1 of 1

SAND93-4100
Unlimited Release
Printed December 1993

Distribution
Category UC-705

Extending the X Window System*

Joseph P. Brenkosh
December 23, 1993

Abstract

The X Window System was originally developed in 1984 at Massachusetts Institute of Technology. It provides client-server computing functionality and also facilitates the establishment of a distributed computing environment. Since its inception the X Window System has undergone many enhancements. Despite these enhancements there will always be a functionality desired in the standard released version of X that is not supported or commercially or academically available.

The developers of the X Window System have designed it in such a way that it is possible to add functionality that is not included in the standard release. This is called an extension. Extensions are one method used to develop a customized version of the X Window System to support a specialized application.

This report presents the mechanics of adding an extension and examines a particular extension that was developed at Sandia National Laboratories to support data compression in X Windows which was one aspect of the Desktop Video and Collaborative Engineering Laboratory Directed Research and Development (LDRD).

* This work performed at Sandia National Laboratories supported by the U.S. Department of Energy under contract DE-AC04-94AL85000.

Extending the X Window System

Table of Contents

Introduction	1
Brief History of the X Window System	1
Features of the X Window System	1
Elements of the X Window System	2
Server and Client	2
Libraries	4
X Internals	5
Requests	5
Replies	6
Events	6
Errors	6
Resources	6
Properties and Atoms	7
Software Structure of the X Window System	8
Extensions	11
Writing an Extension	13
Data Structures	13
Server Routines	18
Error Handling	20
Client Routines	21
Summary	23
Acknowledgments	23
Appendix A	24
Appendix B	27
Appendix C	30
Appendix D	44
Bibliography	56

List of Figures

1	Clients Executing on a Server	3
2	Directory Structure of the X Source Code	8
3	The Extensions Directory	11
4	An Original Image	28
5	An Image After Applying the Alternate Scan Line Algorithm.	29

Introduction

The X Window System is widely used and supported on a variety of machines ranging from PCs to supercomputers. It has many features which helped it gain widespread acceptance. One of these features is extensibility which allows a customized version of the X Window System to be developed to support a specialized application. This report details the steps necessary to add an extension. It begins with an introduction to the X Window System, then examines the underlying X protocol and the software that comprises it. Next, it presents extensions and the necessary procedures to write an extension. The procedures and examples described are based on an extension developed at Sandia National Laboratories to support data compression as part of the Desktop Video and Collaborative Engineering LDRD.

Brief History of the X Window System

Before discussing the features, fundamentals, and software of the X Window system, a brief look at its development is in order.

One of the most significant events to revolutionize the computer industry in recent years was the development and widespread use of the X Window System, or X as it is more commonly called. X was developed in the mid-eighties at MIT. The original goal was to permit users easy access to computing resources that were remote and in geographically diverse locations. X accomplished these goals and gained widespread acceptance and was adopted as a standard by most major workstation manufacturers.

There have been many versions of X, and each new version provided some new functionality and bug-fixes over its predecessor. In February 1987 the alpha test of Version 11 began and in September 1987 MIT made the first public release of Version 11 (X11R1) available [5]. This version has been revised several times and the current release at the time this report is being written is Release 5 or X11R5.

Although originally developed at MIT, control of X passed in January 1988 to the X Consortium which is a group of workstation manufacturers who adhere to or support the X Window System standard.

Features of the X Window System

X has many features that enabled it to achieve its popularity. The more significant features are discussed in the following list.

1. It is portable. This allows programs to execute on a wide variety of machines. With many workstation vendors supporting X, applications will work on a variety of machines usually with only recompilation. With other vendor proprietary systems, most graphical applications would be restricted to a

particular platform of a vendor and require major rewrites to be ported to a new platform.

2. Most X applications use a graphical user interface which is based on pointing to an object on the screen with a mouse or similar type device. In most cases, this reduces the amount of manual command entry (typing), which reduces errors and also the amount of time necessary to become proficient with an application.
3. Because X is network based, users can access applications on remote machines (clients) and display the application on the local machine (server), which gives the user the benefit of remote resources not available on the local machine such as applications, computing power, storage, etc. It also allows users to execute multiple applications simultaneously in different windows on one workstation.
4. It is extensible by building toolkits, which are libraries that call the existing standard X library. Unfortunately, these toolkits many times are vendor specific thus making applications less portable. A good example of this is AT&T's Open Look which is used extensively on most workstations from Sun Microsystems and the Open Software Foundation's Motif which is used on workstations from Digital Equipment Corporation. Unless a vendor makes the commitment to support both, Open Look applications would not work on a Motif based system or vice versa. With the recent decision by Sun Microsystems to support Motif, these incompatibilities may become less frequent until a new toolkit or windowing system such as Microsoft NT challenges X. X can also be extended by making additions to the client and server routines.
5. Much of the underlying network housekeeping is transparent to the user and therefore the user need not be concerned with the networking details.

Elements of the X Window System

The X Window System is comprised of software which is designed to operate on bitmapped graphics displays.¹ It supports both monochrome and color displays. A display in X terminology consists of a workstation with a keyboard, a mouse or other pointing device, and one or more screens which are controlled by a single user.

The software that comprises X consists of two main programs which are the client and the server. There are also additional libraries which include various routines and data such as bitmaps and fonts.

Server and Client

The X Window System program which controls the display and takes input from the keyboard and mouse is called the server. A server can control more than one screen. A

¹ Bitmap or raster graphics is a technology which associates each dot on a screen with one or more bits in the memory of a computer. Applications modify the display by writing to display memory.

set of screens for a single user is called a display. A window is a rectangular section of a screen which shows the client application.

The client which is a user application program is the other main component of X. Clients may execute locally on the same machine as the server if it supports multitasking or on a remote machine via a network. This feature is not mutually exclusive. It is possible to have some clients executing on the same machine as the server (locally) and others to be executing on remote machines over a network. With a non-multitasking system such as a PC, all clients must execute on a remote system. An example of remote and local clients being displayed on a server is illustrated in Figure 1.

Figure 1. A user on the server machine can execute client programs on machine A and machine B and display the results on the server machine in windows CA and CB respectively. The server will also allow local clients to be displayed as denoted by window LC.

A client does not actually determine where its window will appear or its size. Another client program called the window manager controls these actions. The window manager is just another program which uses the standard X library routines but by convention is given certain privileges. It allows the user to move and resize windows, start new applications, and control the stacking of windows on the screen. These actions must adhere to the layout policy of the window manager. The window layout policy is a set of rules that specifies allowable sizes of windows and icons.²

² An icon is a small marker window that indicates that a larger "main" window exists and is available but is not currently mapped on the screen.

Libraries

The X Window System also uses other routines and data which can be considered libraries. The most important of these is called Xlib. It supplies routines for connecting a client to a particular server, creating windows, drawing graphics, and responding to events. Other higher level libraries are built using Xlib routines. An example is the Xt library which is called the toolkit intrinsics library and contains a collection of routines a developer can use to construct user interfaces, etc. Vendors often build their own libraries such as Sun's Open Look Intrinsics Toolkit. Toolkits make the job of the programmer easier by providing predefined functions which draw objects such as pushbuttons, menus, etc. which are called widgets.

Most clients also use data in the form of a bitmap which is a two dimensional array of bits used for many purposes including fonts, cursor definition, and templates for pictures of two colors. Data can also be in the form of a pixmap. A pixmap is a three dimensional array of bits. A pixmap can normally be thought of as a two dimensional array of pixels where each pixel has a value from 0 to $2^N - 1$ where N is the depth (z axis of the pixmap); or more simply a stack of N bitmaps. The pixel values are an index into a colormap containing the corresponding RGB (red, green, and blue) values. Each pixel or picture element on a color display has red, green, and blue phosphors. When an electron beam strikes these phosphors, the relative intensity of these three colors "fools" the eye into thinking it sees a single color. For example, a pixel value of 10 may map to a red value of 255, a green value of 255, and a blue value of 0. This pixel on the screen would appear to the eye as yellow [2].

Both changeable and read-only or immutable colormaps are used in X. The advantages of immutable colormaps are that they can be shared by clients. They also make it possible to calculate pixel values from the colors desired because the mapping between pixel values and colors is predictable. This is required for smooth shading and three dimensional algorithms. The disadvantage of immutable colormaps is that they may not contain the exact color that the client requested. It is possible to have an immutable colormap with several extra read-write colorcells that can be used to obtain special colors.

Image data is stored in three main formats in X; they are XYBitmap, XYPixmap, and ZPixmap. If the data is in the form of an XYBitmap, each pixel is represented by a single bit. The bitmap is in scan line order, with each scan line made up of multiples of the bitmap unit and padded with meaningless bits if necessary. In the case of data in an XYPixmap, each plane is represented as a bitmap, and the planes appear in most-significant to least-significant bit order, with no padding between planes. Data in a ZPixmap is similar to an XYBitmap except that pixels not bits are stored in scan line order. Each pixel is represented by a set of N consecutive bits, which indexes into a colormap.

X Internals

In order to gain a better understanding of X it is necessary to examine the interactions between the client and server. By using a sample session the various requests, replies, events, etc. which define the X protocol can be explained.

For the purposes of this example the client program will be executing on a remote machine. The user, who is at the local machine which is executing the server, logs on to the remote machine using the network. The user then sets the DISPLAY variable to the machine name, server number, and screen in the following format *host:server.screen*. An example would be *sads004:0.0*. Setting the DISPLAY variable signals to the client to display the client window on a specific host and screen. The machine name is then translated into an IP or DECnet address using the */etc/hosts* file or some address resolution program. On machines using an IP network the client then sends information to TCP port 6000 + n on the host, where n is the display number starting at zero. The host can either accept or reject the request for the connection. If the connection is successful, the server executing on the host sends back information about itself which is stored in a data structure used by the client. If it fails a reply with a reason is sent back to the client [1].

Requests

The information sent back to the client if the request for connection was successful describes each of the server's attached screens (if more than one) separately. This information is called a "visual" in X terminology. Screens can be used in a variety of ways such as using a color screen to display windows in white and black.

The client will then issue a *CreateWindow* request. Most requests have a similar format which includes a block of data containing an 8 bit major opcode, a number of fixed length parameters, and possibly a parameter of varying length. The request also contains a length field which defines the total length of the request in units of 4 bytes. The length field must have a value that is equal to the actual length of the request or an error is generated by the server. Some requests require a corresponding reply from the server and some do not. The *CreateWindow* request has no reply from the server and in this case the Xlib client library queues the request in a buffer. When the buffer is filled or when the request requires a reply, the Xlib client library will send the request(s) to the server. The designers of X implemented it in this fashion to be more efficient rather than sending one request at a time. Actual code for generating requests will be examined in the "Writing an Extension" chapter and in Appendices B and C.

Replies

If the request requires a reply, such as the *AllocColor* request which requests a read-only color from a color map, the reply will have a reply opcode which will be equal to 1 if there are no errors or 0 if there are errors. The reply will also contain a sequence number which is assigned sequentially to each request and also the requested information.

Events

The server can also send events to the client. An event is an action such as a key being pressed or mouse movement. The event data structure is 32 bytes long and contains an 8 bit type code. The most significant bit in this code is set if the event was generated from a *SendEvent* request. The client determines what events it is interested in by setting an event mask which filters those events in which the client is not interested. Those events are discarded and not even sent to the client. This filtering of events improves overall performance because it reduces the time clients spend processing unwanted events and also reduces the amount of traffic on the network. A client cannot begin drawing into a window until the window is mapped with the *XMapWindow* or related routine and the client receives an event which tells the client that it has permission to draw in the window. This event is called an *Expose* event.

Errors

Not every request from the client will cause the server to generate an error reply if there was something wrong with the request; however, when an error is generated, it is 32 bytes long with an 8 bit error code. The error reply also includes the major and minor opcode of the failed request and the least significant 16 bits of the request.

Resources

Besides the previously mentioned queuing of commands to reduce the amount of network traffic, X also uses other abstractions called "resources" to improve performance. One such abstraction is a "graphics context" or "GC". A graphics context contains information concerning the foreground and background color of pixels, line style, etc. This is sent once from the client to the server and graphics commands are executed by first applying the graphics context. If certain parameters of the graphics context need to be changed, only those parameters need to be sent, not the whole graphics context. These abstractions are given an integer ID by the server. This system of establishing an integer ID is also used with windows, colormaps, cursors, pixmaps, and fonts. Font libraries are usually kept on the server and clients need only specify the desired font instead of having to send the fonts to the server.

Most graphics commands that are issued by the client are instructions to the server to perform a particular operation at certain coordinates. The X server does the rest of the work. An example is the command *XDrawLine*. Only the x and y endpoints are sent to the server. One of the areas that does require the client to transmit large amounts of data to the server is image processing. An example of such is the *XPutImage* command which sends a bitmap or pixmap to the server to be displayed. As of X11R5, there was no way to do this other than to send the complete bitmap or pixmap. This report presents an alternative to sending all of the image by adding an extension. The next release of X (X11R6) will also incorporate methods to reduce these data transfers between client and server.

Properties and Atoms

A "property" is a packet of information about a window which is made available to all clients using a particular server and allows them to communicate with each other. Properties are used by clients to store information that other clients might need or want to know. An "atom" is a property identifier that is similar to a resource ID but also requires a window to uniquely identify the property. One important use of properties is to communicate information between applications and the window manager.

Software Structure of the X Window System

Now that the basic elements and operation of X has been explained, the actual software hierarchy as distributed by MIT can be discussed. For actual instructions on obtaining and building X see Appendix A.

Figure 2 illustrates the structure of the X source code.

Figure 2. The directory structure of the X source code as distributed by MIT.

Figure 2 shows that the top level directory that contains the server and clients is named **mit**. The **mit** directory contains many subdirectories and files. The more important ones are shown above and will now be described.

config A directory which contains various files which provide information such as the machine type, operating system version and release, etc. This information is used by the various makefiles³ when compiling the X software. It also contains files which allow the user to build all of the X release or just the desired sections such as the server, fonts, extensions, options for installing X manual pages, etc.

clients A directory which contains various subdirectories for client programs provided by MIT. One of the most popular is *xterm* which provides a terminal emulation window on the display of a user.

³ A makefile is a datafile which has compiler instructions and compilation dependencies used as input to a program named *make* which is used to compile software packages.

Another popular client is *xclock* which draws a clock that can be viewed on the display of a user.

server A directory which contains the necessary source code to build a server. It contains the following subdirectories.

ddx A directory which provides the device dependent layer which contains routines that are specific to various hardware such as graphic and input devices. In Figure 2 the subdirectories that are listed under **ddx**, i.e. **dec** and **sun**, contain device specific code for machines from Digital Equipment Corporation and Sun Microsystems, although equipment from other vendors is also supported. The **cfb** subdirectory contains routines useful for color frame buffers (devices for drawing on a screen), which use more than one bit to represent each pixel. The **mi** subdirectory which is somewhat out of place in the MIT distribution contains the machine independent routines which are used to implement a nearly complete **ddx** set of routines in as general a fashion as possible. These routines can perform all graphics requests in a frame buffer as long as certain basic routines are provided.

dix A directory which contains routines that do not depend on graphics hardware, input devices, or the host operating system. These routines manage the event queues, distribute events to clients, manage data structures visible to clients, and direct other parts of the server. Events such as key presses and mouse movement are collected by the **ddx** routines and transferred to the **dix** routines for processing. The **dix** routines determine what clients (if any) will receive X events.

include A directory which contains header files used by the server routines.

os A directory containing code that is operating system specific. The routines in this directory manage client connections, perform memory allocation, map colors into RGB intensities, and other operating system specific tasks.

fonts A directory which contains font handling routines and font data. As of X11R5 it also includes source code for the font server which can be used by the server to access fonts, although the X server can still obtain fonts the traditional way without using the font server.

extensions A directory containing MIT supplied extensions. It will be discussed in detail in the next two chapters.

lib A directory containing the necessary source and header files to build libraries used by X. Two important subdirectories are **X** and **Xt**. The most common library is **libX11.a** which is the filename of the Xlib library in subdirectory **X** which supports all of the communication between the client and the server and other utilities used by the client such as drawing commands. Another popular library is **libXt.a** in subdirectory **Xt** which is the toolkit intrinsics library. This is a set of higher level routines which call Xlib routines to draw objects called widgets such as pushbuttons, scroll bars, etc. It also performs operating system specific initializations for such entities as error logs and data space available to a particular variable.

include A directory containing header files used by X routines.

Makefile The top-level makefile which is used to call all of the other makefiles in the various subdirectories when building X.

doc This directory contains documentation in *troff*⁴ format for most of the various components of X including the server, Xlib, protocol, extensions, etc.

The other directories not listed consist of release notes, bug-fixes, demos, etc. and are not relevant to this report but are indeed a part of the standard X release.

⁴ Troff is a text formatting program originally developed for the UNIX Operating System.

Extensions

Extensions are additions to the X server that enable it to respond to non-standard client requests which the standard X server would not be able to recognize. Extensions require additions to both the X server routines and client side libraries. Clients must also be initially designed for the extension or modified if they will use the extension.

Extensions are usually developed when there is a functionality desired in the server that can not be met efficiently or sometimes at all by other means. For example, if a new type of pushbutton widget was needed this would be easier done by adding a toolkit library routine rather than writing an extension. Support for spline curves, display Postscript, or three dimensional graphics would probably require an extension. Compressing or reducing the amount of pixel data transmitted between client and server would also require an extension.

The **extensions** directory included in the X release is illustrated below.

*Figure 3. The **extensions** directory.*

The **extensions** directory contains the following subdirectories and files.

- server** This subdirectory contains the code for the MIT supplied extensions including the PEX three dimensional graphics extension, the shape extension, and others. These extensions can be compiled into the server or omitted if the extension is not expected ever to be used. This subdirectory also includes makefiles for building the extensions.
- lib** This subdirectory contains the code for building extension libraries which are used by clients to make new requests to the server.

Imakefile This is a datafile for a program named *imake* which is used to automatically build makefiles.

include This subdirectory contains header files which are used by the server and client routines which perform extension related tasks.

test This subdirectory contains test programs which are used to validate that an extension exists and to verify that the extension works correctly.

Writing an Extension

As discussed earlier, to add an extension to X it is necessary to make additions to both the server and client library routines. In order to explain these steps more clearly, an example of a data compression extension developed at Sandia National Laboratories to reduce the amount of pixel data sent over a network is used.

The primary goal was to develop a means of reducing the amount of pixel data used to convey an image to the server. The means to accomplish this task would be to perform some sort of data compression or other technique involving pixel reduction via loss. Because compression techniques will be incorporated into X11R6 and because they can often require a great deal of CPU power to be effective, it was decided to use a pixel loss algorithm. The algorithm that this extension uses is simple but produces relatively good results. It sends every other scan line of pixel data from the client and displays each line twice on the server. This reduces the amount of pixel data transmitted by 50%. This technique is useful only when sending images; it would not be usable for all data transfers between client and server. More information can be found in Appendix D.

The following sections will now examine the data structures, server routines, error handling, and client routines using both the generic extension examples and the compression extension.

Data Structures

The first step in adding an extension is to design the protocol which consists of messages passed between the client and server. The protocol has four basic elements: requests, replies, events, and errors. Clients send requests to the server, which interprets and acts on them. Some requests require the server to send replies back to the client. The server also sends events to clients to inform them of asynchronous conditions it has detected, such as mouse movement or keyboard presses. Clients usually receive only events in which they are interested. If a client makes a request to the server that is incorrect or not recognized, the server sends an error message back to the client.

The file `mit/include/Xproto.h` defines a generic C language structure for making a request [4]. Each request must begin with this four byte structure. The total length however must be a multiple of four bytes, and its total size is only limited by the maximum packet size the network can deliver. The structure is defined as follows.

```
typedef struct {  
    CARD8 reqType;  
    CARD8 data;  
    CARD16 length B16;  
} xReq;
```

This structure defines `xReq` to be a structure with 3 members. `CARD8` and `CARD16` are defined in `mit/include/Xmd.h` to be unsigned char and unsigned short types respectively or one byte and two bytes in length. The first member `reqType` contains the server assigned major opcode in the range of 128-255. Because each extension is allocated only one major opcode, the next member which is named `data` is typically used to hold the minor opcode if there is more than one request in the extension. The `length` member contains the length of the request in 4-byte quantities. The `B16` signifies that it is defined as a 16 bit bitfield.

The compression extension discussed in this report has three requests; one to query whether the extension exists on the server, one to set the type of compression or pixel reducing algorithm used, and one to actually send the reduced image to the server. The request structure used to send the reduced or compressed image data is now shown.

```
typedef struct _PutImageCompReq {
    CARD8      reqType;
    CARD8      CompressionReqType;
    CARD16     length B16;
    Drawable   drawable B32;
    GContext   gc B32;
    CARD16     width B16;
    CARD16     height B16;
    INT16      dstX B16;
    INT16      dstY B16;
    CARD8      leftPad;
    CARD8      depth;
    CARD8      format;
    CARD8      comptype;
} xPutImageCompReq;
#define sz_xPutImageCompReq 24
```

Note that the first three members are the same as the basic request structure except that the `data` member has been renamed `CompressionReqType`. This structure member holds the type of request which is the same as the minor opcode since this extension has more than one request. The `length` member contains the length of the request. It should be noted however that the length member for all requests contains the length in 4-byte quantities which includes the length of the basic request structure. The length member of the reply structure, which will be examined next, however, is the number of 4-byte quantities beyond the basic reply structure. The remaining structure members hold image data. The member `drawable` is the destination window of where the image will be drawn. The member `gc` is the graphics context being applied to graphics operations. The next two members, `width` and `height`, store the width and height of the image in pixels. Following are the `dstX` and `dstY` members which contain the x and y coordinates of where the image will be placed. The next member, `leftPad`, is the padding or number of

bytes to skip in each scan line. The *format* member tells the server what type of image is being sent, i.e. BITMAP, XYPIXMAP, or ZPIXMAP. The *comptype* member holds a value representing the chosen compression type since this extension is designed to be used with more than one compression type. The *#define* statement is not part of the structure but is used to define the length of the structure in bytes so the server can verify that the request it receives is the proper length.

On occasion a request from a client will require a reply from the server. If an extension is being added, it is a good practice to have the server at least reply that the extension exists. This reply must be at least 32 bytes long. The structure is defined in *mit/include/Xproto.h* as follows.

```
typedef struct {
    BYTE        type;
    BYTE        data1;
    CARD16      sequenceNumber B16;
    CARD32      length B32;
    CARD32      data00 B32;
    CARD32      data01 B32;
    CARD32      data02 B32;
    CARD32      data03 B32;
    CARD32      data04 B32;
    CARD32      data05 B32;
} xGenericReply;
```

The *type* member must state that this is an X_Reply structure. The *data1* member is not typically used. BYTE has been defined to be an unsigned char data type. The *sequenceNumber* member contains a server generated number which is incremented after each request. The *length* member is the length of the request beyond the size of xGenericReply. Thus, if a reply had the same size as this structure, its *length* member would contain 0. The rest of the structure can hold data which the client has requested from the server. It should be pointed out that an actual reply packet may be slightly different. The compression extension uses the structure shown at the top of the next page to reply to the xQueryCompression request.

As can be seen in the structure several of the 32 byte data members have been split into two 16 byte members and their type have been changed from CARD to INT since they will now be containing integers. The *data1* member has been renamed to *unused* and its type has been changed from BYTE to CARD8 which really is the same data type. The *data00* member of the XGenericReply has been split into the *majorVersion* and *minorVersion* members which hold the major and minor versions of the extension. The *data01* member has been changed to the *algorithm_mask* member which is used to define the compression types supported by the server. The *data02* member has been split into a *compression_type* member which contains the type of compression that the server

is currently performing if any, and an unused *pad3* member. The rest of the members, *pad4*, *pad5*, and *pad6* are unused. The `#define` statement defines the size of this request structure to be 32 bytes. It should be noted that this is still the basic reply structure except for the redefining of its members and its size remains the same.

```
typedef struct {
    BYTE      type;
    CARD8      unused;
    CARD16     sequenceNumber B16;
    CARD32     length B32;
    CARD16     majorVersion B16;
    CARD16     minorVersion B16;
    INT32      algorithm_mask B32;
    INT16      compression_type B16;
    CARD16     pad3 B16;
    CARD32     pad4 B32;
    CARD32     pad5 B32;
    CARD32     pad6 B32;
} xQueryCompressionReply;
#define sz_xQueryCompressionReply 32
```

If an extension defines new conditions about which clients can be notified, new event types for each of them must be defined. The structure of an event packet is defined by the `xEvent` structure in the file `mit/include/Xproto.h` to be 32 bytes in length and is shown below.

```
typedef struct {
    union {
        struct {
            BYTE type;
            BYTE detail;
            CARD16 sequenceNumber B16;
        } u;
        other event structures
    } u;
} xEvent;
```

The structure is somewhat different than the structures used for requests and replies. It contains a C language construct called a union which allows the same computer memory location to hold different types of data, in this case other event structures. As is shown above the `xEvent` structure is a union which can contain information about different events as denoted by the other event structures line which is used to save space rather than listing each event structure. The *type* member defines the event type. Extensions use event numbers ranging from 64-127 to signify the event type. The *detail* member

has no defined use. The *sequenceNumber* member holds the sequence number of the request the server has just processed on behalf of the client receiving the event. The remaining 28 bytes can be used to hold the information that should be sent to the client when the event occurs. If the information is larger than 28 bytes, then it is necessary to implement other means of delivering the information. One alternative is to add a request to the extension protocol that allows a client to query the needed information when it receives the new event. The compression extension does not define or need any new events [4].

Finally, the extension will need to generate error replies in response to packets that do not meet the protocol requirements. All errors, including those added by extensions, have the following format as defined in the file `mit/include/Xproto.h`, which is now shown.

```
typedef struct {
    BYTE      type;
    BYTE      errorCode;
    CARD16    sequenceNumber B16;
    CARD32    resourceID B32;
    CARD16    minorCode B16;
    CARD8     majorCode;
    BYTE      pad1;
    CARD32    pad3 B32;
    CARD32    pad4 B32;
    CARD32    pad5 B32;
    CARD32    pad6 B32;
    CARD32    pad7 B32;
} xError;
```

The length of the packet is 32 bytes and must have the value of `X_Error` for the *type* member. The *errorCode* member identifies the error type. For extension errors, this value is in the range 128-255. The *sequenceNumber* member contains the 16 least significant bits of the server generated sequence number. The *resourceID* member is often used to hold a resource identifier that was used erroneously in a request. This is beneficial if a resource sent by a client does not belong to the correct resource type for the requested operation. The *minorCode* member contains the minor opcode of the request. Minor opcodes are defined for a particular extension only. The rest of the server does not use this member. The *majorCode* member contains the major opcode of the failed request. Each extension is allocated only one major opcode. The other members of the structure are unused.

Server Routines

After all the data structures are defined in a header file, it is then necessary to write or modify the server routines which will perform the tasks of the extension. The first routine which needs to be modified is `mit/server/ddx/mi/miinitext.c`, which calls the extension initialization routines. The extension additions should be surrounded with a preprocessor `#ifdef` statement so the server can be compiled with or without the extension.

The additions for the extension described in this report are now listed.

```
#ifdef COMPRESSION
extern void XCompressionExtensionInit();
#endif

void InitExtensions(argc, argv)
    int argc;
    char *argv[];
{
    #ifdef COMPRESSION
        XCompressionExtensionInit();
    #endif
}
```

The first `#ifdef` statement declares the function `XCompressionExtensionInit` to be of type external void, meaning it is defined elsewhere and returns a void or null value after executing. This function also takes no arguments. These statements are only executed however if the constant `COMPRESSION` is defined. The second `#ifdef` statement actually calls the initialization function `XCompressionExtensionInit` if the constant `COMPRESSION` is defined.

The extension initialization routine or function is usually contained in the `mit/extensions/server` directory. This routine must also register the extension with the server by calling the `AddExtension` routine, which is defined by the `ddx` set of routines. This routine has the following syntax.

```
ExtensionEntry *AddExtension(name, numEvents, numErrors, mainProc,
                             swapped MainProc, closeDownProc, minorOpcodeProc)
    char *name;
    int numEvents;
    int numErrors;
    int (*mainProc());
    int (*swappedMainProc());
    void (*closeDownProc());
    unsigned short (*minorOpcodeProc());
```

This function, if successful, returns a pointer to an ExtensionEntry structure. The argument *name* defines the name of the extension. The arguments *numEvents* and *numErrors* define the number of new events and errors the extension can generate. The argument *mainProc* is a pointer to a function that handles the extensions requests, and *swappedMainProc* is a pointer to a function that processes requests in a byte-swapped format. The argument *closeDownProc* is a pointer to a function which shuts off the extension. The last argument *minorOpcodeProc* points to a routine that returns the minor opcode of a request.

The actual code used in the compression extension is as follows.

```
if (extEntry = AddExtension(COMPRESSIONNAME, NUMCOMPEVENTS,
                           NUMCOMPERERRORS, ProcCOMPRESSIONDispatch,
                           SProcCOMPRESSIONDispatch,
                           COMPRESSIONResetProc, StandardMinorOpcode))
```

The "if" statement checks to see if a valid pointer to extEntry was returned. The names of the compression extension routines and variables are then used as arguments.

To process client requests that have the extension major opcode, the server has a dispatch routine which will send the request to the correct routine to perform the necessary task. These dispatch routines work by examining the request buffer to get a request and examine the minor opcode which denotes the actual request made by the client. The minor opcode is used to index into a "switch" statement to select the proper routine for processing the request. These minor opcodes are usually defined in header files.

A portion of the routine used in the compression extension is shown below.

```
static int ProcCOMPRESSIONDispatch (client)
    register ClientPtr client;
{
    REQUEST (xReq);
    switch (stuff->data)
    {
        case X_QueryCompression:
            return ReplyCompression (client);
        case X_SetCompression:
            return SetVideoComp (client);
```

This routine gets a request by using the REQUEST macro statement with an xReq structure argument. The REQUEST macro also declares a variable of type xReq called *stuff*. The data member or minor opcode of this structure is then processed by the "switch" statement. In the above example, if *stuff->data* points to the minor opcode that has the value of X_QueryCompression, the *ReplyCompression* routine is called.

After the server has successfully executed a request from the client, it may be necessary for the server to send a reply back to the client. The routine to accomplish this is *WriteToClient* which is in the *os* set of routines. This routine has the following syntax.

```
int WriteToClient(pClient, replySize, replyBuffer)
    ClientPtr pClient;
    int replySize;
    char *replyBuffer;
```

The argument *pClient* is a pointer to a *ClientRec* structure which denotes the client that should receive the reply. The argument *replySize* contains the size of the reply in bytes. The argument *replyBuffer* is a pointer to the reply message. The compression extension uses this function as follows.

```
WriteToClient(client, sizeof(xQueryCompressionReply), (char *)&rep);
```

The only things different in this call are that the *replySize* is set by the call to *sizeof* which returns the size of the *XQueryCompressionReply* structure and *replyBuffer* is type cast to be a pointer to a char with the *(char *)* construct.

Extension events should only be sent to those clients that are designed for them. Therefore, the server must keep track of which clients are to receive the event. If events are added to the extension, a client must send a request notifying the server that the client would like to receive the event.

Events are sent to clients with a call to the *WriteEventsToClient* routine which is defined in the *dix* set of routines. The syntax of the routine is displayed below.

```
int WriteEventsToClient(pClient, numEvents, events)
    ClientPtr pClient;
    int numEvents;
    xEvent *events;
```

The argument *pClient* is a pointer to the *ClientRec* structure that defines the client to which the events will be sent. The argument *numEvents* contains the number of events to be sent. The argument *events* points to an array of *xEvent* structures which has an element for each event to be sent. The compression extension does not add any new events.

Error Handling

When a client has done something wrong, such as misuse the protocol, the extension must recognize this and send an error message back to the client. In many instances the core protocol defines a number of different error types which extensions can use when

various errors are detected. Some examples are a request of incorrect length or an argument that does not have the correct resource type or class. If the error is one that is not implemented by the core protocol, the extension will need to add one. When one of the extension dispatch routines is called, the `dix` set of routines expects a return value of Success. If it receives anything else as a return value, the main dispatch loop in `mit/server/ddx/dispatch.c` sends an error packet to the client.

As was described in the `xError` structure, five pieces of information are contained in the error. They are the major opcode of the failed request, the minor opcode, the error code, the sequence number of the failed request, and in some cases the resource identifier or value that was incorrect.

If the extension has any of its own errors defined, they must be defined sequentially from zero. To return one of these errors to the client, the sum of the `errorBase` member of the extension entry structure for the extension and the symbol for the error is sent. For example, in the compression extension the `BadCompParam` (a bad compression parameter) is defined to be 0 in the compression header file `compression.h` by the following statement.

```
#define BadCompParam 0
```

If a bad compression parameter is received by the server, the following error code is returned to the client.

```
return(BadCompParam + CompressionErrorBase)
```

Client Routines

Obviously, in a client-server environment changing the server implies that the client routines also must be changed in order for the client to request the new functionality from the server. There should be a query routine in the client that will interrogate the server to determine if the server supports the extension. It makes no sense to continue making extension requests if the server does not support the extension.

For many extensions it is usually necessary to enable the extension using client routines. This has the advantage of permitting the server to perform in a standard manner for clients which do not need or cannot use the extra functionality provided by the server extension. This routine may or may not have arguments to pass to the server depending upon whether or not the server extension has various options.

The following client routine is used in the compression extension to query the server in order to determine if the extension exists.

```
status = XQueryCompression(display, &event_basep, &alg_mask,
                           &comp_type, &comp_params)
```

The variable *status* will receive the value returned by this function, TRUE if the extension exists and FALSE otherwise. The argument *display* is the user's display or server. The arguments *event_basep* and *error_basep* are pointers to values assigned by the server to which event codes and error codes are added to respectively to produce an event code or error code. The argument *alg_mask* is a pointer to a bitfield used to hold the intersection of compression algorithms supported by the client and server. The argument *comp_type* is a pointer to the current compression algorithm. The last argument *comp_params* is a pointer to a structure used to hold parameters for the current compression algorithm.

After querying the extension, the client may be able to set certain options in the server. For example the compression extension uses the following function to set the desired type of compression the server will perform.

```
status = XSetCompression(display, comp_type, &comp_params)
```

The argument *display* once again is the display or server to be set, *comp_type* is an index to the desired compression algorithm, and *comp_params* is a pointer to a structure containing parameters for the desired compression algorithm.

The client will also have routines which will use the extension on the server to perform the new tasks. In the compression extension one such function is *XPutImageCompress*. An example of this client routine which compresses and transmits image data to the server is now shown shown.

```
PutImageCompress(dpy, d, gc, image, req_xoffset, req_yoffset, x, y,
                 req_width, req_height)
```

The argument *dpy* is the display or server. The argument *d* is the drawable or window. The argument *image* is a pointer to a structure containing the image data to be drawn or sent. The argument *req_xoffset* is the x coordinate offset in the image data. The *req_yoffset* argument is the y coordinate offset in the image data. The arguments *x* and *y* are the destination x and y coordinates relative to the origin of the drawable while the *req_width* and *req_height* arguments contain the width and height of the image in pixels [3].

Of course there may be other routines which the client may use to gain access to the server extension depending upon the extension. The examples described in this report were relevant to the compression extension and are general enough to be adapted to most new extensions.

Summary

This report details the steps needed to add an extension to the X Window System. After a brief introduction to X, it examines extensions and the necessary procedures in adding an extension. Appendix A contains instructions on obtaining the X source code and building X including extensions. Appendix B discusses compression and the compression extension. Appendix C contains annotated C language source code for the compression extension server. Finally, Appendix D contains annotated C language source code for the compression extension clients.

Acknowledgments

I would like to thank Doug Brown, 1932, for his help in this work. Doug had the original idea of extending X with compression. He also developed the framework for the client routines and coded a significant portion of them. Doug also acted as a reviewer of this report. I would also like to thank Mike Vahle, 1954, and Jerry Friesen, 1952, for reviewing this report.

Appendix A

Obtaining and Building the X Window System With Extensions

Before starting this procedure make sure you have plenty of disk space available. To build everything, you will need around 200 megabytes. To build a minimum working subset you will need around 100 megabytes. Then make a directory on that disk and change your working directory into the directory you have just created.

The quickest and cheapest way to obtain X is to use the *ftp* command to connect to the machine `export.lcs.mit.edu` (IP address of 198.112.44.100) for those who are not running a name server daemon and log in using the name of *anonymous*.

After logging on, change to the proper directory which contains the latest release of X which is currently Version 11 Release 5, using the following command.

```
cd pub/R5
```

To get everything, type the following command which turns off prompting.

```
prompt
```

Then send a copy of all the files back to your machine with the following command.

```
mget *
```

This will take a while (10 minutes or longer) depending upon the network traffic.

Then you need to unpack the distribution using UNIX commands *cat*, *uncompress*, and *tar* using the following command. Note that the filename `mit-1` should be replaced with `mit-2`, `mit-3`, and `mit-4` after using `mit-1`, and also with `contrib-1`, `contrib-2`, `contrib-3`, and `contrib-4` if the user contribution programs are desired.

```
cat mit-1 | uncompress | tar xfp -
```

Apply the bug-fixes. See the file `mit/RELNOTES.txt` and the file `mit/util/patch/README` for current instructions on how this is accomplished. Basically, all that is needed is to compile and run the *patch* program with the distribution supplied patches.

Edit the file `mit/config/site.def` and change the `#define` statement for `ProjectRoot` to the directory where you would eventually like to install the executables you are building. A safe way to do this is to set this variable to the top level directory that contains this

release. That way you do not overwrite existing programs in `/usr/bin`, `/usr/lib`, etc. For example if your X software is in the directory `/x` you could use the following statement.

```
#define ProjectRoot /x
```

If you are compiling the release with the `gcc` compiler, uncomment that line in `site.def` which has the following information.

```
#define HasGcc Yes
```

Other options can be set in `site.def` depending upon whether you desire the PEX three dimensional imaging extension built, fonts, etc.

After this initial configuration setup, X can be built by changing your working directory to the top level directory (the one with `RELNOTES.TXT` and `Makefile`) and typing the following command.

```
make World 2>&1 1>make.out
```

This compiles the whole X release and redirects all compiling messages and errors into a file named `make.out`. To watch the distribution compile, open another window on your workstation, change to the above directory and type the following command.

```
tail -f make.out
```

When your compilation is complete and if there are no error messages, your new server will be located in the directory `mit/server` and will have the name of `Xmachine` where machine is the type of machine on which you are compiling this release. For example for DEC machines it is `Xdec` and for Sun machines it is `Xsun`.

The next step is to run your new server for a few days to make sure it works correctly. Because server startup is operating system dependent, you will need to look at your machines startup files to see how this is accomplished. For DEC machines running Ultrix change the last line in the file `/etc/ttys` to the full pathname of your server and reboot, or go to single-user mode and then back to multi-user mode.

When you are satisfied that your server is working correctly and you need to add an extension, first check to make sure that it can not be accomplished by adding client programs, libraries, or resources. Also check with the X Consortium to make sure that someone else is not doing something similar. The address is listed at the end of this appendix.

If you still need to write the extension, then write your server, client, etc. routines as described in this report and place them in their respective directories under

mit/extensions. Edit the `Imakefiles` in those directories and add your routines. Use an existing extension such as the `mitmisc` extension as an example.

Edit `mit/server/ddx/mi/miinitext.c` and add your extension initialization routines as explained in this report, or you can also use an existing extension as an example.

Make sure you save a copy of your working server in case the new one has problems. You are now ready to recompile, but since you no longer need to build everything use the following command which only compiles your extension and other parts of the server code that needs to be recompiled.

make Everything 2>&1 1>make2.out

When it is done compiling, you will then need to test the extension using your test program and clients. When you get it to work correctly and would like to share it with the X Consortium (and also have them support it) contact:

Robert Scheifler
MIT X Consortium for Computer Science
545 Technology Square
Cambridge, MA 02139

Appendix B

Compression

The compression of transmitted data is nothing new. There have been many compression techniques developed over the years. Some techniques use specialized hardware, some use software, and some use a combination of hardware and software. Generally speaking compression done using hardware is more efficient than software compression.

X has not supported compression, but does use techniques that were discussed in this report to minimize the amount of data sent over the network. The problem arises when transmitting images composed of bitmaps or pixmaps which are changing rapidly. The client must send the complete picture to the server. This can be time and resource consuming especially in times of heavy network traffic. There are companies which make some compression packages using a compression technique called JPEG and although effective they are not always commercially available for all platforms running X. Software compression techniques take cpu time to compress the image on the client and also cpu time to uncompress the image on the server. In most cases, this extra effort does not produce a significant saving. The next release of X (X11R6) will incorporate some compression techniques into the X Image Extension but is not yet available.

A technique was needed which did not use specialized hardware, was portable to many X platforms, and did not require a lot of cpu power. The method used was accomplished strictly in software and is implemented by sending every other scan line in an image to the server which is a reduction in transmitted pixel data by 50%. The server draws each scan line twice, once at its original x and y coordinates and then again at x and y+1. Although this may seem simplistic, the results are quite acceptable when a small loss of resolution is tolerable. Figure 4 on page 28 contains a sample image of logos from The Adams Division of the National Hockey League that has not been altered by this algorithm. Figure 5 on page 29 shows the alternate scan line approach showing only a slight loss in picture quality. This example does not show how well this algorithm works with camera generated images such as people, etc. It was used to illustrate the worse case scenario.

It should be pointed out that this technique is only one of several techniques that can be implemented in this extension. Other techniques which this extension has been designed for but not implemented are JPEG, MPEG, The Phillips Laboratory Compression Algorithm, and a technique developed by Bob Mines at Sandia National Laboratories in California. The extension has been designed in such a way as to make other techniques easy to add if so desired.

Figure 4. The original image

Figure 5. The image after the alternate scan line algorithm was applied.

Appendix C

Server Routines

The following pages contain the source code for the routines listed below:

compression.c A collection of server routines to initialize the extension and perform client requests.

Compression.h A header file used by both server and client routines.

```

/*****
 *      compression.c  This program contains various routines for the      *
 *                      compresion extension. All routines are for the server. *
 *                                                                *
 *                      Joseph P. Brenkosh  1-21-93                    *
 *                      Organization 1954                               *
 *                      Sandia National Laboratories                   *
 *                      Albuquerque, New Mexico                       *
 *****/
#define NEED_EVENTS
#define NEED_REPLIES
#include <stdio.h>
#include "X.h"
#include "Xproto.h"
#include "misc.h"
#include "os.h"
#include "dixstruct.h"
#include "extnsionst.h"
#include "gcstruct.h"
#define _COMPRESSION_SERVER_
#include "Compression.h"
#include "Xproto.h"
#include "windowstr.h"
#include "gcstruct.h"
#include "osstruct.h"
#include "selection.h"
#include "colormapst.h"
#include "cursorstr.h"
#include "scrnintstr.h"
#include "opaque.h"
#include "input.h"
#include "servermd.h"
#include "extnsionst.h"

static unsigned char COMPRESSIONReqCode; /* jpb 1-22-93 */
static int ProcCOMPRESSIONDispatch(), SProcCOMPRESSIONDispatch();
static void COMPRESSIONResetProc();
static int compression_type;
static int CompressionErrorBase ;

```

```

/* This routine adds the extension entry to the server */
void
XCompressionExtensionInit() /* jpb 1-21-92 */
{
    ExtensionEntry *extEntry, *AddExtension();
    if (extEntry = AddExtension(COMPRESSIONNAME, NUMCOMPEVENTS,
                                NUMCOMPERRORS,
                                ProcCOMPRESSIONDispatch,
                                SProcCOMPRESSIONDispatch,
                                COMPRESSIONResetProc, StandardMinorOpcode))
    {
        COMPRESSIONReqCode = (unsigned char)extEntry->base;
        CompressionErrorBase = extEntry->errorBase;
        compression_type = XCompType_Default; /* Default Compression */
    }
}

/* This routine resets the extension. Currently this extension has nothing
   to reset so it does nothing */
static void
COMPRESSIONResetProc (extEntry) /* jpb 1-22-93 */
ExtensionEntry *extEntry;
{
}

/* This routine performs the "null" compression which takes each scan line
   and displays it twice. */
static int
uncomp_null (client) /* jpb 3-5-93 */
register ClientPtr client;
{
    register GC *pGC;
    register DrawablePtr pDraw;
    long length;
    int i;
    REQUEST(xPutImageCompReq); /* Get a request. store it in a stuff structure*/
#ifdef DEBUG /* Conditional compilation for debugging purposes */
    FILE *fp, *fopen();
#endif
/* The next two lines perform checks on the request information */
    REQUEST_AT_LEAST_SIZE(xPutImageCompReq);
    VALIDATE_DRAWABLE_AND_GC(stuff->drawable, pDraw, pGC, client);
#ifdef DEBUG
    fp = fopen("/tmp/DEBUG", "A");
    fprintf(fp, "Format=%s\n", stuff->format);
    fflush(fp);

```

```

fprintf(fp, "Depth=%s\n", stuff->depth);
flush(fp);
fprintf(fp, "width=%s\n", stuff->width);
flush(fp);
fprintf(fp, "dstX=%s\n", stuff->dstX);
flush(fp);
fprintf(fp, "dstY=%s\n", stuff->dstY);
flush(fp);
fclose(fp);
#endif

/* Perform some checks on XYBitmaps and XYPixmaps, this is only a stub because
currently only ZPixmaps are supported */
if (stuff->format == XYBitmap)
{
    if ((stuff->depth != 1) ||
        (stuff->leftPad >= screenInfo.bitmapScanlinePad))
        return BadMatch;
    length = PixmapBytePad(stuff->width + stuff->leftPad, 1);
}
else if (stuff->format == XYPixmap)
{
    if ((pDraw->depth != stuff->depth) ||
        (stuff->leftPad >= screenInfo.bitmapScanlinePad))
        return BadMatch;
    length = PixmapBytePad(stuff->width + stuff->leftPad, 1);
    length *= stuff->depth;
}
else if (stuff->format == ZPixmap)
{
    if ((pDraw->depth != stuff->depth) || (stuff->leftPad != 0))
        return BadMatch;
    length = PixmapBytePad(stuff->width, stuff->depth);
}
else
{
    client->errorValue = stuff->format;
    return BadValue;
}
length *= stuff->height;
if (((length + 3) >> 2) + (sizeof(xPutImageCompReq) >> 2)) != stuff->length)
    return BadLength;

```

```

/* Call the original PutImage to draw a scan line, then increment i which is
   added to the existing y coordinate which then becomes the start of the next
   scan line and repeat the existing scan line */
for(i=0; i<2; i++)
{
    (*pGC->ops->PutImage) (pDraw, pGC, stuff->depth, stuff->dstX,i+stuff->dstY,
                          stuff->width, stuff->height,
                          stuff->leftPad, stuff->format,
                          (char *) &stuff[1]);
}
    return (client->noClientException);
}

/* The following is a stub to add jpeg compression */
static int
uncomp_jpeg (client,req) /* jpb 3-5-93 */
    register ClientPtr client;
    xPutImageCompReq req;
{
}

/* The following is a stub to add mpeg compression */
static int
uncomp_mpeg (client,req) /* jpb 3-5-93 */
    register ClientPtr client;
    xPutImageCompReq req;
{
}

/* The following is a stub to add Phillips Lab compression */
static int
uncomp_plab (client,req) /* jpb 3-5-93 */
    register ClientPtr client;
    xPutImageCompReq req;
{
}

/* The following is a stub to add Bob Mines compression */
static int
uncomp_mines (client,req) /* jpb 3-5-93 */
    register ClientPtr client;
    xPutImageCompReq req;
{
}

```

```

/* The following routine allows a client to set the type of compression */
static int
SetVideoComp (client) /* jpb 3-4-93 */
    register ClientPtr client;
{
    REQUEST(xSetCompressionReq);
    xSetCompressionReq req;
    register int n;

    unsigned int mask = 0;
    REQUEST_SIZE_MATCH(xSetCompressionReq); /* Get a request */
    switch ( stuff->compression_type ) /* Look at the type of compression */
    {
        case XCompType_Null: /* Null Compression */
            compression_type=stuff->compression_type ;
            return Success;
        case XCompType_JPEG: /* JPEG Compression */
            compression_type=stuff->compression_type ;
            return Success;
        case XCompType_MPEG: /* MPEG Compression */
            compression_type=stuff->compression_type ;
            return Success;
        case XCompType_PLab: /* Phillips Lab Compression */
            compression_type=stuff->compression_type ;
            return Success;
        case XCompType_Mines: /* Bob Mines Compression */
            compression_type=stuff->compression_type ;
            return Success;
        default: /* Invalid compression requested */
            return(BadCompParam + CompressionErrorBase);
    }
}

/* This routine is called by the PROCompressionDispatch routine to call
the correct uncompression routine */
static int
ProcPutImageCompress (client) /* jpb 3-4-93 */
    register ClientPtr client;
{
    switch ( compression_type )
    {
        case XCompType_Null:
            return uncomp_null(client);
    }
}

```



```

        break;
    case XCompType_JPEG:
        return uncomp_jpeg(client);
        break;
    case XCompType_MPEG:
        return uncomp_mpeg(client);
        break;
    case XCompType_PLab:
        return uncomp_plab(client);
        break;
    case XCompType_Mines:
        return uncomp_mines(client);
        break;
    default:
        return(BadCompression+CompressionErrorBase);
    }
}

/* This routine returns information to the client about compression params */
static int
ReplyCompression(client)
    register ClientPtr client;
{
    REQUEST(xQueryCompressionReq);
    xQueryCompressionReply rep;
    register int n;

    unsigned int mask = 0;
    REQUEST_SIZE_MATCH(xQueryCompressionReq);
    rep.type = X_Reply; /* Load the rep structure with information */
    rep.length = 0;
    rep.sequenceNumber = client->sequence;
    rep.majorVersion = COMPRESSION_MAJOR_VERSION;
    rep.minorVersion = COMPRESSION_MINOR_VERSION;
    rep.algorithm_mask = mask | COMPMASK; /* Set the last 5 bits to */
                                           /* say we can do everything */
    rep.compression_type = compression_type; /* Default Compression */
    if (client->swapped) { /* If client has LSB, MSB reversed from us */
        swaps(&rep.sequenceNumber, n);
        swapl(&rep.length, n);
        swaps(&rep.majorVersion, n);
        swaps(&rep.minorVersion, n);
        swaps(&rep.algorithm_mask, n);
        swaps(&rep.compression_type, n);
    }
}

```

```

/* Send the info to the client */
WriteToClient(client, sizeof(xQueryCompressionReply), (char *)&rep);
return (client->noClientException);

}/* This routine examines requests and calls the correct compression routine */
static int
ProcCOMPRESSIONDispatch (client)
    register ClientPtr      client;
{
    REQUEST(xReq);
    switch (stuff->data)
    {
        case X_QueryCompression: /* The client checking compression params */
            return ReplyCompression (client);
        case X_SetCompression: /* The client is setting compression params */
            return SetVideoComp (client);
        case X_PutImageComp: /* The client is sending a compressed image */
            return ProcPutImageCompress (client);
        default: /* The client made a mistake */
            return BadRequest;
    }
}

/* The following routine is a stub for clients with architectures which
   have MSB, LSB reversed from us */
static int
SProcCOMPRESSIONDispatch (client)
    register ClientPtr      client;
{
    REQUEST(xReq);
    switch (stuff->data)
    {
        case X_QueryCompression:
            return ProcCOMPRESSIONDispatch (client);
        case X_SetCompression:
            return ProcCOMPRESSIONDispatch (client);
        default:
            return BadRequest;
    }
}

```

```

/*****
 *      Compression.h  A header file for the Sandia Extension to support data *
 *                    compression in X Windows.                             *
 *                    Both server and client use this file.                  *
 *                                                                           *
 *                    C. Douglas Brown 1-21-93                             *
 *                    Organization 1954                                     *
 *                    Sandia National Laboratories                         *
 *                    Albuquerque, New Mexico                             *
 *                    Modified by Joseph P. Brenkosh                       *
 *****/
#endif _COMPRESSION_H_
#define _COMPRESSION_H_

#include <X11/Xfuncproto.h> /* jpb 3-1-93 */
/* Define constants for this extension */

#define COMPRESSIONNAME "Video-Compression"
#define COMPRESSION_MAJOR_VERSION 1 /* current version
numbers */
#define COMPRESSION_MINOR_VERSION 0

#define NUMCOMPEVENTS 0 /* Number of events */
#define NUMCOMPERRORS 2 /* Number of errors */
#define COMPMASK 037 /* Default compression mask */

/*
 * Define new error codes for video compression
 */
#define BadCompParam 0 /* Compression parameter invalid or missing */
#define BadCompression 1 /* Image could not be uncompressed */

/*
 * Define Video Compression Requests.
 */
#define X_QueryCompression 0 /* Query video compression */
#define X_SetCompression 1 /* Set video compression parameters */
#define X_PutImageComp 2 /* Put compressed image */

```

```

/*
 * Define Video Compression Types supported.
 */
#define XCompType_Null      0      /* Null compression algorithm */
#define XCompType_JPEG      1      /* JPEG standard */
#define XCompType_MPEG      2      /* MPEG standard */
#define XCompType_PLab      3      /* Air Force Phillips Lab algorithm */
#define XCompType_Mines      4      /* Bob Mines pixel averaging alg. */
#define XCompType_Default    XCompType_Mines

/*
 * Define constants for compression algorithms.
 */
#define MAXDCTSIZE    256      /* maximum space required for Discrete */
                                /* Cosine Transforms when encoded */
#ifndef _COMPRESSION_SERVER_

/* Define structure for video compression parameters */

typedef struct {
    int dummy;
} XCompress_null;

typedef struct {
    int Q_factor;
    int huffman_tab_size; /* huffman table size */
    int huffman_tab [256];
} XCompress_JPEG;

typedef struct {
    int Q_factor;
    int huffman_tab_size; /* huffman table size */
    int huffman_tab [256];
} XCompress_MPEG;

typedef struct {
    int subtype;          /* 8-bit, 24-bit, or pallette compression */
} XCompress_PLab;

typedef struct {
    int rect_x;           /* x-size of rectangles to compress */
    int rect_y;           /* y-size of rectangles to compress */
} XCompress_Mines;

```

```

typedef struct _XCompressionParams {
    int comp_type;          /* index to video compression algorithm */
    union {
        XCompress_null dummy;      /* null algorithm */
        XCompress_JPEG JPEG;        /* JPEG standard */
        XCompress_MPEG MPEG;        /* MPEG standard */
        XCompress_PLab PLab;        /* Air Force Phillips Lab algorithm */
        XCompress_Mines Mines;      /* Bob Mines pixel averaging alg. */
    } params;
} XCompressionParams;
/*
 * Define structure for compressed video image
 */
typedef struct _XImageComp {
    int width, height;          /* size of image */
    int xoffset;                /* number of pixels offset in x direction */
    int format;                 /* XYBitmap, XYPixmap, ZPixmap */
    char *data;                 /* pointer to compressed image data */
    int byte_order;             /* data byte order, LSBFirst, MSBFirst */
    int bitmap_unit;            /* quant. of scan line 8, 16, 32 */
    int bitmap_bit_order;       /* LSBFirst, MSBFirst */
    int depth;                  /* depth of image */
    int bytes_per_line;         /* accelerator to next line */
    int bits_per_pixel;         /* bits per pixel (ZPixmap) */
    XCompressionParams *params; /* pointer to structure containing video compression
                                parameters */
} XImageComp;

/* This section defines function prototypes */
_XFUNCPROTOBEGIN

extern XCompressImage (
    #if NeedFunctionPrototypes
        XImage*          /* image */,
        XImageComp*      /* compressed image */
    #endif
);

```

```

extern XPutImageCompress (
#ifdef NeedFunctionPrototypes
    Display*      /* display */,
    Drawable      /* d */,
    GC            /* gc */,
    XImage*       /* image */,
    int           /* src_x */,
    int           /* src_y */,
    int           /* dest_x */,
    int           /* dest_y */,
    unsigned int  /* width */,
    unsigned int  /* height */
#endif
);
extern XQueryCompression (
#ifdef NeedFunctionPrototypes
    Display*      /* display */,
    int *event_basep /* event base pointer */,
    int *error_basep /* error base pointer */,
    long int*      /* algorithm mask */,
    int*           /* compression type */,
    XCompressionParams* /* video compression parameters for selected type */
#endif
);

extern XSetCompression (
#ifdef NeedFunctionPrototypes
    Display*      /* display */,
    int           /* compression type */,
    XCompressionParams* /* video compression parameters for selected type */
#endif
);

_XFUNCPROTOEND

#endif

```

```

/*
 * This section defines the new request and reply packets
 */
typedef struct _PutImageCompReq { /* Request for drawing a compressed image */
    CARD8      reqType;
    CARD8      CompressionReqType;
    CARD16     length B16;
    Drawable   drawable B32;
    GContext   gc B32;
    CARD16     width B16;
    CARD16     height B16;
    INT16      dstX B16;
    INT16      dstY B16;
    CARD8      leftPad;
    CARD8      depth;
    CARD8      format;
    CARD8      comptype;
} xPutImageCompReq;
#define sz_xPutImageCompReq 24 /* Size in bytes of this request */

/* This request packet is used to query the server for compression */
typedef struct {
    CARD8      reqType; /* always codes->major_opcode */
    CARD8      CompressionReqType; /* always X_QueryVideoComp */
    CARD16     length B16; /* length in 4-byte quantities of request, */
                                /* including this header */
} xQueryCompressionReq;
#define sz_xQueryCompressionReq 4

/* This is a reply packet sent from server to client */
typedef struct {
    BYTE       type; /* X_Reply */
    CARD8      unused;
    CARD16     sequenceNumber B16;
    CARD32     length B32; /* 4-byte quantities beyond size of GenericReply */
    CARD16     majorVersion B16;
    CARD16     minorVersion B16;
    INT32      algorithm_mask B32;
    INT16      compression_type B16;
    CARD16     pad3 B16;
    CARD32     pad5 B32;
    CARD32     pad6 B32;
    CARD32     pad7 B32;
} xQueryCompressionReply;
#define sz_xQueryCompressionReply 32

```

```

/* This is a request to set the compression type on the server */
typedef struct {
    CARD8      reqType;          /* always codes->major_opcode */
    CARD8      CompressionReqType;
    CARD16     length B16;
    INT16      compression_type B16;
    CARD16     pad3 B16;
    CARD32     pad4 B32;
    CARD32     pad5 B32;
    CARD32     pad6 B32;
    CARD32     pad7 B32;
    /* XCompressionParams params B32; Compression parameters follow pads, will add
    later */
} xSetCompressionReq;
#define sz_xSetCompressionReq 24

#endif /* _COMPRESSION_H_ */

```


Appendix D

Client Routines

The following pages contain the source code for the routines listed below:

XCompression.c A collection of client routines used to query and set compression.

XPutImageCompress.c A collection of routines used to compress image data and send it to the server.

```

/*****
 *      Xcompression.c  This program is a library of client routines for the
 *                      compression extension.
 *
 *                      C. Douglas Brown 1-15-93
 *                      Organization 1954
 *                      Sandia National Laboratories
 *                      Albuquerque, New Mexico
 *****/

#define NEED_REPLIES
#include <X11/Xlibint.h>
#include "Compression.h"
#include <Xext.h>
#include <extutil.h>

static XExtensionInfo _compression_info_data;
static XExtensionInfo *compression_info = &_compression_info_data;
static char *compression_extension_name = COMPRESSIONNAME;

int compress_major_opcode;
long int client_algorithm_mask;
int compression_type;
int jpeg_fd;      /* file descriptor for JPEG hardware device */
XCompressionParams compression_params;

#define CompressionCheckExtension(dpy,i,val) \
  XextCheckExtension (dpy, i, compression_extension_name, val)

/*****
 *
 *                      private utility routines
 *
 *****/

```

```

static int close_display();
static /* const */ XExtensionHooks compression_extension_hooks = {
    NULL,      /* create_gc */
    NULL,      /* copy_gc */
    NULL,      /* flush_gc */
    NULL,      /* free_gc */
    NULL,      /* create_font */
    NULL,      /* free_font */
    close_display, /* close_display */
    NULL,      /* wire_to_event */
    NULL,      /* event_to_wire */
    NULL,      /* error */
    NULL,      /* error_string */
};

static XEXT_GENERATE_FIND_DISPLAY (find_display, compression_info,
                                   compression_extension_name,
                                   &compression_extension_hooks,
                                   CompressionNumberEvents, NULL)

static XEXT_GENERATE_CLOSE_DISPLAY (close_display, compression_info)

/*
 * XQueryCompression - determine video compression capability.
 *
 * Returns          True, if X server supports video compression.
 *                  False, if not.
 */
Bool XQueryCompression (dpy, event_basep, error_basep, alg_mask,
                        alg_index, alg_params)
    Display *dpy;
    int *event_basep, *error_basep;
    long int *alg_mask;
    int *alg_index;
    XCompressionParams *alg_params;

/* returns bit mask of compression
   algorithms */
/* supported by both X client and server. */
/* returns index of current video
   compression */
/* algorithm (default is Null algorithm) */
/* pointer to structure to receive params */
/* for current video compression algorithm
 */
{
    int nbytes;
    xQueryCompressionReq *req;
    xQueryCompressionReply rep;

```

```

XCompressionParams server_compression_params;

XExtDisplayInfo *info = find_display (dpy);

/* See if the server supports video compression */

if (XextHasExtension(info)) {
    compress_major_opcode = info->codes->major_opcode;
    *event_basep = info->codes->first_event;
    *error_basep = info->codes->first_error;
}
else {
    return False;
}

/* First find compression types supported by the client */

*alg_index = XCompType_Default; /* default to null compress. algorithm */
client_algorithm_mask = 1 << XCompType_Null |
    1 << XCompType_Mines;
if ((jpeg_fd = open("/dev/vfc0",2)) != -1) {
    /* Add JPEG to our list of client-supported algorithms */
    client_algorithm_mask |= 1 << XCompType_Null;
    close(jpeg_fd);
}

/* Get the server's algorithm mask, as well as its current compression
   type and parameters. */

GetReq(QueryCompression, req);
req->reqType = info->codes->major_opcode;
req->CompressionReqType = X_QueryCompression;
LockDisplay(dpy);

if (_XReply (dpy, (xReply *) &rep, 0, xFalse) == 0) {
    /* Error getting compression params from server */
    UnlockDisplay(dpy);
    SyncHandle();
    return False;
}

nbytes = (long)rep.length << 2;
if (nbytes > sizeof(struct _XCompressionParams)) {
    /* Error - server returning more bytes than the expected reply */
    _XEatData(dpy, (unsigned long) nbytes);
}

```

```

    UnlockDisplay(dpy);
    SyncHandle();
    return False;
}

if (nbytes > 0) {
    _XReadPad (dpy, server_compression_params, nbytes);
    bcopy(alg_params, server_compression_params,
          sizeof (struct _XCompressionParams));
    bcopy(compression_params, server_compression_params,
          sizeof (struct _XCompressionParams));
}

/* Find intersection of compression types supported by client & server
 * If server supports compression type reported by server, and
 * the intersection is non-null, okay */

*alg_mask = client_algorithm_mask & rep.algorithm_mask;

if (*alg_mask != 0) {
    *alg_index = compression_type = rep.compression_type;
    UnlockDisplay(dpy);
    SyncHandle();
    return True;
} else {
    /* server doesn't support its own current compression type */
    UnlockDisplay(dpy);
    SyncHandle();
    return False;
}
}

```

```

/*
 * XSetCompression - set video compression algorithm and parameters.
 *
 * Returns      True, if requested parameters can be set.
 *              BadValue, if an invalid algorithm index is specified.
 *              BadCompParam, if a compression parameter is invalid or missing.
 */
int XSetCompression (dpy, alg_index, alg_params)
    Display *dpy;
    int alg_index;          /* index of desired video compress. algorithm */
    XCompressionParams *alg_params; /* pointer to structure containing params */
                                /* for the desired video compression */
                                /* algorithm */
{
    xSetCompressionReq *req;
    xGenericReply rep;

    XExtDisplayInfo *info = find_display (dpy);
    compress_major_opcode = info->codes->major_opcode;

    if (client_algorithm_mask & (1 << alg_index) == 0)
        return False; /* client doesn't support this compression algorithm */

    /* Get the server's algorithm mask, as well as its current compression
       type and parameters. */

    GetReq(SetCompression, req);
    req->reqType = info->codes->major_opcode;
    req->CompressionReqType = X_SetCompression;
    req->compression_type = alg_index;
    LockDisplay(dpy);
    /* server accepted our compression parameters, so save them */
    compression_type = alg_index;
    UnlockDisplay(dpy);
    SyncHandle();
    return True;
}

/*
 * XCompressImage - compress video image using current compression parameters.
 *
 * Returns      True, if requested parameters can be set.
 *              BadValue, if an invalid algorithm index is specified.
 *              BadCompParam, if a compression parameter is invalid or missing.
 */

```

```

int XCompressImage (image, image_comp, comp_params)
    XImage          *image;
    XImageComp      *image_comp;
    XCompressionParams *comp_params;
{
    if ((image->format != ZPixmap) || (image->bits_per_pixel % 8 != 0))
        return BadCompression; /* Can't compress the image */
    switch (comp_params->comp_type) {
    case XCompType_Null:
        image_comp->data = image->data;
        break;
    default:
        return BadCompression; /* Only support null algorithm initially */
    }
    return True;
}

```

```

/*****
 *      XPutImageCompress.c  This routine compresses images and sends
 *                          them to the server to be displayed.
 *
 *                          C. Douglas Brown 2-15-93
 *                          Joseph P. Brenkosh
 *                          Organization 1954
 *                          Sandia National Laboratories
 *                          Albuquerque, New Mexico
 *****/

```

```

#define NEED_EVENTS
#define NEED_REPLIES
#include <stdio.h>
#include <X11/Xlibint.h>
#include "Compression.h"
#include <Xext.h>
#include <extutil.h>
#include <stdio.h>
/* assumes pad is a power of 2 */
#define ROUNDUP(nbytes, pad) (((nbytes) + ((pad) - 1)) & ~(long)((pad) - 1))

extern int compress_major_opcode;
extern long int client_algorithm_mask;
extern int compression_type;
extern int jpeg_fd; /* file descriptor for JPEG hardware device */
extern XCompressionParams compression_params;

static void
PutImageCompNull (dpy, d, gc, image)
    register Display *dpy;
    Drawable d;
    GC gc;
    register XImage *image;
{
#ifdef DEBUG /* Used for debuffing purposes */
    FILE *fp;
#endif
    register xPutImageCompReq *req;
    int Available;
    char *data;
    char *jdata;
    int x,y;
    int i, n;

```



```

long bytes_per_src, bytes_per_dest, length;
int inc = 1;
x = 0;
y = 0;
data = image->data;
jdata = image->data;
#ifdef DEBUG
fp=fopen("/tmp/xdata", "w");
if(fp == NULL)
{
    fprintf(stderr, "Error opening file\n");
    exit(-1);
}
/* Print out image information */
fprintf(fp, "bits Per pixel=%d\n", image->bits_per_pixel);
fprintf(fp, "format=%d\n", image->format);
fprintf(fp, "bitmap_unit=%d\n", image->bitmap_unit);
fprintf(fp, "bitmap_pad=%d\n", image->bitmap_pad);
fprintf(fp, "bitmap_depth=%d\n", image->depth);
fprintf(fp, "bytes_per_line=%d\n", image->bytes_per_line);
fprintf(fp, "red bits Per pixel=%u\n", image->red_mask);
fprintf(fp, "green bits Per pixel=%u\n", image->green_mask);
fprintf(fp, "blue bits Per pixel=%u\n", image->blue_mask);
fclose(fp);
#endif

/* This section sends every other scan line to the server */
for (y=0; y< image->height; y=y+(2*inc))
{
    GetReq(PutImageComp, req); /* Get a request structure */
    req->reqType = compress_major_opcode; /* Load it with request info */
    req->CompressionReqType = X_PutImageComp;
    req->drawable = d;
    req->gc = gc->gid;
    req->dstX = x;
    req->dstY = y;
    req->width = image->width;
    req->height = inc;
    req->depth = image->depth;
    req->format = ZPixmap;
    req->comptype = XCompType_Null;
    req->leftPad = 0;
    bytes_per_src = ROUNDUP((long)req->width * image->bits_per_pixel, 8)>>3;
    bytes_per_dest = bytes_per_src;
    length = bytes_per_dest * req->height;
}

```

```

    req->length += (length + 3) >> 2;

/* send the image to the server */
    Data(dpy, data, inc*image->bytes_per_line);

/* Skip the next line and repeat the above operation*/
    data += 2*inc*image->bytes_per_line;
}
}

/* This routine is a stub which will be used to compress images via jpeg */
static void
PutImageCompJPEG (dpy, d, gc, image)
    register Display *dpy;
    Drawable d;
    GC gc;
    register XImage *image;
{
    register xPutImageCompReq *req;
    int left_pad, Available;
    int x,y,length;

    Available = ((65536 < dpy->max_request_size) ? (65536 << 2)
                : (dpy->max_request_size << 2)) - SIZEOF(xPutImageReq);

    GetReq(PutImageComp, req);
    req->reqType = compress_major_opcode;
    req->CompressionReqType = X_PutImageComp;
    req->drawable = d;
    req->gc = gc->gid;
    req->dstX = 0;
    req->dstY = 0;

    for (y=0; y<image->height; y+=8) {
        for (x=0; x<image->width; x+=8) {
            if (MAXDCTSIZE < Available) {

                /* Compute Discrete Cosine Transforms here and
                 * stuff the Huffman encoded values directly into
                 * the display buffer, so we won't need a buffer copy
                 */
                req->length += MAXDCTSIZE;
                dpy->bufptr += MAXDCTSIZE;
                left_pad = 0;
            }

```

```

        else {
            _XFlush(dpy);
            GetReq(PutImageComp, req);
            req->reqType = compress_major_opcode;
            req->CompressionReqType = X_PutImageComp;
            req->drawable = d;
            req->gc = gc->gid;
            req->dstX = x;
            req->dstY = y;
        }
    }
    _XFlush(dpy); /* for debugging -- may remove this later */
}

/* This routine performs error checking and calls the correct compression routine */
XPutImageCompress (dpy, d, gc, image, req_xoffset, req_yoffset, x, y, req_width,
                  req_height)
    register Display *dpy;
    Drawable d;
    GC gc;
    register XImage *image;
    int x, y;
    unsigned int req_width, req_height;
    int req_xoffset, req_yoffset;
{
    if ((image->format != ZPixmap) || (image->bits_per_pixel % 8 != 0) ||
        (req_xoffset != 0) || (req_yoffset != 0) ||
        (req_width != image->width) || (req_height != image->height))
        return BadCompression; /* Can't compress the image */

    /* Later we want to automatically invoke PutImage, if we can't compress.

        XPutImage(dpy, d, gc, image, req_xoffset, req_yoffset, x, y, req_width,
                  req_height);
        return;
    */
    LockDisplay(dpy);
    FlushGC(dpy, gc);
    switch (compression_type) {

    case XCompType_Null:
        PutImageCompNull(dpy, d, gc, image); /* Perform null compression */
        break;

```

```
case XCompType_JPEG:
    PutImageCompJPEG(dpy, d, gc, image); /* Perform jpeg compression */

default:
    return BadCompression;
}
UnlockDisplay(dpy);
SyncHandle();
return True;
}
```

Bibliography

- [1] Adrian Nye. *Volume Zero: X Protocol Reference Manual*. O'Reilly & Associates, Inc. 1990. ISBN 0-937175-50-1.
- [2] Adrian Nye. *Volume One: Xlib Programming Manual*. O'Reilly & Associates, Inc. 1988. ISBN 0-937175-27-7.
- [3] Adrian Nye. *Volume Two: Xlib Reference Manual*. O'Reilly & Associates, Inc. 1988. ISBN 0-937175-28-5.
- [4] Elias Israel and Erik Fortune. *The X Window System Server*. Digital Press. 1992. ISBN 1-55558-096-3.
- [5] Robert W. Scheifler, James Gettys, and Ron Newman. *X Window System C Library and Protocol Reference*. Digital Press. 1988. ISBN 1-55558-012-2.

Distribution:

MS 0320 P. L. Mattern, 1010
MS 0320 C. E. Meyers, 1011
MS 0321 E. H. Barsis, 1400
MS 0360 A. R. C. Westwood, 1000
MS 0619 Technical Publications, 7151
MS 0805 A. R. Iacoletti, 1904
MS 0805 C. D. Brown, 1932
MS 0805 R. W. Simons, 1932
MS 0806 D. C. Jones, 1903
MS 0803 R. M. Cahoon, 1902
MS 0806 W. D. Swartz, 1957
MS 0806 T. V. Archuleta, 1957
MS 0806 G. W. Bollig, 1957
MS 0806 D. P. Duggan, 1957
MS 0806 M. J. Hannah, 1957
MS 0806 R. A. Haynes, 1957
MS 0806 S. M. Kelly, 1957
MS 0806 J. P. Long, 1957
MS 0807 M. O. Vahle, 1954
MS 0807 A. R. Breckenridge, 1954
MS 0807 J. P. Brenkosh, 1954 (20)
MS 0807 S. A. Gossage, 1954
MS 0807 T. C. Hu, 1954
MS 0807 J. H. Maestas, 1954
MS 0807 L. G. Martinez, 1954
MS 0807 J. H. Nagle, 1954
MS 0807 L. G. Pierson, 1954
MS 0807 T. J. Pratt, 1954
MS 0807 N. Testi, 1954
MS 0807 B. C. Whittet, 1954
MS 0807 R. C. Dougherty, 1955 (Actg.)
MS 0807 J. P. Sena, 1955-1
MS 0809 E. Baldonado, 10327
MS 0811 R. J. Pryor, 1956
MS 0811 P. L. Campbell, 1956
MS 0811 S. P. Goudy, 1956
MS 0811 G. D. Machin, 1956
MS 0811 M. M. Myerly, 1956
MS 0899 Technical Library, 7141 (5)
MS 1035 M. J. Murphy, 10300
MS 1109 A. L. Hale, 1424
MS 1119 Document Processing for DOE/OSTI, 7613 (10)
MS 1375 L. D. Bertholf, 4400
MS 9003 D. L. Crawford, 1900

MS 9011 F. T. Bilecki, 1951
MS 9011 H. Y. Chen, 1951
MS 9011 J. C. Berry, 1951
MS 9011 C. L. Ford, 1951
MS 9011 J. A. Hutchins, 1951
MS 9011 P. W. Dean, 1951
MS 9011 R. D. Gay, 1952
MS 9011 R. F. Mines, 1952
MS 9011 R. E. Cline, Jr., 1952
MS 9011 J. A. Friesen, 1952
MS 9011 R. E. Palmer, 1901
MS 9012 D. H. Ching, 1951-1
MS 9017 Central Technical Files, 8523
MS 9040 G. Gutierrez, 1905

DATE

FILMED

3 / 1 / 94

END