calls the initialization procedure of all extensions the CORE X will support. An addition will be made to InitExtension to call the XIE initialization routine. In this implementation, this procedure is referred to as XieInit.

Note: InitExtension and all XIE components will be linked with the CORE X in an OS dependent manner (see Chapter 4).

2.1.1 XieInit

This procedure will be called when CORE X initializes or resets itself. XieInit will define the XIE extension to the CORE X and will perform all extension resource and device initialization.

2.1.1.1 Making XIE Known to the CORE X

XIE must make the extension known to the CORE X so that XIE will be able to co-exist with other extensions. This is done by a call to a CORE X routine - AddExtension (EXTENSION.C):

AddExtension (Name,NumEvents,NumErrors,MainProc,SwappedMainProc,ResetProc)

This routine returns a pointer to a CORE X structure ExtensionEntry (see Appendix E) which defines extension parameters, including its base opcode (used by SendErrorToClient - see Section 2.2.1.1), base and last error codes and event codes. It should be noted that this structure also contains a private pointer which, in the future, XIE can use for its own purposes.

Name is a string indicating the name of the extension ("xImageExtension").

NumEvents is the number of events defined by XIE. (3)

NumErrors is the number of errors defined by XIE. (0)

MainProc specifies where the CORE X dispatcher will call when it sees an XIE request. For this implementation this will be named XieDispatcher. XieDispatcher is called with a pointer to the client structure (from the server transport). See Appendix E.

SwappedMainProc is called in place of MainProc if the client's byte ordering is different from the server. This routine must rearrange the bytes in place and then act like MainProc. (The CORE X modules SWAPREP.C, MISC.H, DIXUTILS.C contain useful macros and procedures for swapping). For this implementation this procedure will be called SwapXieDispatcher. This routine is called with a pointer to the client structure (from the server transport).

ResetProc is called when the server closes down (has no more clients). The purpose of reset is to return the server to the state it was in when it started up. After reset, the extension is no longer defined. For this implementation this procedure will be named XieReset. Specific functionality is to be determined later, however it is envisioned that for this implementation this routine may just return. For hardware related implementations, this routine will provide a mechanism in getting the hardware to an idle state.

2.1.1.2 Defining Errors and Events

XieInit must also define the XIE error and event codes using the errorBase and eventBase fields, respectively, within the ExtensionEntry structure. This is done by adding the XIE error and event constants (see Appendix C) to the extension error and event base, respectively.

For the events XIE defines, this module must load the CORE X vector table EventSwapVector[ ] with the XIE procedure to swap the event data. See Section 2.2.2.2.10.1 for a discussion on XIE events.

In addition, XieInit will define a mask for each XIE event (see Section 2.2.2.2.10.1).

The following are the event codes XIE will use:

- XIE_COMPUTATION_COMPLETE
- XIE_NEW_FRAME
- XIE_SEQUENCE_COMPLETE

Currently there are no XIE server error codes. All errors that the extension can generate will use an appropriate CORE X error code.

2.1.1.3 Defining XIE Resource Class and Types

XieInit must also define an XIE resource class and all the XIE resource types. To create an XIE resource class, XIE will use the CORE X procedure - CreateNewResourceClass (RESOURCE.C)

CreateNewResourceClass returns a new class (int) for use in creating new resources. This class will be referred to as RC_Xie.The format of the call is:

```
CreateNewResourceClass( )
```

To define XIE resource types, XIE will call the CORE X procedure - CreateNewResourceType (RESOURCE.C).

CreateNewResourceType returns a new resource type (int) which is also used in other resource routines. Each XIE resource (IOBs, IDCs etc.) will be associated with a type - RT_Iob, RT_Idc etc..

The format of this call is:

```
CreateNewResourceType ( )
```

See Section 2.2.2.2.3 for a description on the use of XIE resources.

2.1.2 XieDeviceInit

This procedure will initialize any hardware or accelerators the extension may use by calling the routine Init_xxx where xxx is the symbol for the particular accelerator. The mechanism to perform this is addressed in Chapter 4.

In addition, XieDeviceInit must load entries into the XIE/DDX procedure vector table for those functions which benefit from accelerators. As an example, if there is an accelerator which performs scale, XieDeviceInit must load the DDX scale entry with the entry to the DDX routine which takes the same arguments as the machine independent version and uses the accelerator to perform the scale. The mechanism to do this will be similar to the above.

2.2 DIX/XIE Request Processing

The DIX/XIE request procedures include the XIE dispatcher and a DIX procedure for each XIE protocol request. The latter is broken into two processing phases: Phase I and II.

2.2.1 XieDispatcher

XieDispatcher is called by CORE X when its dispatcher recognizes that the client's request is an XIE request. XieDispatcher's responsibilities are to:

- o Determine which function was requested
- o Swap the function request if necessary
- o Vector to the appropriate DIX Phase I processing procedure

CORE X calls XieDispatcher with a pointer to the client structure (see Appendix E). This structure represents the context of the client request and is used by the CORE X for its wire transport. This structure is used for any subsequent communication to the client and thus must be passed along to lower levels if the lower levels are to communicate with the client (macros and routines such as SendErrorToClient, WRITE_REPLY_TO_CLIENT and WriteToClient use a pointer to this structure as one of its arguments).

XieDispatcher's first job is to determine the XIE function requested. This is accomplished by using a CORE X macro - REQUEST (see Section F.1). REQUEST is called with a structure xieReq that exists in a XIE include file XIEPROTO.H. This structure represents the header of all XIE requests. See Appendix C for a list of this and other request structures.

The REQUEST macro defines a record called "stuff" which is of type xieReq, and which is bound to the request buffer. XieDispatcher initiates the appropriate DIX/XIE Request Processing function by case switching on the opcode represented in stuff->data. This represents the XIE request code. (Note: For XIE defined structures this field is represented by stuff->opcode.) If XieDispatcher does not recognize the requested opcode, then a call to SendErrorToClient is made with a CORE X error code - BadRequest and a "return(BadRequest)" is made.

XieDispatcher returns success or failure to the CORE X. It is expected that any errors generated by lower level routines will use SendErrorToClient and return the error code back to the dispatcher so that it can, in turn, return the error code back to CORE X.

2.2.1.1 Error Notification to Client

If an error is detected anytime during the dispatcher or the DIX processing phases, XIE must notify the client via a CORE X routine, SendErrorToClient (EXTENSION.C), and make a "return(errorcode)" back to the XIE dispatcher.

NOTE: When we state that the error must be reported, it is implied that a SendErrorToClient and a "return (errorcode)" is made.

SendErrorToClient routine has the following format:

SendErrorToClient (client,reqCode,minorCode,ResourceID,errorcode)

Client is the client structure passed to the extension with the request; reqCode is the base field of the ExtensionEntry structure returned by AddExtension performed during initialization in the XieInit component; minorCode is the extension request opcode (stuff->opcode); errorcode is one of the error codes assigned to the calling extension or a CORE X error; and resourceID is a parameter that the extension may use to return a resourceID that is involved in the error. If none is involved, the routine should pass NONE defined in the CORE X include file - X.h.

2.2.1.2 Swapping Requests

If the byte order on the server is different than the client, the CORE X, upon seeing an XIE request, will call SwapXieDispatcher. This routine will be responsible for swapping the request (in place) and then performing the same tasks as XieDispatcher. To ease the swapping task, several requests have pads within their request structure. This is to prevent having a word in the middle of a longword.

2.2.2 DIX Request Procedures

There exist two phases of DIX request processing. The first phase does the request length checking while the second phase manages the XIE resources, checks for valid arguments, and prepares data for DDX processing.

2.2.2.1 Processing of Requests - Phase I

Phase I request processing entails checking the request length. This is facilitated by the use of the following CORE X macros (see Section F.1):

- o REQUEST
- o REQUEST_SIZE_MATCH
- o REQUEST_AT_LEAST_SIZE

REQUEST is called with the structure of the particular request. This implies that a structure exist for each of the XIE protocol requests (see Appendix C). Calling this macro allows the extension to verify the length of the request via another CORE X macro: REQUEST_SIZE_ MATCH. This macro makes sure that the request in "stuff" is the right size for this particular request. It will return BadLength (back to CORE X) if the size is incorrect.

REQUEST_AT_LEAST_SIZE is the same as REQUEST_SIZE_MATCH except it is used for variable length requests. It checks to make sure that the request is no shorter than the minimum size. It returns BadLength if the size is invalid. See next section.

If no errors exist during request matching, then this component will initiate the second phase of DIX request processing. See Section 2.2.2.2.

2.2.2.1.1 Variable Length Requests

REQUEST_AT_LEAST_SIZE is used by requests such as XIEP_PUT_TILE which will have a variable length request size due to the tile dimensions. In this case the first phase of DIX processing checks the minimum size allowed and passes the responsibility of absolute length checking to the second phase of DIX request processing. The second phase will then read the request length and the dimension arguments and verify that they agree. If not, then a "BadLength" error is reported. Return is back to the first phase which, in turn, will return back to the XIE dispatcher.

2.2.2.2 Processing of Requests - Phase II

Depending on the request, the Phase II processing logic may be inline with the Phase I processing code (the length check code) or may initiate calls to subroutines. For the latter case, the procedures may reside in module XIE_UTILS.C (see Appendix B). For simple requests, such as resource create, they can be inline with the Phase I code, while for more complex calls it may be better to have a separate subroutine. This phase is responsible for the following functions:

- o Organization and sending of client replies.
- o Establishment of context for an XIE client/server session.
- o Resource creation, validation, deletion , modification and inquisition.
- o Preparation of services, data and/or resources for input into the CORE X or the DDX layer.
- o Organization and sending of events.

For DIX/DDX related tasks this phase may represent a functional subset of high level routines which interface to an image processing library. Such routines may in turn call a set of lower level image processing algorithms within the DDX layer.

Depending on the request, this phase of processing may not need procedures from the DDX component. In this case we say that the processing mode for this request is a DIX only type of process. If DIX/XIE calls procedures in the DDX component we say that the processing mode of the request is a DIX/DDX type of process.

2.2.2.2.1 XIE Replies

There exist several XIE requests which will generate replies to the client. The XIE macro - WRITE_REPLY_TO_CLIENT (see Section F.2) is used to send replies back to the requesting client. The format of the macro is:

```
WRITE_REPLY_TO_CLIENT (client, length, buf)
```

Client is the client structure; length is the length of the reply in bytes; and buf contains the reply. This macro will call the reply swap routine for this request (based on the request opcode represented by stuff->opcode) using XieReplySwapVector table if swapping is necessary (i.e. if client->swapped == TRUE).

XieReplySwapVector table consists of addresses of procedures which swap replies. For each request, there is a location within this table. Each entry/request in this table will either have a "NotImplemented" procedure which will return an error (indicating that the request has no replies) or an address of the reply swap procedure.

The format of the reply must match the protocol and is 32 bytes long. Each XIE request that generates a reply will have a corresponding structure, so length will be specified by sizeof(xReplyForThisRequest). See Appendix C.

There are several replies which have pads within their reply structure. This is to prevent having a word in the middle of a longword thus making the swapping tasks easier.

For replies of length greater than 32 bytes, XIE, in addition to calls to WRITE_REPLY_ TO_CLIENT, will call the CORE X routine - WriteToClient (see Section 2.2.2.2.5.4). XIE is responsible for byte swapping in the latter case. To accomplish this, XIE must first load the client structure field, client->pSwapReplyFunc, with the address of a function that swaps and writes to the client (via WriteToClient) and then use the XIE macro - WRITE_SWAPPED_ DATA_TO_CLIENT:

WRITE_SWAPPED_DATA_TO_CLIENT (client, length, pBuf)

Client is the client structure, length is the length in longwords, and pBuf is a pointer to the reply. This macro will check the client structure field, client->swapped, for true, and if so will call the function specified by client->pSwapReplyFunc else it will just write to the client via WriteToClient (see Section F.2).

Note: There exists swap utilities within CORE X and is expected that XIE will use these utilities. These reside in the CORE X modules SWAPREQ, SWAPREP.C, and others.

2.2.2.2.2 Session Services

This service establishes the bounds and semantics of the server/client XIE session.

2.2.2.2.2.1 XIEP_INIT_SESSION

The function of this request is for the client to determine if XIE is available, and if so, return information (via a reply) describing the extension and its functions back to the client. This function will enter the client ID and other client specific data into a static XIE table (client_ table). See Appendix D. The quantization context pointer and the event mask are examples of the client specific data necessary for XIE to maintain. For the former, memory for the quantize structure will be allocated.

2.2.2.2.2.2 XIEP_TERM_SESSION

This request will release the memory resources used by the requesting client and established at XIEP_INIT_SESSION. This involves deleting the client's entry from the client_table and deallocating the memory used by any of it entries.

Note: Other XIE resources which use the CORE X facility will be terminated upon the client closing the connection.

2.2.2.2.3 Resource Services

The DIX component is responsible for creating, validating, deleting, and querying XIE resources. These services require use of the CORE X resource management facility (RESOURCE.C).

2.2.2.2.3.1 Creating XIE Resources

Before XIE can create a resource, it must determine if the ID of the resource received from the client (via XIELIB) is a legal ID. This is accomplished by doing the following:

- o Verifying that the ID is not previously defined via LookupId (see Section 2.2.2.2.3.2)
- o Making sure that the ID "and" CORE X parameter SERVER_BIT is FALSE
- o Making sure the client structure field "clientAsMask" equals the upper 12 bits of the ID.

The XIE macro LEGAL_NEW_XIE_RESOURCE (see Section F.2) does the above:

```
LEGAL_NEW_XIE_RESOURCE (id, rType)
```

The id is the resource id and rType is the resource type (RT_Iob, RT_Idc etc.).

The CORE X procedure, AddResource, allows extensions to add a new resource class-value pair into the CORE X resource management given a resource ID (provided by client), class type (created by XieInit), and value (pointer to the just allocated resource).

For example, in creating an IOB, the client passes a resource ID, and the IOB resource data. XIE first checks to see if its a legal ID, then allocates memory for the IOB and passes the memory pointer (value) to AddResource. Now if the client some time later wishes to query this IOB, XIE will use the client passed ID as a handle to accessing the IOB data by using a CORE X routine LookupID (see Section 2.2.2.2.3.2) which returns the pointer.

AddResource will create the resource if it does not exist. The following represents the call to AddResource:

```
AddResource (ID, Type, Value, Func, RC_Xie)
```

Func represents the procedure to call when the resource is freed (see Section 2.2.2.2.3.3) and is called with the address of the resource data. There will be a delete procedure, which will de-allocate the memory used by the resource, for each XIE resource. Entries to these procedures will be DestroyIob, DestroyIdc etc.. These will reside in the XIE_UTILS.C module (see Appendix B.

Adding the resource will allow XIE to establish the location of XIE data from the resource ID (see next section).

2.2.2.2.3.2 Validating XIE Resources

The validation of XIE resources is done via a call to the CORE X procedure LookupID (in RESOURCE.C):

LookupID (ID, rType, RC_Xie)

Where rType is the resource type (RT_iob, RT_idc, RT_ipc) and RC_Xie is the XIE resource class. Both are defined in XieInit.

LookupId will return a pointer (value) if it is valid otherwise it returns a NULL. All extension requests having a resource ID will use LookupID as a validity check. If a Null is returned then a BadIDChoice error will be reported to the client.

Note: All requests which have an XIE resource ID will be validated during the processing phase of request processing.

2.2.2.2.3.3 Freeing XIE Resources

To free a resource, XIE must call FreeResource (ID,SkipDeleteCallClass). SkipDeleteCallClass indicates those classes the caller wishes not to free. For XIE related resources this will mean a value (not RC_Xie) where RC_Xie represents the class of XIE resources. This routine will call the delete function specified by "func" in the AddResource call.

2.2.2.2.3.4 Querying XIE Resources

Querying a resource requires that the resource be owned by the client requesting the resource information. Therefore, a validity check must be done before the attributes within the resource are read and reported back to the client. (See Section 2.2.2.2.3.2).

2.2.2.2.4 Import/Export Services

These tasks allow an image to be moved between a CORE drawable (pixmap or window) and an IOB. They depend on some procedures (GetImage and PutImage for example) from the CORE X in order to accomplish their task.

2.2.2.2.4.1 XIEP_IMPORT_IMAGE

This request will take image data from a pixmap or window and copy it to an IOB. The IOB must be valid or a BadIDChoice error will be reported (see Section 2.2.2.2.3.2). Functionally this request will do what the CORE X GetImage procedure does (calls CORE X routines GetImage and WriteToClient) without sending the data to the client. GetImage is called through the drawable (*pdraw->pScreen->GetImage) and transfers data within the drawable to a buffer. The buffer represents the image data. Depending on the current state of the IOB, DIX may have to allocate a larger or smaller buffer than is currently allocated for this IOB. In

addition, new IOB attributes are derived from the drawable specified within the client request. The format of GetImage is:

GetImage (pDraw, x, y, width, height, format, planemask, pBuf)

The arguments, with the exception of pDraw and pBuf, are derived from the drawable. pDraw and pBuf are the drawable passed by the client and the IOB data pointer, respectively.

The specified drawable must be valid or a BadDrawable error will be reported. To validate the drawable, XIE will use the CORE X macro LOOKUP_DRAWABLE:

pDraw = LOOKUP_DRAWABLE (drawable_id, client)

The drawable_id is the drawable ID; client is the client structure, and pDraw is a pointer to a drawable type. If pDraw is a NULL then the drawable is not valid and a BadDrawable must be reported. If the drawable is an input only drawable, a BadMatch error will be reported. (see Section F.1)

2.2.2.2.4.2 XIEP_EXPORT_IMAGE

This request transfers image data and its attributes from an existing IOB into a drawable. The IOB must be valid or a BadId error will be reported (see Section 2.2.2.2.3.2). The pixmap must be a valid drawable or a BadDrawable error will be reported. Similarly, the graphics context (also within this request protocol) must also be valid or a BadGc error will be reported.

To validate a drawable and a GC, XIE will use a CORE X macro VALIDATE_DRAWABLE_ AND_GC:

VALIDATE_DRAWABLE_AND_GC (dst_drawable_id, pDraw, pGC, client)

dst_drawable_id is the pixmap drawable Id (from the client); pDraw is a pointer to a drawable type (returned); pGC is a pointer to the GC (returned); and client is the client structure. This macro will return a BadDrawable, BadGc or BadMatch error if input is not valid (see Section F.1)

This procedure will use the Graphics Context, which the client must pass, and make calls through the GC to PutImage (pGc->PutImage). PutImage has the following format:

PutImage (pDraw, pGc, depth, dstX, dstY, width, height, left_pad, format, IOB_Image_ Data)

All arguments, with the exception of left_pad, format, and IOB_Image_Data, are within the request block as stated in the XIE Protocol. The left_pad, format, and IOB_Image_Data arguments are inferred from the specified IOB.

2.2.2.2.5 Transport Services

These services allow the client to transfer image data between the client and server. Once the client has established the transport context, clients can then move images to/from IOBs.

2.2.2.2.5.1 IOB Data Formats and Organization

The DIX will have the following IOB image data types (see Appendix D):

- o XieK_Bitstream
- o XieK_ByteImage
- o XieK_WordImage
- o XieK_LongImage
- o XieK_FloatImage

XieK_Bitstream represents a packed image where each pixel is one bit long. There will be no padding allowed either between pixels or scanlines. The start of the image data will always be byte aligned.

XieK_xxxImage represents image data in either 8,16 or 32 bit pixels with no padding pixels allowed between pixels or between scanlines. The start of the image data will always be byte aligned.

XieK_FloatImage represents image data using the server's machine representation of single precision floating point.

This implies that Transport Services must reorganize and reformat the data when data is transferred over the wire. For example, when the client transfers an image (to the server) which includes pixel and/or scanline strides, the DIX will be responsible in "weeding" out the data which will not be used in any subsequent rendering or DDX operation. In addition, the DIX/XIE must choose the integer type large enough to support the incoming data. Similarly, when the transfer is from the server to the client, the DIX will be responsible for packing if dictated by the transport context.

To accomplish this reformatting, the DIX may depend on DDX functions which will facilitate the processing of data from one organization to another.

The server will use the IOB attribute list to keep track of the various formats. See Section 2.2.2.2.6.1 for more on IOB data formats.

For multispectral image data, DIX/XIE must reorganize the data to ease subsequent processing. Thus when transferring multispectral images from the client, DIX/XIE will un-bundle the data into the specified IOB (parent IOB) and n-1 data entities (child IOBs) of data format byte, word or long, where n represents the number of spectral components. The parent IOB will have pointers to the child IOBs whose meaning is inferred by the spectral type (of the parent IOB). Thus clients can still access multispectral IOBs via the IOB id of the parent.

As an example, if client sends an RGB image of 3,3,2 bits, respectivily, DIX/XIE will use the destination IOB specified by the client as the red component of data type XieK_ByteImage and will create two "child IOBs" to represent the two components green and blue of data type XieK_ByteImage. Thus when the client wishes to scale this RGB image, DIX/XIE will recognize that this is a multispectral image and will initiate the DDX scale algorithm for each component - parent and children. This will allow DDX writers not to be concern with "difficult" data formats.

2.2.2.2.5.2 XIEP_SET_TRANSPORT

This request will establish the required context necessary to support image data wire transport operations (either direction). The context includes information about the data organization, compression scheme, and component information of the image as it exists on the client side. The DIX responsibility is to read the information within the request and attach it to the specified IOB.

The IOB field, structure XieTransport, will contain the information describing the data organization of the client image (see Appendix D). Thus, subsequent wire transporting of the data to/from an IOB will be fully described by the IOB fields.

2.2.2.2.5.3 XIEP_PUT_TILE

This request allows the client to randomly transfer client image data to an IOB via tiles. Prior to calling this request the client must have previously called XIEP_SET_TRANSPORT and must have allocated an IOB. This function will copy the data from the protocol request buffer to an allocated buffer within the IOB. Granularity, formats, size and location of the tiles etc. are provided by the arguments within the request and by the specified IOB fields.

The DIX responsibility is to:

- check that the IOB is valid using the IOB id passed by the client (see Section 2.2.2.2.3.2) and if not report a BadIDChoice error.
- check that the IOB is constrained and if not, report a BadMatch error.
- check that the transport field within the IOB has been previously defined (i.e. not NULL) and if not, report a BadAccess error.
- check that the component organization as specified in the IOB is consistent with that within the transport fields and if not, report a BadMatch error.
- set the IOB transport field to NULL on the last transfer.

In addition, the DIX will be responsible for changing the data format and organization from that specified in the IOB's XieTransport field to one of the DIX supported image constructs.

NOTE: If byte swapping is necessary (i.e client->swapped == TRUE), then the swapping must be done before reading from the client and storing to the specified IOB.

2.2.2.2.5.4 XIEP_GET_TILE

This request allows the client to randomly transfer IOB image data to a client buffer via tiles. Prior to calling this request the client must have previously called XIEP_SET_TRANSPORT and must have allocated an IOB. This function will send data to the client via a CORE X procedure WriteToClient:

WriteToClient (client, numBytes, pbytes)

Client is the client structure passed to XIE, numBytes are the numBytes passed to client, and pbytes is pointer to the bytes to pass.

Granularity, formats, size and location of the tiles etc. are provided by the arguments within the request and by the specified IOB fields.

The DIX responsibility is to:

- check that the IOB is valid using the IOB id passed by the client (see Section 2.2.2.2.3.2) and if not report a BadIDChoice error.
- check that the IOB is constrained and if not report a BadMatch error.
- check that the transport field within the IOB has been previously defined (i.e. not NULL) and if not, report a BadAccess error.
- check that the component organization as specified in the IOB is consistent with that within the transport fields and if not, report a BadMatch error.
- set the IOB transport field to NULL on the last transfer.

In addition, the DIX will be responsible for changing the data format and organization to that specified in the IOB's XieTransport field from one of the DIX supported image constructs.

NOTE: If byte swapping is necessary (i.e client->swapped == TRUE), then the swapping must be done before sending data to the client.

2.2.2.2.5.5 XIEP_PUT_STREAM

This request allows the client to sequentially transfer client image data to an IOB. This protocol request may be used for both uncompressed and compressed image data. Prior to calling this request the client must have previously called XIEP_SET_TRANSPORT and must have allocated an IOB. This function will copy the data from the protocol request buffer to an allocated buffer within the IOB. Granularity, formats, etc. are provided by the arguments within the request and by the specified IOB fields.

The DIX responsibility is to:

- check that the IOB is unconstrained and if not report a BadMatch error.
- check that the IOB is valid using the IOB id passed by the client (see Section 2.2.2.2.3.2) and if not report a BadIDChoice error.

- check that the transport field within the IOB has been previously defined (i.e. not NULL) and if not, report a BadAccess error.
- check that the component organization as specified in the IOB is consistent with that within the transport fields and if not, report a BadMatch error.
- set the IOB transport field to NULL on the last transfer.
- keep track of the offset into the IOB data area by using the "data_offset" field within the IOB (see Appendix D). This offset will be a bit offset when the IOB is of type XieK_ Bitstream and a pixel offset otherwise.

In addition, the DIX will be responsible for changing the data format and organization from that specified in the IOB's XieTransport field to one of the DIX supported image constructs. This may also entail decompressing the data; in this case the DIX will call the DDX function to perform the decompression.

NOTE: If byte swapping is necessary (i.e client->swapped == TRUE), then the swapping must be done before reading from the client and storing to the specified IOB.

2.2.2.2.5.6 XIEP_GET_STREAM

This request allows the client to sequentially receive server image data. This protocol request may be used for both uncompressed and compressed image data. Prior to calling this request the client must have previously called XIEP_SET_TRANSPORT. This function will send data to the client via a CORE X procedure WriteToClient (see Section 2.2.2.2.5.4).

Granularity, formats, etc. are provided by the arguments within the request and by the specified IOB fields.

The DIX responsibility is to:

- check that the IOB is unconstrained and if not report a BadMatch error.
- check that the IOB is valid using the IOB id passed by the client (see Section 2.2.2.2.3.2) and if not report a BadIDChoice error.
- check that the transport field within the IOB has been previously defined (i.e. not NULL) and if not, report a BadAccess error.
- check that the component organization as specified in the IOB is consistent with that within the transport fields and if not, report a BadMatch error.
- set the reply's final field to TRUE on the last transfer.
- set the IOB transport field to NULL on the last transfer.
- keep track of the offset into the IOB data area by using the "data_offset" field within the IOB (see Appendix D). The units of this offset will be dictated by the data type field (bit offset, byte offset, etc.).

In addition, the DIX will be responsible for changing the data format and organization to that specified in the IOB's XieTransport field from one of the DIX supported image constructs. This may also entail compressing the data; in this case the DIX will call the DDX function to perform the compression.

NOTE: If byte swapping is necessary (i.e client->swapped == TRUE), then the swapping must be done before sending data to the client.

2.2.2.2.6 Process Services

For these services, DIX will perform argument checking and if in error, report the appropriate error code. Also the DIX will check all the resources specified by the client as being valid (see Section 2.2.2.2.3.2) and if no errors exist, vector through the DDX component to perform the desired function. If the resource is in error, an appropriate invalid resource error is reported.

For those operations which require additional requirements on the client specified resources, the DIX will check that these resources are valid for the particular process and if not, report a BadMatch error.

If the client requested notification, the DIX will send an XIE_COMPUTATION_COMPLETE event upon the completion of the requested function (see Section 2.2.2.2.10.1).

One of the major tasks of the DIX for this request class is to prepare data to operate within the semantics of a DDX computation.

2.2.2.2.6.1 XIE's Handling of IOB Data Formats

Each process request will have a control flag indicating whether the operation should be "constrained" or "unconstrained". The latter implies that precision in the computation be maintained. For example, when adding two image objects, DIX/XIE will create a destination IOB whose format will either be integer (if the two IOBs pixel depth indicate that the result can fit into an integer format - 8,16, or 32 bits) or a float (if 32 bit integer representation cannot guarantee precision). The DIX responsibility is to determine this before initiating the DDX process. (The DDX process will read the format field within the specified IOB(s) and will operate accordingly.) Thus for some operations, DIX will be responsible for changing the allocation of an existing IOB's data to that of another type. Whenever possible, the DIX will call upon the DDX to compute using integer formats for efficiency reasons.

For cases when the DIX does not know the possible outcome, it will take the conservative approach (float) for the computation to assure precision.

For client requested "constrained" operations, the DIX function will generate positive integral data by quantizing before returning to caller. For example on subtraction, the DDX function could generate negative results but the DIX will force integral positive based on the quantization context and the DDX quantize function.

In addition, DIX must also call the quantize DDX function whenever it needs to constrain the data for operations which require such data (mask, bit oring etc.). The quantize context will dictate what method it will use to do this. See Appendix G for a breakdown on the DIX and DDX responsibilities for each of the DDX functions.

2.2.2.2.7 Statistics Services

These services allow the client to obtain statistics on an IOB. The DIX will check all the resources specified by the client as being valid (see Section 2.2.2.2.3.2) and if no errors exist, vector through the DDX component to perform the desired function. If the resource is in error, an appropriate invalid resource error is reported.

In addition, the DIX is responsible for checking the arguments for validity and if in error, reporting a BadValue or BadMatch error. If no errors exist, the DIX will call the appropriate DDX function and then return the statistical results in the form of a reply to the client (see Appendix C).

One of the major tasks for the DIX for this request class is to prepare data to operate within the semantics of a DDX computation. See Section 2.2.2.2.6.1 for information on the way DIX handles image data.

For the XIEP_AREA_STAT request if the client requested notification, the DIX will send an XIE_COMPUTATION_COMPLETE event upon the completion of the requested function (see Section 2.2.2.2.10.1).

2.2.2.2.8 Process Control Services - TBS

2.2.2.2.9 Display Service

This service allows the client to display a sequence of prerendered image frames which are stored as pixmaps. The XIEP_DISPLAY_SEQUENCE request is used to do this.

2.2.2.2.9.1 XIEP_DISPLAY_SEQUENCE

The processing of this request requires use of the CORE X procedure CopyArea which copies one drawable to another. Before XIE can initiate CopyArea it must:

- validate the client's destination drawable and GC
- validate the client's source drawable.
- check to see that the parameters of the source and destination drawable match.

To validate the drawable and GC, XIE will use the macro VALIDATE_DRAWABLE_AND_ GC (see XIEP_EXPORT_IMAGE above).

To validate the source drawable, XIE will use the CORE X macro LOOKUP_DRAWABLE (see Section 2.2.2.2.4.1).

Argument checking is done by assuring that pSrc->pScreen (pSrc is a pointer to the source drawable) equals pDst->pScreen (pDst is a pointer to the destination drawable) and that pSrc->depth equals pDst->depth, otherwise a BadMatch must be reported.

The format of the CopyArea procedure is:

CopyArea (pSrc, pDst, pGc, src_x, src_y, width, height, dst_x, dst_y)

The arguments src_x, src_y, width, height, dst_x and dst_y are inferred by the IDC during some preprocess and the other arguments are as defined above.

2.2.2.2.10 Event Services

Events are situations or occurrences which are detectable by XIE and are of interest to the client.

2.2.2.2.10.1 Event Notification to Client

There are three events defined in XIE. The CORE X routine, WriteEventsToClient, allows XIE to send events to interesting clients:

WriteEventsToClient (client, count, events)

Client is the client structure, count is the number of events to send (usually one), and events is a pointer to the particular event structure (see Appendix D). XIE will load the event structure with the pertinent data before invoking WriteEventsToClient.

If swapping is necessary, this routine will call EventSwapVector[code] for the particular event code (defined during XieInit). Therefore XIE must tell the CORE X about the routine which does the swapping for each event that XIE defines. This is done during the XieInit phase by setting the CORE X array entry EventSwapVector[code] to the XIE procedure that swaps that particular event code. This procedure is called with three arguments: client structure, the count (as stated above) and a pointer to the event(s). Each event swapping routine will have an entry name of the form SwapXxxEvent.

Only events that the client has expressed interest in will be sent. This is accomplished be using the client_table entry field - eventMask (see Appendix D). The DIX will use the client ID to find the client_table entry which will point to the eventMask for this client.

The event mask as defined by the protocol (client) will be stored within this table and will be used to indicate whether the client wishes for a particular event to be sent (see Appendix C). This mask is set by XIEP_SELECT_EVENTS.

2.2.2.2.10.2 XIEP_SELECT_EVENTS

Several XIE requests can generate events, however, an event will be sent to the client if and only if the client has expressed interest in the event. Thus any request which can potentially send an event will read the event mask from the client_table entry field to determine if it should send the event. The client is able to set this mask via a call to XIEP_SELECT_ EVENTS.

The XIEP_SELECT_EVENTS request will load the client passed event mask into the client_ table entry field - eventMask.

2.2.2.2.10.3 XIEP_QUERY_EVENTS

This request is for clients to determine which events it has expressed interest in being notified of. This request will return the event mask resident in the client_table back to the requesting client.

Chapter 3

DDX Component

This component represents the device dependent component of the extension.

For this implementation, entries into this subcomponent will comprise of a subset of lower level image processing routines which will be called from either the DIX Request procedures or from procedures within XIE_UTILS.C. Entry to DDX procedures is made via a table of entries called the procedure vector table.

Although this document does not address the machine dependent design, it should be stated that it is at this level where hardware related tasks get executed. That is, the procedure vectors will either be the entries to the lower level image processing routines and/or hardware routines to perform the various tasks. This implies that at server/extension initialization time, the extension must recognize the fact that it may have hardware resources, and if so, load the appropriate procedure vector. Since such routines must have knowledge about the particular device used, these reside within the DDX component and are said to be machine dependent.

Note: For severe hardware related errors, this level must set a client structure field, client->noClientException, to the errorcode. This will cause the CORE X scheduler to shutdown the client connection.

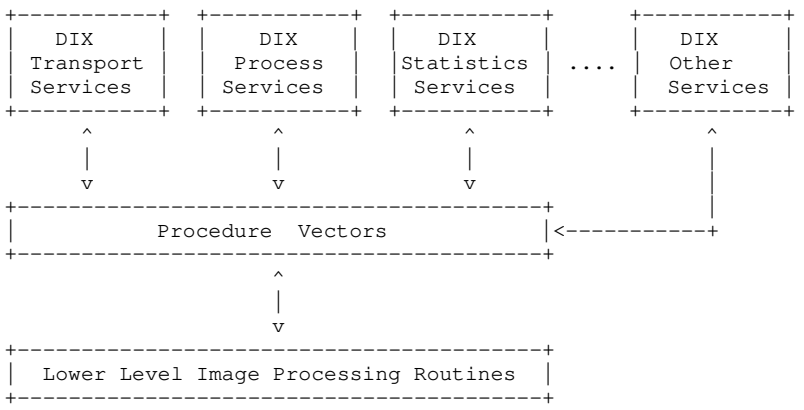
3.1 Procedure Vector Table

The procedure vector table consists of entries to two classes of XIE functions: low level math tools (e.g. add, sub etc.) and higher level functions such as decompression, scaling, image rotation etc.. Each of these procedures will define a minimum set of arguments necessary to carry out its function. The exact specification of the arguments will be determined later, however, it is envisioned that the image inputs to the DDX function will most likely pointers to a structure that defines the organization and type of data. To that end, each DDX function must support a set of recognized data types: XieK_Bitstream, XieK_ByteImage, XieK_WordImage, XieK_LongImage, and XieK_FloatImage. In particular, some permutations of integer and float must be supported. For example, ADD must support integer in and float out, float in and float out, etc. See Appendix G for a description of the DDX support of data formats/types.

The DDX functions will presume that the arguments as valid since it will be the DIX responsibility to perform argument checking.

Figure 3–1 outlines the flow of the DDX component in relationship to the DIX component.

Figure 3–1: DDX Subcomponents



The following represents the procedure table:

Table 3–1: DDX Procedure Vector Table

ROUTINE	FUNCTION
_POINT	remap input IOB to output IOB
_AREA	apply area operators to input IOB and store to output IOB
_ARITH	perform arithmetic operation on two IOBs or IOB and a constant and store to output IOB
_COMPARE	comparing two IOBs or one IOB and a constant and store boolean result to a 2D boolean bitmap
_LOGICAL	perform logical operation on two IOBs or one IOB and a constant and store to output IOB
_MATHF	apply mathematical function on input IOB and store to output IOB
_DECOMPRESS	decompress input IOB and store to output IOB
_SCALE	scale input IOB and store to output IOB
_ROTATE	rotate input IOB and store to output IOB
_MIRROR	flip input IOB and store to output IOB
_CROP	crop input IOB and store to output IOB

Table 3–1 (Cont.): DDX Procedure Vector Table

ROUTINE	FUNCTION
_TRANSLATE	overlay two IOBs or one IOB and constant and store to output IOB
_CONSTRAIN	apply quantization on input IOB and store to output IOB
_MATCH_HISTOGRAM	apply transformation on input IOB to conform with specified histogram form and store to output IOB
_CHROMINANCE_SEPARATE	create 3 IOBs from multispectral IOB
_CHROMINANCE_COMBINE	combine 3 IOBs to multispectral IOB
_DITHER	reduce number of levels of input IOB and store to output IOB
_LUMINANCE	extract luminance component of polychromatic IOB and store to output IOB
_CALC_HISTOGRAM	generates histogram of input IOB
_POINT_STATISTICS	generate statistics for an IOB
_AREA_STATISTICS	generate statistical IOB from input IOB
_DISPLAY_SEQUENCE	display time sequences of pixmaps
_SELECT_EVENTS	select which events to be notified of
_QUERY_EVENTS	query which events have been set for notification
_COMPRESS	compress input IOB - DIX private
_FLOAT_DATA	float buffer - DIX private
_MULTIPLE_REQUEST	hardware dependent routine

Some of these vectors represent those functions that may have accelerators to perform the client image request. If the accelerators exist, the DIX must load the appropriate procedure vector with the entry of the routine which uses the accelerators to perform the particular function. This will be done at XieDeviceInit time.

Note: These vectors represent a superset of those functions that are within the XIE protocol. There exist DDX functions which can not explicitly be called by the client (decompress, float_ data etc.)

For those functions (if any) that are not supported, XIE during initialization time (via XieInit) must insert a NotImplemented procedure which returns a BadRequest error.

The DDX component must always return to the DIX component so that status can return to the CORE X dispatcher.

3.1.1 Lower Level Image Processing Routines

For this implementation these procedures will be implemented by software. There will be no accelerators for this implementation.

3.2 DIX/DDX Interface

All DDX routines will use the protocol document as a functional guide. The semantics of operation (integer vs. float) will be determined by the data_type field within the specified IOB(s). All input constants will be represented by integer unless deemed otherwise.

Chapter 4

Operating System Considerations

This chapter addresses some operating system dependencies in the implementation of XIE.

4.1 Memory Management

Allocation, reallocation and deallocation of memory is performed by using the CORE X routines Xalloc, Xrealloc and Xfree, respectively.

4.2 Linking and Loading XIE

There exist two ways an extension can be loaded with the server: linked during the linking of the server and loaded at runtime using sharable images. This document has addressed the linked method since we intend to develop the initial version under Ultrix and Ultrix does not support sharable images.

4.3 Hardware Considerations

Although this document's goal is to address the machine independent implementation, we have attempted to address some of the implications of the current model on other implementations. As an example of this we have provided a multiple request function in the DDX procedure vector table. This is intended as a possible way for handling hardware devices that can perform several functions at once such as scaling and rotating. More importantly it illustrates how functionality may be increased with the addition of DDX vectors; the DDX procedure table is not necessarily bounded.

4.3.1 Hardware Recognition

XIE must identify, by some defined mechanism during XieInit, particular hardware platforms and the procedures for initializing the hardware and for performing the DDX processes. The CORE X has a similar scenario when it initializes itself. The CORE X for VMS, for example, utilizes system tables to identify the name of a device and then initializes the device by using a consistent naming scheme for the entry. For XIE, a similar mechanism will be employed. CORE routines InitOuput.C, Logical_Setup.C can provide some insight as to how this is performed in the VMS environment. Additional information will be provided once a machine independent model has been implemented. This model will need no such mechanism since no accelerators are defined.

4.3.2 Multiple Function Request

Due to hardware components which can accelerate several processes by pipelining data to its components, there will be a need for XIE to recognize a multiple function call. As an example of this, suppose there exists an accelerator which can scale and rotate with one call. XIE must have a DDX entry for such a call. This is accomplished upon initialization when XIE reads a file or table which fully describes the various multiple possibilities.

Note that these permutations only exist for upper level DDX functions such as scale and rotate. This reduces the number of combinations to a manageable set and represents a likely correspondence with display accelerator functions.

A mechanism will be defined such that XIE can determine the possible combinations so that it can report them to the client at INIT_SESSION time. This will be reported as a subcode (for the multiple request call) which represent the sequence of two or greater series of opcodes which indicate both order of functionality and the specific multiple capability. The procedure that implements this may be a operating system dependent function.

XIE at initialization time will enter this capability within an internal structure and upon seeing a multiple function request from the client, it will invoke the proper procedure based on the subcode.

Appendix A

XIE Protocol Requests

Table A–1: Image Extension Functions

ROUTINE	CLASS	MODULE	FUNCTION
_INIT_SESSION	1	DIX	begin client-extension server session
_TERM_SESSION	1	DIX	terminate client-extension server session
_QUERY_DEFAULTS	1	DIX	query operational defaults
_SET_DEFAULTS	1	DIX	establish/modify operational defaults
_CREATE_RESOURCE	2	DIX	create XIE resource
_COPY_RESOURCE	2	DIX	copy an XIE resource
_DELETE_RESOURCE	2	DIX	delete an XIE resource
_QUERY_RESOURCE	2	DIX	query an XIE resource
_IMPORT_IMAGE	2	DIX	transfer image data from drawable to IOB
_EXPORT_IMAGE	2	DIX	transfer image data from IOB -> PIXMAP
_SET_TRANSPORT	3	DIX	establishes context for image transfer
_GET_TILE	3	DDX	randomly transfers image data from IOB to client
_PUT_TILE	3	DDX	transfers image data from client to IOB
_GET_STREAM	3	DIX	transfers stream data from IOB -> client
_PUT_STREAM	3	DIX	transfers stream data from client -> IOB

Table A–1 (Cont.): Image Extension Functions

ROUTINE	CLASS	MODULE	FUNCTION
_POINT	4	DDX	remap input IOB to output IOB
_AREA	4	DDX	apply area operators to input IOB and store to output IOB
_ARITH	4	DDX	perform arithmetic operation on two IOBs or IOB and a constant and store to output IOB
_COMPARE	4	DDX	comparing two IOBs or one IOB and a constant and store boolean result to a 2D boolean bitmap
_LOGICAL	4	DDX	perform logical operation on two IOBs or one IOB and a constant and store to output IOB
_MATHF	4	DDX	apply mathematical function on input IOB and store to output IOB
_DECOMPRESS	4	DDX	decompress input IOB and store to output IOB
_SCALE	4	DDX	scale input IOB and store to output IOB
_ROTATE	4	DDX	rotate input IOB and store to output IOB
_MIRROR	4	DDX	flip input IOB and store to output IOB
_CROP	4	DDX	crop input IOB and store to output IOB
_TRANSLATE	4	DDX	overlay two IOBs or one IOB and constant and store to output IOB
_CONSTRAIN	4	DDX	apply quantization on input IOB and store to output IOB
_MATCH_HISTOGRAM	4	DDX	apply transformation on input IOB to conform with specified histogram form and store to output IOB
_CHROMINANCE_SEPARATE	4	DDX	create 3 IOBs from multispectral IOB
_CHROMINANCE_COMBINE	4	DDX	combine 3 IOBs to multispectral IOB
_DITHER	4	DDX	reduce number of levels of input IOB and store to output IOB
_LUMINANCE	4	DDX	extract luminance component of polychromatic IOB and store to output IOB
_CALC_HISTOGRAM	5	DDX	generates histogram of input IOB

Table A–1 (Cont.): Image Extension Functions

ROUTINE	CLASS	MODULE	FUNCTION
_POINT_STATISTICS	5	DDX	generate statistics for an IOB
_AREA_STATISTICS	5	DDX	generate statistical IOB from input IOB
_DISPLAY_SEQUENCE	7	DIX	display time sequences of pixmaps
_SELECT_EVENTS	8	DIX	select which events to be notified of
_QUERY_EVENTS	8	DIX	query which events have been set for notification

Table A–2: Service Classes

Class	Service
1	session services
2	resource services
3	transport services
4	process services
5	statistics services
6	process control services
7	display services
8	event services

Appendix B

Modules and Procedures

For coding purposes, all XIE/DIX procedures will reside in two modules: XIE.C and XIE_ UTILS.C.

The following is a list of XIE server related software modules:

- XieProto.h - contains definitions and structures that are used on both sides of the wire.
- XieAppl.h - contains public definitions and structures
- XieServer.h - contains definitions and structures that pertain to the server.
- XIE.C - contains XIE dispatch logic and DIX request logic
- XIE_UTILS.C - contains utilities for DIX Phase II Request Processing

The following represents the procedures within the XIE.C module:

- XieDispatcher - vectors to the appropriate DIX process - Phase I.
- XieInit - identifies XIE to the CORE X and initializes XIE
- XieDeviceInit - initializes any hardware devices and modifies the DDX procedure vector table.
- ProcXxx - perform DIX Phase I processing for request Xxx and depending on the request, perform Phase II processing or call procedures that reside in XIE_UTILS.C.

The following represents the type of procedures that will reside within the XIE_UTILS.C module:

- DestroyXxx - logic to delete an XIE resource Xxx.
- TransportTileImage - procedure to randomly transfer image data.
- TransportStreamImage - procedure to sequentially transfer image data.

Appendix C

XIE Application and Wire Definitions and Structures

The following represents a draft of some of the XIE wire definitions and structures that will reside in XieProto.h.

Note: this file represents a draft and may contain inconsistencies with the text and the referenced documents.

```
    /*
    ** Symbol XIEPROTO allows XieProto.h to be included multiple times.
    */
#ifndef XIEPROTO
#define XIEPROTO /* the "endif" MUST be the last line of this file   */

/*
**  Include files
*/
#ifdef VMS
#include    "XieAppl.h"   /* XIE inter-layer definitions     */
#endif
#ifdef ultrix
#include    "XieAppl.h"   /* XIE inter-layer definitions     */
#endif

/*
**  Wire protocol constant definitions.
*/
    /*
    ** Version number
    */
#define XieK_MajorVersion  1
#define XieK_MinorVersion  0

    /*
    ** Number of supported Events and Errors
    */
#define XieK_NumEvents   0
#define XieK_NumErrors   0

    /*
    ** Resource Types
    */
#define XieK_ImageObject  0
#define XieK_ImageContext  1
```

```
/************************************************************************
** Protocol request packet minor op-codes
************************************************************************/
#define X_ieInitSession                 1
#define X_ieSetTransport                2
#define X_iePutTile                     3
#define X_ieExportImage                 4
#define X_ieCreateResource              5
#define X_ieTermSession                 6
#define X_iePutStream                   7
#define X_ieLastRequest   99

/************************************************************************
** Protocol packet size definitions required by Xlibint.h
************************************************************************/
#define sz_xieReq       4
#define sz_xieInitSessionReq      8
#define sz_xiePutTileReq     28
#define sz_xiePutStreamReq     16
#define sz_xieExportImageReq     40
#define sz_xieSetTransportReq     36
#define sz_xieCreateResourceReq     12
#define sz_xieCreateIobReq     36
#define sz_xieTermSessionReq      4

/************************************************************************
** Miscellaneous protocol structure definitions
************************************************************************/
    /*
    ** Wire floating point format
    */
typedef struct {
    unsigned char   e_sign;
    unsigned char   m_sign;
    unsigned char   exponent;
    unsigned char   pad1;
    unsigned short  mantissa;
    unsigned short  pad2;
} XieFloat;

/************************************************************************
** Protocol request packets
************************************************************************/

    /*
    ** Generic part of all XIE requests
    */
typedef struct {
    unsigned char   reqType; /* major op-code assigned to extension     */
    unsigned char   opcode; /* minor op-code == XIE function     */
    unsigned short  length; /* length in longwords of entire request    */
} xieReq;

    /*
    ** Init Session Request
    */
typedef struct {
    unsigned char   reqType;
    unsigned char   opcode;  /* function == X_ieInitSession     */
    unsigned short  length;
    unsigned short  major_version; /* XIE major version */
    unsigned short  minor_version; /* XIE minor version */
} xieInitSessionReq;
```

```
    /*
    ** Term Session Request
    */
typedef struct {
    unsigned char   reqType;
    unsigned char   opcode;     /* function == X_ieTermSession     */
    unsigned short  length;
} xieTermSessionReq;

    /*
    ** Set transport Request (set transport parameters)
    */
typedef struct {
    unsigned char   reqType;
    unsigned char   opcode;  /* function == X_ieSetTransport     */
    unsigned short  length;
    unsigned long   iob_id;
    struct {
 unsigned long   scanline_stride;
 unsigned char   scanlines_abut;
 unsigned char   pixel_stride;
 unsigned char   pad[2];
    } comp_data[XieK_MaxComponents];
    unsigned char   compression_scheme;
    unsigned char   tile_format;
    unsigned char   component_organization;
    unsigned char   pad[1];
} xieSetTransportReq;

    /*
    ** Generic create resource
    */
typedef struct {
    unsigned char   reqType;
    unsigned char   opcode;     /* function == X_ieCreateResource     */
    unsigned short  length;
    unsigned long   resource_id;    /* resource identifier      */
    unsigned long   resource_type;  /* resource type       */
} xieCreateResourceReq;

    /*
    ** Create image object request
    */
typedef struct {
    unsigned char   reqType;
    unsigned char   opcode;  /* function == X_ieCreateResource   */
    unsigned short  length;
    unsigned long   resource_id; /* resource identifier      */
    unsigned long   resource_type; /* resource type == XieImageObject  */
    unsigned long   pixels_per_scanline;
    unsigned long   number_of_scanlines;
    XieFloat     aspect_ratio;
    unsigned char   number_of_components;
    unsigned char   bits_per_component[XieK_MaxComponents];
    unsigned char   component_mapping;
    unsigned char   pixel_progression;
    unsigned char   line_progression;
    unsigned char   polarity;
} xieCreateIobReq;
```

```
    /*
    ** Create IDC Request
    */
typedef struct {
    unsigned char reqType;
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of "whole" request */
    unsigned long resource_type;
    unsigned long idc_id; /* IDC identifier */
    unsigned char  idc_type; /* type of data for this dPC req */
}xieCreateIdcReq;

/*
** The following request structures are used with CreateIdc depending on
** idc_type
*/
    /*
    ** Roi type
    */
typedef struct {
    unsigned char pad[3];  /* get on 32 word boundary */
    unsigned long ul_x;
    unsigned long ul_y;
    unsigned long lr_x;
    unsigned long lr_y;
} xieIdcRoiReq;

    /*
    ** Cpp type
    */
typedef struct {
    unsigned char pad[3];  /* get on 32 word boundary */
    unsigned long ul_x;
    unsigned long ul_y;
    unsigned long iob_id;
} xieIdcCppReq;

    /*
    ** Tmp type
    */
typedef struct {
    unsigned char pad[3];  /* get on 32 word boundary */
    unsigned long centerx;
    unsigned long centery;
    unsigned long ListOfTmpEntries;/* no. of tmp entries (8 unsigned chars) that follow */
} xieIdcTmpReq;

    /*
    ** Import Image Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long iob_id;
    unsigned long drawable;
} xieImportImageReq;
```

```
    /*
    ** Export Image Request (copy image data: Iob --> drawable)
    */
typedef struct {
    unsigned char   reqType;
    unsigned char   opcode;  /* function == X_ieExportImage     */
    unsigned short  length;
    unsigned long   iob_id;
    unsigned long   drawable;
    GContext     gc;
    unsigned long   src_x;
    unsigned long   src_y;
    unsigned long   width;
    unsigned long   height;
    unsigned long   dst_x;
    unsigned long   dst_y;
} xieExportImageReq;

    /*
    ** General Query and Delete Resource
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode; /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long resource_id;
} xieQueryOrDeleteReq;

    /*
    ** Copy Resource
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode; /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long src_id;
    unsigned long dst_id;
} xieCopyResourceReq;

    /*
    ** Get Tile Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode; /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long iob_id;
    unsigned long x;
    unsigned long y;
    unsigned long width;
    unsigned long height;
    unsigned char final;
    unsigned char all_planes;
    unsigned char plane_num;
    unsigned char pad;
} xieGetTileReq;
```

```
    /*
    ** Put Tile Request (transport image data: client --> server)
    */
typedef struct _xiePutTileReq {
    unsigned char   reqType;
    unsigned char   opcode;  /* function == X_iePutTile     */
    unsigned short  length;
    unsigned long   iob_id;
    unsigned long   x;
    unsigned long   y;
    unsigned long   width;
    unsigned long   height;
    unsigned char   final;
    unsigned char   all_planes;
    unsigned char   plane_num;
    unsigned char   pad[1];
} xiePutTileReq;

    /*
    ** Put Stream Request (transport image data: client --> server)
    */
typedef struct _xiePutStreamReq {
    unsigned char   reqType;
    unsigned char   opcode;  /* function == X_iePutStream     */
    unsigned short  length;
    unsigned long   iob_id;
    unsigned char   final;
    unsigned char   all_planes;
    unsigned char   plane_num;
    unsigned char   number_of_lengths;
    unsigned long   image_len[1]; /* at run time [number_of_lengths]  */
} xiePutStreamReq;

    /*
    ** Get Stream Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long iob_id;
    unsigned char all_planes;
    unsigned char plane_num;
    unsigned char pad[2];
} xieGetStreamReq;

    /*
    ** Point Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long src_iob_id;
    unsigned long dst_iob_id;
    unsigned long ipc_id;
    unsigned char new_quantum[XieK_MaxComponents];
/* list of list follow */
} xiePointReq;
```

```
    /*
    ** Area Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long src_iob_id;
    unsigned long dst_iob_id;
    unsigned long ipcc_id;
    unsigned long ipct_id;
    unsigned char op1;
    unsigned char op2;
    unsigned char constrain;
    unsigned char pad;
} xieAreaReq;

    /*
    ** Arith Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long src1_iob_id;
    unsigned long src2_iob_id;
    unsigned long constants[XieK_MaxComponents];
    unsigned long dst_iob_id;
    unsigned long ipc_id;
    unsigned char op;
    unsigned char constrain;
    unsigned short pad;
} xieArithReq;

    /*
    ** Compare Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long src1_iob_id;
    unsigned long src2_iob_id;
    unsigned long constants[XieK_MaxComponents];
    unsigned long cpp_id;
    unsigned long ipc_id;
    unsigned char op;
    unsigned char combine;
    unsigned short pad;
} xieCompareReq;

    /*
    ** Logical Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long src1_iob_id;
    unsigned long src2_iob_id;
    unsigned long constants[XieK_MaxComponents];
    unsigned long dst_iob_id;
    unsigned long ipc_id;
    unsigned char op;
    unsigned char pad[3];
} xieLogicalReq;
```

```
    /*
    ** Math Function Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long src1_iob_id;
    unsigned long src2_iob_id;
    unsigned long ipc_id;
    unsigned char op;
    unsigned char constrain;
    unsigned short pad;
} xieMathfReq;

    /*
    ** Crop Image Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long src_iob_id;
    unsigned long dst_iob_id;
    unsigned long ipc_id;
} xieCropImageReq;

    /*
    ** Translate Image Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long src_iob_id;
    unsigned long constants[XieK_MaxComponents];
    unsigned long dst_iob_id;
    unsigned long ipc_id;
} xieCropImageReq;

    /*
    ** Scale Image Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long src_iob_id;
    unsigned long dst_iob_id;
    unsigned short pixels_per_scanline;
    unsigned short number_of_scanlines;
} xieScaleImageReq;

    /*
    ** Rotate Image Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* xie function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long src_iob_id;
    unsigned long dst_iob_id;
    XieFloat angle;
    unsigned char clip;
    unsigned char pad[3];
    unsigned long bits_per_component[XieK_MaxComponents];
} xieRotateImageReq;
```

```
    /*
    ** Mirror Image Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long src_iob_id;
    unsigned long dst_iob_id;
    unsigned char x_mirror;
    unsigned char y_mirror;
    unsigned char pad[2];
} xieMirrorImageReq;

    /*
    ** Constrain Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long iob_id;
} xieConstrainReq;

    /*
    ** Match Histogram Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long iob_id1;
    unsigned long iob_id2;
    unsigned long ipc_id;
    XieFloat param1;
    XieFloat param2;
    unsigned char h_shape;
    unsigned char   pad[3];
} xieMatchHistogramReq;

    /*
    ** Chrominance Separate Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long iob_id1;
    unsigned long iob_id2;
    unsigned long iob_id3;
    unsigned long dst_iob_id;
} xieChromeSepReq;

    /*
    ** Chrominance Combine Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long src_iob;
    unsigned long iob_id1;
    unsigned long iob_id2;
    unsigned long iob_id3;
} xieChromeComReq;
```

```
    /*
    ** Dither Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long src_iob_id;
    unsigned long dst_iob_id;
    unsigned char result_bits[XieK_MaxComponents];
    unsigned char pad;
} xieDitherReq;

    /*
    ** Luminance Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long src_iob_id;
    unsigned long dst_iob_id;
} xieLuminanceReq;

    /*
    ** Calculate Histogram Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long iob_id;
    unsigned long ipc_id;
    unsigned char by_component];
    unsigned char pad[3];
} xieCalcHistReq;

    /*
    ** Point Statistics Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long iob_id;
    unsigned long x_location;
    unsigned long y_location;
    unsigned char  edge_size;
    unsigned char   pad[3];
} xiePointStatsReq;

    /*
    ** Area Statistics Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long src_iob_id;
    unsigned long dst_iob_id;
    unsigned long ipc_id;
    unsigned char  edge_size;
    unsigned char  stat_type;
    unsigned char   pad[2];
} xieAreaStatsReq;
```

```
    /*
    ** Display Sequence Request
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long window_id;
    unsigned short frame_count;
    unsigned short frame_timing;
    unsigned char  forward;
    unsigned char  number_of_pixmaps;
} xieDisplaySeqReq;

    /*
    ** Events Requests
    */
typedef struct {
    unsigned char reqType; /* Extension type */
    unsigned char opcode;  /* XIE function */
    unsigned short length; /* length in longwords of whole request */
    unsigned long event_param;
} xieEventsReq;

/*****************************************************************************
** Replies
*****************************************************************************/

/* Reply for Get Image */

typedef struct {
    char type;        /* XIE_Reply */
    char pad;
    unsigned short sequenceNumber;
    unsigned long length;      /* length of returned image data
            in bytes*/
    unsigned long pad1[6];
} xieGetImageReply;

/* Reply for IOB query */

typedef struct {
    char type;        /* XIE_Reply */
    char pad;
    unsigned short sequenceNumber;
    unsigned long length;      /* 0 */
    unsigned short pixels_per_scanline;
    unsigned short number_of_scanlines;
    unsigned char  number_of_spectral_components;
    unsigned char  total_bits_per_pixel;
    unsigned char  polarity;
    unsigned char  grid_type;
    char   pad0;
    unsigned char  num_of_aspect_ratio;     /* will this exceed 32 chars ? */
    unsigned long pad1;
    unsigned long pad2;
    unsigned long pad3;
} xieQueryIobReply;

/* Reply for IPC/ROI query */
```

```
typedef struct {
    char type;        /* XIE_Reply */
    char pad;
    unsigned short sequenceNumber;
    unsigned long length;      /* 0 */
    unsigned char ipc_type;
    char pad1;
    unsigned short ul_x;
    unsigned short ul_y;
    unsigned short lr_x;
    unsigned short lr_y;
    unsigned short pad2;
    unsigned long pad3;
    unsigned long pad4;
    unsigned long pad5;
} xieQueryIpcRoiReply;

/* Reply for IPC/CPP query */

typedef struct {
    char type;        /* XIE_Reply */
    char pad;
    unsigned short sequenceNumber;
    unsigned long length;      /* 0 */
    unsigned char ipc_type;
    char pad1;
    unsigned short ul_x;
    unsigned short ul_y;
    unsigned long ipc_data;
    unsigned short pad2;
    unsigned long pad3;
    unsigned long pad4;
    unsigned long pad5;
} xieQueryIpcCppReply;

/* Reply for IPC/TMP query */

typedef struct {
    char type;        /* XIE_Reply */
    char pad;
    unsigned short sequenceNumber;
    unsigned long length;      /* length of entries */
    unsigned char ipc_type;
    char pad1;
    unsigned short ListOfTmpEntries;
    unsigned short centerx;
    unsigned short centery;
    unsigned long pad2;
    unsigned long pad3;
    unsigned long pad4;
    unsigned long pad5;
} xieQueryIpcTmpReply;

/* Each TMP entry uses the folowing structure */

typedef struct {
    CARD8 x_index;
    CARD8 y_index;
    unsigned short numerator;
    unsigned short denominator;
    unsigned char   positive;
    char   pad;    /* 32 bit boundary for swapping */
} xieTmpEntry;
```

```
    /*
    ** Reply for Init Session
    */
typedef struct {
    char     type;  /* XIE_Reply */
    unsigned char   success;
    unsigned short  sequenceNumber;
    unsigned long   length;  /* length in chars of additional data */
    unsigned long   service_classes;
    unsigned short  major_version;
    unsigned short  minor_version;
    char     vendor;
    char     model;
    unsigned char   video_digitizer;
    char     pad;
    unsigned long   accelerators;
    unsigned short  number_of_screens;
    char     pad1[6];
} xieInitSessionReply;

/* the following structure is used for screen data for CONNECTION reply */
/* one set per screen - 20 chars */

typedef struct {
    unsigned short  screen_width_millimeters;
    unsigned short  screen_height_millimeters;
    unsigned short  screen_width_pixels;
    unsigned short  screen_height_pixels;
    unsigned short  screen_depth;
    unsigned short  screen_luts_size;
    unsigned char   screen_number;
    unsigned char   screen_luts;
    unsigned char   screen_brightness_control;
    unsigned char   screen_contrast_control;
    unsigned char   screen_hue_control;
    char   pad;
    unsigned short pad1[2];  /* 32 bit boundary */
} screen_data;

/* Reply for Export */

typedef struct {
    char type;    /* XIE_Reply */
    char pad;
    unsigned short sequenceNumber;
    unsigned long length;  /* 0 */
    unsigned long pixmap;
    unsigned long pad1[5];
} xieExportReply;

/* Reply for Luminance Distribution */

typedef struct {
    char type;    /* XIE_Reply */
    char pad;
    unsigned short sequenceNumber;
    unsigned long length;  /* length in chars of stat data */
    unsigned long pad1[6];
} xieLuminanceDistributionReply;

/* the folowing structure is used for the above reply */
```

```
typedef struct {
    unsigned long minimum;
    unsigned long maximum;
    unsigned long mean;
    unsigned long std_dev;
    unsigned long variance;
} statistics_data;

/* Reply for Point Statistics*/

typedef struct {
    char type;    /* XIE_Reply */
    char pad;
    unsigned short sequenceNumber;
    unsigned long length;  /* length of point stat data that
        follows */
    unsigned long pad1[6];
} xiePointStatisticsReply;

/* the following structure is used for Point Statistics */

typedef struct {
    unsigned long pixel;
    unsigned long minimum;
    unsigned long maximum;
    unsigned long mean;
    unsigned long std_dev;
    unsigned long variance;
} point_statistics_data;

/* Reply for Dot Product */

typedef struct {
    char type;       /* XIE_Reply */
    char pad;
    unsigned short sequenceNumber;
    unsigned long length;     /* 0 */
    unsigned long pad1[6];
} xieDotProductReply;

/*** following display stuff may be deleted or expanded */

typedef struct {
    char type;       /* XIE_Reply */
    char pad;
    unsigned short sequenceNumber;
    unsigned long length;
    unsigned char idc_type;
    unsigned char  skip_or_show;
    unsigned long pixmap;
    unsigned short pad1;
} xieSingleStaticReply;

typedef struct {
    char type;       /* XIE_Reply */
    char pad;
    unsigned short sequenceNumber;
    unsigned long length;
    unsigned char idc_type;
    unsigned char map_x_tiles;
    unsigned char map_y_tiles;
    unsigned long current_frame;
    unsigned long number_of_display_frames;
} xieSpatialMapReply;
```

```
typedef struct {
    unsigned char skip_or_show;
    unsigned long pixmap;
    unsigned short pad1;
    char   pad2;
} display_frame;

typedef struct {
    char type;       /* XIE_Reply */
    char pad;
    unsigned short sequenceNumber;
    unsigned long length;
    unsigned char idc_type;
    unsigned char map_x_tiles;
    unsigned char map_y_tiles;
    unsigned long current_frame;
    unsigned long number_of_display_frames;
} xieTimeSequenceReply;

/*
**  This "endif" MUST be the last line of this file.
*/
#endif /* end of XIEPROTO -- NO DEFINITIONS ALLOWED BEYOND THIS POINT */
```

The following represents a draft of some of the XIE application definitions and structures that will reside in XieAppl.h.

Note: this file represents a draft and may contain inconsistencies with the text and the referenced documents.

```
    /*
    ** Symbol XIEAPPL allows XieAppl.h to be included multiple times.
    */
#ifndef XIEAPPL
#define XIEAPPL     /* The "endif" MUST be the last line of this file.

    /*
    ** X Image Extension name string
    */
#define XieS_Name     "xImageExtension"

    /*
    ** Implementation bounds
    */
#define XieK_MaxComponents 3   /* number of spectral components supported*/

    /*
    ** Spectral Component Mappings
    */
#define XieK_Bitonal  1
#define XieK_GrayScale  2
#define XieK_RGB  3

    /*
    ** Component Space Organizations
    */
#define XieK_BandByPixel        0
#define XieK_BandByLine  1
#define XieK_BandByPlane        2
#define XieK_BitByPlane         3

    /*
    ** IDC Types
    */
#define XieK_SingleStatic 0
#define XieK_SpatialMap  1
#define XieK_TimeSequence 2

    /*
    ** IPC Types
    */
#define XieK_CPP  0
#define XieK_ROI  1
#define XieK_TMP  2

    /*
    ** Compression Schemes
    */
#define XieK_PCM                0
#define XieK_G42D               3
#define XieK_DCT                5

    /*
    ** Line Progressions
    */
#define XieK_LP90  0
#define XieK_LP270  1
```

```
    /*
    ** Pixel Progressions
    */
#define XieK_PP0  0
#define XieK_PP90  1
#define XieK_PP180  2
#define XieK_PP270  3

    /*
    ** Polarities
    */
#define XieK_ZeroBright  0
#define XieK_ZeroDark  1

    /*
    ** Statistics Types
    */
#define XieK_Minimum  0
#define XieK_Maximum  1
#define XieK_Mean  2
#define XieK_StdDev  3
#define XieK_Variance  4

    /*
    ** Gray to Bitonal bit reduction methods
    */
#define XieK_Dither  0
#define XieK_Threshold  1

    /*
    ** Support Vector
    */
#define XieK_Transport  1
#define XieK_Enhancement 2
#define XieK_Pipelines  4
#define XieK_ImageCapture 8

    /*
    ** XieK_ Events
    */
#define XieK_ComputationEvent 1
#define XieK_NewFrameEvent 2
#define XieK_SequenceEvent 3
#define XieK_EdgeDetectEvent 4

    /*
    ** Edge Contact on Edge_detect Event
    */
#define XieK_North  0
#define XieK_East  1
#define XieK_South  2
#define XieK_West  3

    /*
    ** Conditional Operator Types
    */
#define XieK_LT   0
#define XieK_LE   1
#define XieK_EQ   2
#define XieK_GT   3
#define XieK_GE   4

/*
**  This "endif" MUST be the last line of this file.
*/
#endif /* end of XIEAPPL -- NO DEFINITIONS ALLOWED BEYOND THIS POINT */
```

Appendix D

Server Definitions and Structures

The following represents a draft of some definitions and structures that pertain to the XIE server. These will reside in XieServer.h.

Note: this file represents a draft and may contain inconsistencies with the text and the referenced documents.

```
#define XieK_MaxNumberOfClients 100

/* Image Data Types */

#define XieK_Bitstream  0
#define XieK_ByteImage  1
#define XieK_WordImage  2
#define XieK_LongImage  3
#define XieK_FloatImage  4
#define XieK_NoFormat  5

/* XIE Event Mask */

#define XieK_ComputationComplete    1<<0
#define XieK_NewFrame      1<<1
#define XieK_SequenceComplete     1<<2
#define XieK_EdgeDetect      1<<3


/*
** When a connection between the client and XIE is established, the following
** structure will have its fields defined: (TBDL)
*/

static struct client_struct
    {
    unsigned short event_mask;
    unsigned long client_id;
    unsigned long *quantize_ptr;
    } client_table[XieK_MaxNumberOfClients];

/*The following represents XIE resource structures. */

typedef struct components
    {
    unsigned long scanline_stride;
    unsigned char scanlines_abut;
    unsigned char pixel_stride;
    unsigned char pad[2];
} XieCompData;
```

```
typedef struct transport {
 unsigned char  compression_scheme;
 unsigned char  tile_format;
 unsigned char  component_organization;
        unsigned char  component_mapping;
 XieCompData comp_data[XieK_MaxComponents];
} XieTransportRec;

typedef struct transport *XieTransportPtr;

typedef struct IobResource {
    unsigned long pixels_per_scanline;
    unsigned long number_of_scanlines;
    unsigned char  number_of_components;
    unsigned char  pixel_progression;
    unsigned char  line_progression;
    XieFloat aspect_ratio;
    unsigned char  polarity;
    unsigned char constrained;
    XieTransportPtr transport_ptr;
    unsigned long *data_pointer[XieK_MaxComponents];
    unsigned long data_offset[XieK_MaxComponents];
    unsigned long data_type[XieK_MaxComponents];
    unsigned char bits_per_comp[XieK_MaxComponents];
} IobResourceRec;

typedef struct IobResource *IobPtr;

typedef struct IdcRoiResource {
    unsigned short  ul_x;
    unsigned short  ul_y;
    unsigned short  lr_x;
    unsigned short  lr_y;
} IdcRoiResourceRec;

typedef struct IdcRoiResource *Roi_ptr;

typedef struct IdcCppResource {
    unsigned short  ul_x;
    unsigned short  ul_y;
    unsigned long pixmap;
} IdcCppResourceRec;

typedef struct IdcCppResource *Cpp_ptr;

typedef struct {
    char  center_x;
    char  center_y;
    char  num_tmp_entries;
} IdcTmpResource;

/*
** The following structures represent those used for XIE events
*/

typedef struct {
    char type;       /* event code */
    char pad;
    unsigned short sequenceNumber;  /* client sequence */
    unsigned long iob_id;
    char   direction;      /* CLIENT_TO_SERVER or SERVER_TO_CLIENT */
    char    pad0[3];      /* pads for 32 char long event */
    unsigned long pad1[5];
} xie_transport_complete;
```

```
typedef struct {
    char type;       /* event code */
    char pad;
    unsigned short sequenceNumber;  /* client sequence */
    unsigned long src_iob_id;
    unsigned long dst_iob_id;
    unsigned long pad1[5];     /* pad for 32 char long event */
} xie_computation_complete;

typedef struct {
    char type;       /* event code */
    char pad;
    unsigned short sequenceNumber;  /* client sequence */
    unsigned long idc_id;
    unsigned long frame_number;     /* frame that is currently displayed */
    unsigned long pad1[5];     /* pad for 32 char long event */
} xie_new_frame;

typedef struct {
    char type;       /* event code */
    char pad;
    unsigned short  sequenceNumber; /* client sequence */
    unsigned long idc_id;
    unsigned long pad1[6];     /* pad for 32 char long event */
} xie_sequence_complete;

typedef struct {
    char type;       /* event code */
    char pad;
    unsigned short sequenceNumber;  /* client sequence */
    unsigned int  pixmap;
    unsigned long  window;
    unsigned long  pixmap_x;     /* upper left x pixmap location */
    unsigned long  pixmap_y;     /* upper left y pixmap location */
    char   edge;      /* NORTH, EAST, SOUTH, WEST */
    char    pad0[3];      /* pads for 32 byte long event */
    unsigned long  pad1[2];
} xie_edge_detect;
```

Appendix E

CORE X Structures

The following CORE X structure is defined in EXTNSIONST.H:

```
typedef struct _ExtensionEntry {
    int index;
    void (* CloseDown)();       /* called at server shutdown */
    char *name;               /* extension name */
    int base;              Definitions and    /* base request number */
    int eventBase;
    int errorBase;
    int errorLast;
    pointer extPrivate;
} ExtensionEntry;
```

The following CORE X structure is defined in DIXSTRUCT.H and represents the client structure that is used throughout XIE:

```
typedef struct _Client {
    int index;
    Mask clientAsMask;
    pointer requestBuffer;
    pointer osPrivate;                  /* for OS layer, including scheduler */
    Bool swapped;
    void (* pSwapReplyFunc)();
    int errorValue;
    int sequence;
    int closeDownMode;
    int clientGone;
    int noClientException;       /* this client died or needs to be killed*/
    DrawablePtr lastDrawable;
    int lastDrawableID;
    GCPtr lastGC;
    int lastGCID;
    pointer *saveSet;
    int numSaved;
    pointer screenPrivate[MAXSCREENS];
#ifndef ultrix
    REQSB Reqsb;                /* holds request buffer process info */
    pointer *pXportWriteBuffer;
#endif ultrix
} ClientRec;
```

Appendix F

MACROS

F.1 CORE X Macros

These macros will be copied from the CORE X source files (PROCMUMBLE.C and DIX.H) to XieServer.H. This redundancy will allow for cleaner looking code.

- o REQUEST
- o REQUEST_SIZE_MATCH
- o REQUEST_AT_LEAST_SIZE
- o LOOKUP_DRAWABLE

Note: There exists other useful macros within the CORE X modules SWAPREQ.C, DIXUTILS.C and MISC.H to name a few.

```
#ifndef ultrix
#define REQUEST(type) \
        register type *stuff = (type *)client->Reqsb.pCurrentReq
#else ultrix
#define REQUEST(type) \
        register type *stuff = (type *)client->requestBuffer
#endif ultrix

#define REQUEST_SIZE_MATCH(req)\
    if ((sizeof(req) >> 2) != stuff->length)\
         return(BadLength)

#define REQUEST_AT_LEAST_SIZE(req) \
    if ((sizeof(req) >> 2) > stuff->length )\
         return(BadLength)

#define LOOKUP_DRAWABLE(did, client)\
    ((client->lastDrawableID == did) ? \
     (DrawablePtr)client->lastDrawable :
         (DrawablePtr)LookupDrawable(did, client))
```

F.2 XIE MACROS

The following macros are XIE specific macros:

- o LEGAL_NEW_XIE_RESOURCE_
- o CHECK_XIE_ERROR_
- o VALIDATE_DRAWABLE_AND_GC
- o FIND_CLIENT_INDEX_
- o WRITE_REPLY_TO_CLIENT_
- o WRITE_SWAPPED_DATA_TO_CLIENT_

```
/*
**  MACRO definitions
*/
    /* This macro checks "value" for NULL and if so, sends errorcode indicated
    ** by "error_code" with the relevant resource "id".
    */
#define CHECK_XIE_ERROR_(value,id,error_code) \
    if (!value) \
 {\
 SendErrorToClient (client, xieReqCode, stuff->opcode, id, error_code);\
     return (error_code);\
 }

    /* This macro makes sure that the a drawable and GC is valid */

#define VALIDATE_DRAWABLE_AND_GC_(drawID, pDraw, pGC, client)\
    if ((client->lastDrawableID != drawID) || (client->lastGCID != stuff->gc))\
    {\
        if (client->lastDrawableID != drawID)\
 {\
         pDraw = (DrawablePtr)LookupID(drawID, RT_DRAWABLE, RC_CORE);\
     CHECK_XIE_ERROR_(pDraw, drawID, BadDrawable);\
     if ((pDraw->type == DRAWABLE_WINDOW) || \
  (pDraw->type == DRAWABLE_PIXMAP))\
         {\
         client->lastDrawable = (DrawablePtr)pDraw;\
         client->lastDrawableID = drawID;\
     }\
            else\
     {\
  SendErrorToClient (client, xieReqCode, stuff->opcode, drawID,\
   BadDrawable);\
              return (BadDrawable);\
     }\
        }\
        else\
     pDraw = (DrawablePtr)client->lastDrawable;\
        if (client->lastGCID != stuff->gc)\
 {\
     pGC = (GC *)LookupID(stuff->gc, RT_GC, RC_CORE);\
     CHECK_XIE_ERROR_(pGC, stuff->gc, BadGC);\
            client->lastGC = (GCPtr)pGC;\
            client->lastGCID = stuff->gc;\
        }\
        else\
            pGC = (GC *) client->lastGC;\
        if ((pGC->depth != pDraw->depth) || (pGC->pScreen != pDraw->pScreen))\
 {\
     client->lastGCID = -1;\
```

```
     client->lastGC = (GCPtr)NULL;\
     SendErrorToClient (client, xieReqCode, stuff->opcode, stuff->gc,\
   BadMatch);\
     return (BadMatch);\
         }\
    }\
    else\
    {\
        pGC = (GC *) client->lastGC;\
        pDraw = (DrawablePtr)client->lastDrawable;\
    }\
    if (pGC->serialNumber != pDraw->serialNumber)\
    { \
 ValidateGC(pDraw, pGC);\
    }

    /* This macro loops through the current number of clients that the
    ** is supporting and finding a match with "index".  If no match is
    ** found, an error is reported to the client.
    */

#define FIND_CLIENT_INDEX_(index)\
    for (index=0; index <= num_of_clients; index++)\
 {\
 if (client->index == client_table[index].client_id) break;\
     }\
    if (index == num_of_clients)\
 {\
 SendErrorToClient (client, xieReqCode, stuff->opcode, client->index, \
  BadMatch); \
     return (BadMatch);\
 }

    /* This macro checks that an XIE resource ID is valid else returns
    ** error to client
    */

#define LEGAL_NEW_XIE_RESOURCE_(id,rType)\
    if ((LookupID(id, rType, RC_xie) != 0) || (id & SERVER_BIT) \
 || (client->clientAsMask != CLIENT_BITS(id)))\
    {\
 SendErrorToClient (client,xieReqCode, stuff->opcode, id, BadIDChoice);\
        return(BadIDChoice);\
    }

    /* This macro verifies that a GC is valid else returns error to client*/

#define VERIFY_GC(pGC, rid, client)\
    if (client->lastGCID == rid)\
        pGC = client->lastGC;\
    else\
 pGC = (GC *)LookupID(rid, RT_GC, RC_CORE);\
    CHECK_XIE_ERROR_(pGC, rid, BadGC);\


#define WRITE_REPLY_TO_CLIENT_ (pClient, size, pReply) \
   if (pClient->swapped) \
      (*XieReplySwapVector[((XieReq *)pClient->requestBuffer)->data]) \
           (pClient, size, pReply); \
      else WriteToClient(pClient, size, (char *) pReply);

#define WRITE_SWAPPED_DATA_TO_CLIENT_ (pClient, size, pbuf) \
   if (pClient->swapped) \
      (*pClient->pSwapReplyFunc)(pClient, size, pbuf); \
   else WriteToClient (pClient, size, (char *) pbuf);
```

Appendix G

DIX/DDX INTERFACE

The following represent the DIX and DDX responsibilities regarding data formats for each DDX call. For constrained operations, the DIX will apply the quantize function upon completion.

Note: the format {i,j,k} refers to two input IOB data types (i,j) and its result (k), while the format {i,c,k} refers to one input IOB of type i and a constant (c), and {i,k} refers to one input IOB of data type i and a resultant IOB of data type k. The symbol c is presumed to be of type LONG, the symbol frac is the symbol for fractional constants, and the symbols i,j and k take on the following possible values:

XieK_Bitstream 0
XieK_ByteImage 1
XieK_WordImage 2
XieK_LongImage 3
XieK_Float 4

_POINT

DIX: create destination IOB of type indicated by the request parameter - new_quantum.
DDX: must be able to handle all permutations of data formats {i,k} with the exception of i,k = 4.

_AREA (Addition)

DIX: create destination IOB of type 4.
DDX: must be able to handle all permutations of data formats {i,4}.

_AREA (Multiplication)

DIX: create destination IOB of type 4.
DDX: must be able to handle all permutations of data formats {i,4}.

-ARITH (ADD_II)

DIX: create destination IOB of type k where k = max(i,j)+1.

DDX: must be able to handle all permutations of data formats {i,j,k} where k = max(i,j)+1.

_ARITH (ADD_IC)

DIX: create destination IOB of type 4.
DDX: must be able to handle all permutations of data formats {i,c,4}.

_ARITH (SUB_II)

DIX: create destination IOB of type 4.
DDX: must be able to handle all permutations of data formats {i,j,4}.

_ARITH (SUB_IC)

DIX: create destination IOB of type 4.
DDX: must be able to handle all permutations of data formats {i,c,4}.

_ARITH (SUB_CI)

DIX: create destination IOB of type 4.
DDX: must be able to handle all permutations of data formats {c,i,4}.

_ARITH (MUL_II)

DIX: create destination IOB of type k where k = max(i,j) + 1, unless i or j = 0, then k = max(i,j).
DDX: must be able to handle all permutations of data formats {i,j,k} where k = i or k = i + 1.

_ARITH (MUL_IC)

DIX: create destination IOB of type 4.
DDX: must be able to handle all permutations of data formats {i,c,4}.

_ARITH (DIV_IC)

DIX: create destination IOB of type 4.
DDX: must be able to handle all permutations of data formats {i,c,4}.

_ARITH (DIV_CI)

DIX: create destination IOB of type 4.
DDX: must be able to handle all permutations of data formats {i,c,4}.

_ARITH (DIV_II)

DIX: create destination IOB of type 4.
DDX: must be able to handle all permutations of data formats {i,c,4}.

_ARITH (MIN_II)

DIX: create destination IOB of type k where k = max(i,j)
DDX: must be able to handle all permutations of data formats {i,j,k} where k = max(i,j).

_ARITH (MIN_IC)

DIX: create destination IOB of type k where k = i.
DDX: must be able to handle all permutations of data formats {i,c,k} where k = i.

_ARITH (MAX_II)

DIX: create destination IOB of type k where k = max(i,j)
DDX: must be able to handle all permutations of data formats {i,j,k} where k = max(i,j).

_ARITH (MAX_IC)

DIX: create destination IOB of type k = max(i,c).
DDX: must be able to handle all permutations of data formats {i,c,k} where k = max(i,c).

_COMPARE

DIX: create destination IOB of type 0.
DDX: must be able to handle all permutations of data formats {i,j,0}.

_LOGICAL (AND_II)

DIX: create destination IOB of type k = max(i,j).
DDX: must be able to handle all permutations of data formats {i,j,k} where i,j and k are not 4.

_LOGICAL (AND_IC)

DIX: create destination IOB of type 3.
DDX: must be able to handle all permutations of data formats {i,c,k} where i = 0,1,2,3 and k = 3.

_LOGICAL (OR_II)

DIX: create destination IOB of type k = max(i,j).
DDX: must be able to handle all permutations of data formats {i,j,k} where i,j and k are not 4.

_LOGICAL (OR_IC)

DIX: create destination IOB of type 3.

DDX: must be able to handle all permutations of data formats {i,c,k} where i = 0,1,2,3 and k = 3.

_LOGICAL (XOR_II)

DIX: create destination IOB of type k = max(i,j).
DDX: must be able to handle all permutations of data formats {i,j,k} where i,j and k are not 4.

_LOGICAL (XOR_IC)

DIX: create destination IOB of type 3.
DDX: must be able to handle all permutations of data formats {i,c,k} where i = 0,1,2,3 and k = 3.

_LOGICAL (SHIFT_UP or SHIFT_DOWN)

DIX: create destination IOB of same type as input.
DDX: must be able to handle all permutations of data formats {i,j,k} where i,j and k are not 4.

_LOGICAL (NOT)

DIX: create destination IOB of same type as input.
DDX: must be able to handle all permutations of data formats {i,j,k} where i,j and k are not 4.

_MATHF (PWR)

DIX: create destination IOB of type 4.
DDX: must be able to handle all permutations of data formats {i,frac,4}.

_MATHF (LOG2, LN, LOG10)

DIX: create destination IOB of type 4.
DDX: must be able to handle all permutations of data formats {i,4}.

_MATHF (SQUARE)

DIX: create destination IOB of type k where k = i + 1, unless i = 0, then k = i.
DDX: must be able to handle all permutations of data formats {i,k} where k = i or k = i + 1.

_MATHF (SQUARE_ROOT)

DIX: create destination IOB of type 4.
DDX: must be able to handle all permutations of data formats {i,4}.

_SCALE

DIX: create destination IOB of type k = i.
DDX: must be able to handle all cases of data types of the form {i,k} where k = i.

_ROTATE

DIX: Create destination IOB of type k = i.
DDX: must be able to handle all cases of data types of the form {i,k} where k = i.

_MIRROR

DIX: Create destination IOB of type k = i.
DDX: must be able to handle all cases of data types of the form {i,k} where k = i.

_CROP

DIX: Create destination IOB of type k = i.
DDX: must be able to handle all cases of data types of the form {i,k} where k = i.

_TRANSLATE (II)

DIX: create destination IOB of type k where k = max(i,j).
DDX: must be able to handle all permutations of data formats {i,j,k} where k = max(i,j).

_TRANSLATE (IC)

DIX: create destination IOB of type k where k = max(i,c)
DDX: must be able to handle all cases of data types of formats {i,c,k} where k = max(i,c).

_CONSTRAIN

DIX: create destination IOB of type k where k is dictated by the quantize context.
DDX: must be able to handle all permutations of data formats {i,k} with the exception of k = 4.

_MATCH_HISTOGRAM (TBS)

_CHROMINANCE_SEPARATE

DIX: if input IOB is of type 4, quantize to IOB of type specified by the quantization context, then create IOBs of type 1,2 or 3 depending on component depth.
DDX: must be able to handle all types of input IOBs except 4.

_CHROMINANCE_COMBINE

DIX: If input IOBs are of type 4, quantize according to quantization context, create destination IOB of type 1,2, or 3 as dictated by the "quantized" input IOBs.
DDX: must be able to handle all types of input IOBs except 4.

_DITHER

DIX: if input IOB is of type 4, quantize to IOB of type specified by the quantization context, then create destination IOB of type indicated by "result_bits"
DDX: must be able to handle all cases of data formats {i,k} where i or k not equal to 4

_LUMINANCE (TBS)

_CALC_HISTOGRAM

DIX: if input IOB is of type 4, quantize to IOB of type specified by the quantization context.
DDX: must be able to handle all types of input IOB except 4.

POINT_STATS

DIX: NONE
DDX: must be able to handle all types of input IOBs.

AREA_STATS

DIX: Create destination IOB of type 4.
DDX: must be able to handle all types of input IOB i for {i,4}.