A Low Power, Low Bandwidth Protocol for Remote Wireless Terminals

by

George Ioannou Hadjiyiannis

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 1995

Author .....
Department of Electrical Engineering and Computer Science
August 1, 1995

Certified by .....
Srinivas Devadas
Associate Professor
Thesis Supervisor

Certified by .....
Anantha P. Chandrakasan
Assistant Professor
Thesis Supervisor

Accepted by .....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

A Low Power, Low Bandwidth Protocol for Remote Wireless Terminals

by

George Ioannou Hadjiyiannis

Submitted to the Department of Electrical Engineering and Computer Science
on August 1, 1995, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

A protocol for a low power low bandwidth wireless remote terminal is described in this thesis. With computing becoming so widespread and portable computing devices becoming correspondingly more significant, low power is a major design criterion. One way of achieving very low power consumption is to perform all tasks, other than managing hardware for the display and input, on a stationary workstation and exchange information between that workstation and the portable terminal via a wireless link. A protocol for such a system that emphasizes low bandwidth and low power requirements is presented herein. We describe error correction and retransmission methods capable of dealing with burst error noise up to BERs of 10^{-3} as well as suggestions for improving the protocol and recommendations for the terminal hardware. The final average bandwidth required is 140Kbits/sec for 8-bit color applications.

Thesis Supervisor: Srinivas Devadas
Title: Associate Professor

Thesis Supervisor: Anantha P. Chandrakasan
Title: Assistant Professor

Acknowledgments

I would like to thank a number of people whose help and contributions made this work possible. Dr. Robert Armstrong provided endless debugging tips and information on the low-level system software that had to be modified to make the emulator environment work. Richard Han of U.C. Berkeley provided the initial guidance that allowed me to localize the necessary modifications to the X server software. Erik Fortune, through his book and personal e-mails helped me understand the X server internals enough to modify it successfully. Mike Ehrlich provided help and recommendations for error correction. He also helped correct many of the mistakes I made in creating this document. Paul Flaherty created the ECC code that forms the basis of the error correction scheme used here. Last but not least, I would like to thank my supervisors, Prof. Srinivas Devadas for providing a direction to the project, suggestions and feedback, resources and facilities, and Prof. Anantha P. Chandrakasan whose experience and background with the Infopad project and his feedback, made my work a lot easier. Both of them also spent a lot of time reading and correcting the original manuscripts and making recommendations to improve it.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent years have seen a dramatic increase in the demand for computational resources. As users become more comfortable with computers they demand more and more computational resources. In some cases this manifests itself as the demand for more powerful computers that can execute complex tasks faster. However, the major trend in recent years has been for users to simply demand easier access to computers. Personal workstations did not satisfy this need so portable computing became the focus of considerable attention. Various researchers (e.g., Mark Weiser at Xerox PARC [13]), intend to take this even further until computers are so common place and so much in tune with human needs that their users are never even aware of them. This is what has been termed "ubiquitous computing"[13]. Researchers are currently engaged in a program to investigate various computing devices, from tiny calculator-size tabs that are little more than a combination of beepers and notepads, through small portable terminals that have the full power of a personal computer, to wall-size devices that take the human-computer interaction to a whole new level.

For the moment, portable computing is the only manifestation of ubiquitous computing that has gained any commercial success. Unlike the Xerox PARC portable terminals, portable computing has centered mostly around self-contained computers that do not require an external source of power or network connection. These devices,

while generally inferior to the standard stationary personal computers, offer access to computational resources virtually at all times and places. They do, however, suffer from two serious problems (other than their inferior computational power):

1. They are generally heavy due to sizeable rechargeable batteries, and
2. they cannot be used for extended periods of time away from a power supply (generally about 3 hours).

Both these problems can be alleviated by reducing the power consumption. For this reason, low-power techniques are currently a popular topic of research. While power consumption is becoming a serious concern for the designers of the new breed of ultra-fast microprocessors (some of them requiring as much as 30 watts of power [4]), it is the field of portable computing that best motivates research into low-power dissipative technology.

1.1 The Stationary Cycle Server Model

The easiest way of reducing the power consumption of any computing device is to reduce the amount of computation it performs. Power consumption is proportional to $C \times V^2 \times f$ where C is the effective capacitance switched, V is the voltage at which the circuitry operates, and f is the switching frequency. Reducing power consumption involves reducing any or all of the above factors. Reducing C means reducing the computational units active at any given time. This is contrary to the normal goal of performing as much as possible in parallel for reasons of speed. Even if one were to only use one computational unit at any given time, since now the task takes correspondingly longer, the total energy consumed remains the same. In short, the only reasonable way of reducing the capacitance switched is by deferring some of the computation to other resources and thus eliminating some of these computational

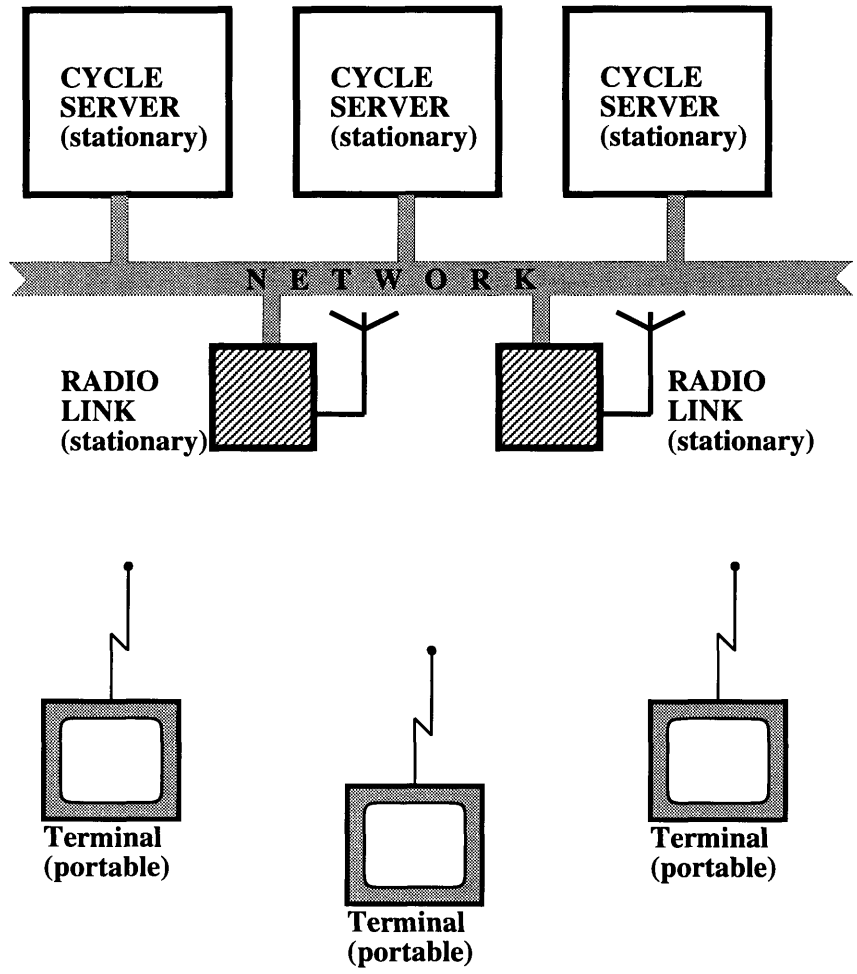

Figure 1-1: The Stationary Cycle Server Model.

units[1]. Reducing f reduces the rate at which these units operate. Reducing V increases propagation delays which forces a reduction in maximum f [2]. Therefore, any attempt to reduce the power of a computational device by any significant amount[2] will result in lower throughput from that device.

While the above conclusion might sound rather pessimistic, an interesting way

[1]Note, however, that one can opt to use more computational units to offset the reduction in speed caused by reducing the voltage V. This can result in a net decrease in power consumption.

[2]For the scope of the above statement, a significant amount would be something close to one order of magnitude.

to work around it has been used by both the Xerox PARC group and the InfoPad group at Berkeley [3, 2]. The portable computer (hereafter called the terminal) can do just enough work to manage the input and output devices and defer all other computation to a separate cycle server (hereafter called the stationary cycle server or just cycle server for short) that will actually perform all the computationally intensive tasks. The terminal and the cycle server can be connected in such a fashion as to allow the terminal to transmit all input to the cycle server and the cycle server to transmit all output back. Then the cycle server can be made stationary, and therefore have an external power source. If the connection method is wireless, then the terminal becomes effectively a portable computer (see Figure 1-1). Since the cycle server has an external power source, the only consumption that needs to be reduced is that of the terminal. This becomes an easier task since the terminal can be designed to perform the minimal amount of computation necessary to manage the input and output devices. Nonetheless, to the user, the terminal appears to have all the processing power of the stationary cycle server.

Under this scheme, all the applications run on the cycle server while the display and input device management programs (e.g., a window server) run on the terminal. Input from the devices is relayed by the terminal through the radio link to the stationary radio station, which then places it on the network. From there, it finds its way to one of the stationary cycle servers, which processes the input and feeds it to the applications. The applications then inform the cycle server of any impending output (e.g., updates to the display), which puts it in the right format and places it on the network. The stationary radio stations take the properly formatted output and relay it over the radio link to the appropriate terminal. Finally, the terminal decodes the message and performs the appropriate updates or output actions.

Both the InfoPad and the Xerox PARC Pads work on this principle. The Xerox PARC Pads are not much lower in power consumption than normal laptops and the main reason why they use the cycle server model is to create a uniform distributed

computing environment. The InfoPad uses the cycle server model specifically for reducing power consumption and, with the core chip-set consuming no more than $5mW$, it is probably the most successful attempt to do so. [3]

1.2 The Xerox PARC Pads

The Xerox PARC Pads are not really closely related to the class of terminals with which this project is concerned, but they constitute an important research project that has employed the Stationary Cycle Server Model. The main idea was to make numerous inexpensive portable terminals available to each person. It was the requirement for the terminals to be inexpensive, rather than the requirement for low power, that led to the use of the cycle server model. Because the Pads had to be inexpensive, no high-speed hardware could be used in their manufacture. Since the users still had to be able to run all their selected applications, they had to run them on hardware other than the terminal hardware. The cycle server model is ideally suited for this.

The Pads manage a 3-level gray LCD display of 640×480 resolution and an internal speaker as the output devices. The input devices consist of a tethered electromagnetic sensing pen and a built-in microphone. They run on a rechargeable battery for about three hours and weigh slightly over 5 pounds. They have 4MB of DRAM, 1/2MB of VRAM, and 1/4 MB of EPROM. Pads are made with off the shelf components exclusively – no custom silicon was designed for them. The bandwidth of the radio link is 250Kb/s and that of the infra-red link is 19.2Kb/s.

The Pad hardware runs a complete X-Windows server which manages all the input and output devices. It then exchanges X-protocol requests over a radio link with stationary workstations which act as the cycle servers and run the applications. In effect, it is a portable X-terminal that operates over a radio link. The Pads achieved their goal of making workstation caliber computing available to everyone and

[3] For comparison's sake, that is not much more than the power consumed by one of the LEDs on a standard keyboard or portable.

in making such computing portable, but they have not been successful at resolving the issues of weight and short operational times without a recharge. The explicit decision to avoid custom components meant that the power consumption of Pads was substantially higher than it could have been. At the same time, since the hardware was relatively standard and there were no significant software issues[4], the Pads could be developed with less effort than similar systems that opted for a full custom design (such as the InfoPad). More effort could then be spent on integrating the Pads into the model of ubiquitous computing which was the final goal of the project.

1.3 The InfoPad Project

The InfoPad project was the true precursor to the work presented in this thesis. As such, it deserves a much closer look than any of the other systems. The InfoPad originated at the University of Berkeley as an attempt to make a low power multimedia portable terminal. The main idea was to make a terminal that required a minimum amount of work on the terminal hardware and deferred all other operations to the cycle server. The hardware could then be optimized for low power given the task at hand.

The first generation terminal consists of a monochrome 640×480 LCD display and an internal speaker. The display is used for standard graphics arising from applications with a graphical interface[5]. The input to the InfoPad consists of a pen and a built-in microphone. Sound input is digitized and encoded in a μ-law encoder, and then packetized and transmitted over the radio link. The uplink carries 64Kb/s of sound input and 4Kb/s of pen input. On the other side, about 1Mb/s of data is sent from the cycle server to the terminal for the text and graphics display, the live video display and the audio output.

[4]The main piece of software that runs on the Pads is the X-window Server. The MIT version of the server is particularly portable and is currently available for over 17 platforms.

[5]There was a second display on later generations that would handle real-time video in color.

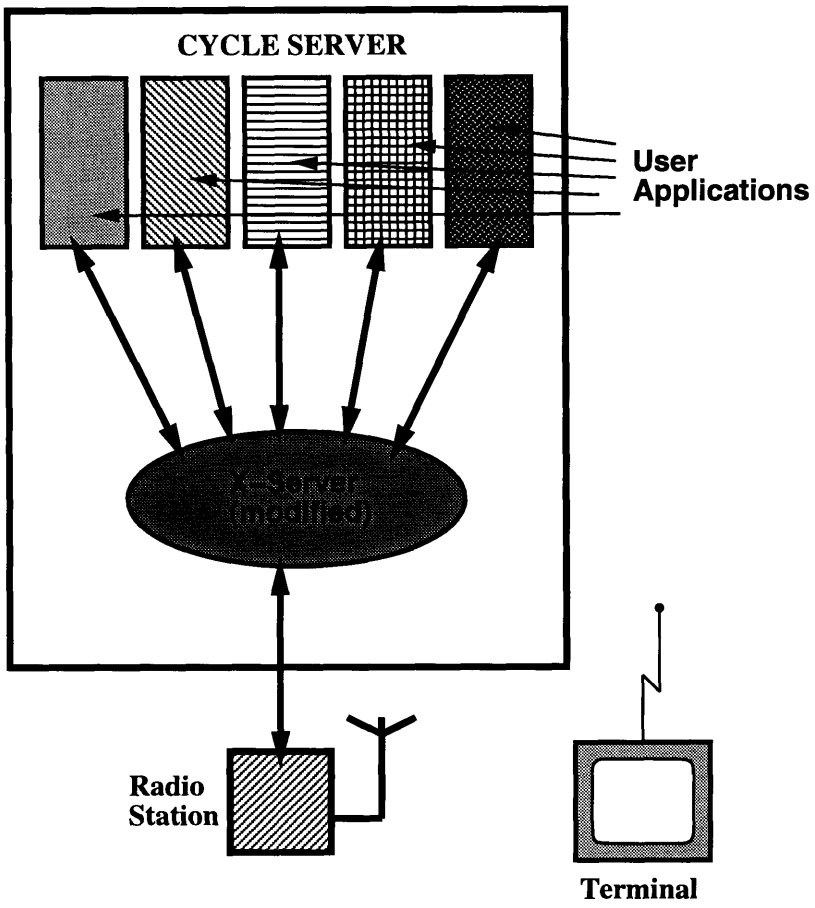

Figure 1-2: The Structure of the InfoPad System.

The InfoPad is also based on the X-protocol although the approach was much different than the one used by the Xerox PARC Pad. Rather than running a complete X-window server on the InfoPad hardware, a new protocol was designed that allowed most of the X-window server tasks to be moved to the cycle server. Thus, input from the pen is relayed from the InfoPad terminal to the cycle server over the radio link. There, it is sent to a modified X-window server that is running on the cycle server. The X-server processes the input as if it came from a local keyboard and mouse and then generates events for the applications. The applications perform any processing appropriate for the input they receive, and send any output requests to the X-server (just like they would if the X-server was to display it on the local screen). The X-

server calculates the necessary updates to the display, packs that information through a special protocol, and relays it over to the terminal. Finally, the terminal unpacks that information and updates the screen accordingly (see Figure 1-2).

One of the main design issues was the nature of the protocol to be used between the X-server that was running on the cycle server, and the terminal itself. Since the main design goal was to make the power consumption of the terminal as small as possible, the protocol was designed such that the hardware requirements on the terminal side were as simple and as minimal as possible. For the first generation of terminals, it was decided to send the full display updates as a series of bits for every pixel that had to be modified on the display. For every scan-line[6] or group of scan-lines, an address would be sent that would say where the scan-line begins (which column and row)[7]. Then all the pixels that needed to be updated on that scan-line would have their values transmitted to the terminal. Later generations used enhanced protocols that had lower bandwidth requirements. Note that this description only applies to the text and graphics display. It does not cover the video portion of the output, which used a different protocol to transmit compressed video. The protocol hardware would determine which unit a given packet was destined for and then dispatch it to the appropriate one of the three (graphics, video or audio).

Because of the relatively simple protocol that was used between the X-server and the terminal, there was very little processing to be done after a packet for the graphics display was received. That makes the hardware small and fast and the InfoPad group took advantage of this by casting all processing to silicon. Given that the best way of reducing power is reducing the supply voltage which causes a system slowdown[2], the fact that the protocol could be decoded in hardware on the terminal side was a particularly important feature.

[6] A scan-line is a row of pixels on the screen, usually represented by a series of consecutive memory cells in the frame buffer memory. Updates do not have to be complete scan-lines – in fact, multiple segments of the same scan-line might be updated by a single request.

[7] The information contained in the row/column pair and the index into the frame buffer is identical so the two are interchangeable.

However, that simple protocol turned out to have a some serious consequences. To begin with, the fact that all the bits that were updated in the frame buffer had to be transmitted to the terminal caused a serious bandwidth problem. For example, some applications would draw text by copying the pixels for the characters to the screen. While this was bad enough in terms of bandwidth requirements in monochrome, the problem would be 8 times more severe for 8-bit color displays. This design causes two problems:

- It would require large bandwidths on the downlink that sends the data from the cycle server to the terminal. This means that all the protocol and decoding logic would have to be correspondingly faster to deal with the large influx of data. Also it means that when large portions of the display change (such as the example of putting a large amount of text on the screen), the latency of transferring such a large amount of data becomes the dominant portion of the response time of the whole system.

- Since the implementation of the protocol is now in hardware, it is particularly inflexible, requiring the design and fabrication of new components for every change in protocol that needs to be made.

These are the two main problems with the InfoPad terminals. Also, there was a strong tendency to move to color[8]. Most color frame-buffers use 8-bit pseudo-color, which means that each pixel now needs 8 bits of information. For a protocol as simple as the InfoPad protocol to move to 8-bit pseudo-color, one would pretty much have to design the terminal hardware to handle 8 times the amount of data and increase the bandwidth available from the radio link by about 8 times. This would definitely be a serious problem which is unlikely to be solved without changing the protocol.

This thesis is part of a project to extend the work performed by the InfoPad group[9]. The basic premise of a low power terminal achieved by using the stationary

[8]Actually, this only applies to the graphics display. The InfoPad video system is in color.
[9]The InfoPad group is also working on ways of solving the above problems.

cycle server model is still the goal, but having the benefit of all the data collected by the InfoPad group, we decided to use a different approach to the problem. To begin with, we decided to replace the InfoPad's hardwired protocol with a general-purpose processor, highly optimized for the task at hand, at the expense of higher complexity and larger power consumption. Thus, the terminal and its protocol would become infinitely flexible, allowing us to make use of better algorithms, new extensions to the X-server, and to make additions and modifications to the protocol at will. Then we decided to employ a more complex protocol which would reduce the bandwidth requirements and alleviate all the problems associated with the high bandwidth, while at the same time allow the use of color. This more complex protocol was the first step in the project and is the subject of this thesis.

Chapter 2

The Communication Protocol

As described in section 1.3, we decided to use a more intelligent protocol than the simple protocol that was used by the InfoPad, mainly to reduce the amount of information that has to be sent on the downlink to the terminal. By giving the protocol the notion of higher level elements (for example, rectangles, polygons, lines etc.), it is possible to create a protocol that requires much less information to perform the same activities. Consider again the example of section 1.3. If the protocol had a notion of what a text character is, one would only need to send a command that identifies the character, its position, and the color to which it should be drawn. We decided to give the protocol the notion of such higher level elements to reduce the bandwidth requirements. We also decided to use color, therefore the protocol had to work for 8-bit pseudo-color frame buffers with a 640×480 resolution.

2.1 The Raw Graphics Protocol

The first step in developing a protocol was to come up with the exact type of requests that would be necessary, and the information that these requests would require in order to allow the terminal to reproduce the display that the X-server is trying to build. Obviously, a big part of this task is simply deciding which higher level elements

(and operations on such elements) would be the best to include in the final protocol. This set of requests, elements and operations forms what we call the Raw Graphics Protocol. Given this protocol, it would be possible to make a perfectly functioning terminal that would operate with no errors assuming that none of the request data was corrupted. Just like the InfoPad and the Xerox PARC Pads, we decided to base our protocol on the X-window system.

2.1.1 The Reasons for Choosing the X-window System

The decision to use the X-window system as the basis for our protocol was actually one of the easiest design decisions that had to be made. There are so many advantages to using X that almost all other windowing systems were excluded very early on. Here is a list of the advantages that were the most important for this project:

Architecture The X-server (the only piece of software that would have to be modified on the cycle server) is designed in such a way as to separate the code as much as possible into distinct functional modules[7].This makes it much easier to intercept communication between such modules, which will be shown later to be absolutely necessary to implement the raw graphics protocol. The main module that had to be modified was the Color Frame Buffer module, or "cfb" for short[1]. This code is in fact the code that performs all the actual drawing on the screen. Other modules that had to be modified were the Machine-independent Frame Buffer module, or "mfb", that takes care of some of the most esoteric drawing operations, and the Hardware module, or "`hw/sun`"[1] module, which takes care of some of the hardware specific issues (such as handling colormaps and input from the protocol instead of the cycle server's keyboard and mouse).

[1]We tried to make the emulation environment as general as possible. However, when it was necessary to make hardware specific changes, only the Sun hardware drivers were modified to avoid excess work. We used Sun platforms for all parts of our emulation environment.

Efficiency The current version[2] of the X-server has been very heavily optimized for most operations on frame buffers having 8 bits per pixel.[3] Since the X-server code forms the basis of all our drawing operations, we did not need to expend any effort optimizing it and it was in fact much more efficient than anything we could have done on our own.

Availability of Source Code Having access to the source code would allow us to reuse all the highly optimized code in the X-server drawing routines. It would also allow us to modify an already existing and developed system, rather than develop one from scratch. Given the fact that the X-server code contains over 100,000 lines of code, developing it from scratch would require a huge amount of time. Unfortunately, source code is not available for almost any of the commercial windowing systems. MIT makes available the source code for its own version of the X-server, which is very widely distributed.

Portability All of the above mentioned advantages would be totally useless unless the code could be used in the final protocol on the final terminal hardware. The MIT implementation of the X-server is one of the most portable software systems publicly available, and currently supports more than 17 different platforms ranging from 16-bit machines on the low end, to 64-bit machines on the high end, with widely different operating systems and hardware architectures[5]. This is expected to make the task of porting the code to the terminal hardware a lot easier. Furthermore, given the fact that the hardware architecture is likely to suffer a lot of changes during the experimentation phase, portability becomes a particularly important concern.

[2]The emulation environment and the protocol were built on top of the X11 R6 version of the MIT X-server.

[3]This is by far the most common configuration of frame-buffer hardware although 24-bit per pixel true-color displays are becoming very popular for multi-media applications.

Flexibility The X-window server has provisions for what are called "extensions" to the server. By writing an extension one can give the X-server new abilities that were not part of the original protocol as defined by the X Consortium. For example, various people have written input extensions to handle pens and graph-pads, MIT has written the Shared Memory extension that allows the X-server to communicate with clients running on the same machine much faster, and so on.

Networking Capability The X-window system was designed from scratch to support clients that are not local to the machine running the X-server itself. This means that a single terminal could service user applications running on multiple machines at the same time. None of these machines have to be the same as the one that is running the modified X-server. This gives an added degree of portability to the terminal since it can be used to access machines which are at very remote locations, as long as those machines can communicate with the cycle server that serves the terminal.

Widespread Popularity The X-window system is one of the most widespread windowing systems in use currently. It comes standard with most engineering workstations and is available for most other platforms. This means that there are a multitude of applications that would be capable of running on the terminal the moment the first prototype of the terminal is available.

2.2 The Partitioning Issue

The X-server is built with a particularly modular structure that consists of 3 main modules (see Figure 2-1): On the top level is the DIX layer, or Device Independent layer, and below that are the Operating System layer or OS, and the DDX layer or Device Dependent layer. Each layer consists of sub-layers that exchange information with each other. At the top of the DIX layer is the Transport layer that receives

requests from applications. It then feeds these requests to the lower sub-layer of the DIX layer, the Screen functions that determine where the request information should be sent next[4]. The Screen functions then pass the request information to the top sub-layer of the DDX layer, a set of functions called the Graphics Context Operations (or GC-ops for short). These functions decide how to best go about servicing the request and are responsible for most optimization procedures inside the X-server. The same request, for example, might be serviced by two different routines at two different times, depending on the current state of the X-server. These functions then pass the request information to the bottom sub-layer of the DDX layer, called the Color Frame Buffer code, or CFB module for short. Finally, the CFB layer changes the appropriate pixels in the frame-buffer to actually service the request. The OS layer acts as an interface between the rest of the X-server layers and the Operating System of the machine. The memory allocation operators, for example, are part of the OS layer.

A communication protocol for the terminal can be implemented by partitioning the X-server between any two of the above layers and intercepting the communication between them, to relay that information to the terminal. By reproducing all the layers below the partition on the terminal side, one can reproduce all the actions of the X-server while servicing the request, and therefore one can reproduce the final display (see Figure 2-2). That, of course, poses the question of where the partition should be placed. Note that the partition does not necessarily have to lie on layer boundaries[5]. It could be arbitrarily placed somewhere in the middle of a sub-layer. It is also possible (and in fact necessary for optimization reasons) to place the partition at different levels for different parts of the code.

Generally, levels that are higher up have access to more of the basic elements (see

[4]Each X-server can handle multiple screens. This sub-layer decides which screen each request is meant for.

[5]In fact, despite the modular structure of the X-server, the boundaries between sub-layers are not all that clear. Only the boundaries between layers are.

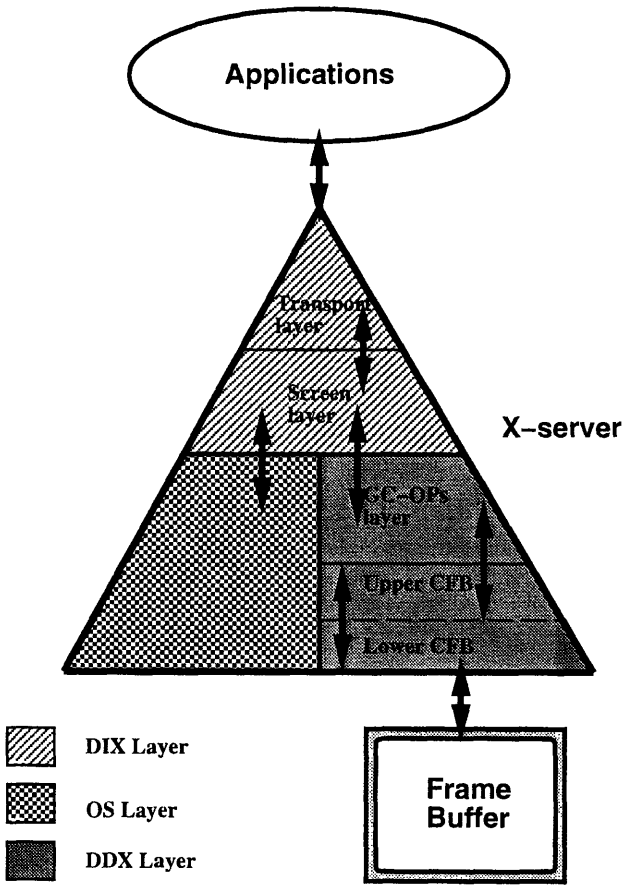

Figure 2-1: The Internal Structure of the X-server.

section 2.2.2) than levels that are lower. However, by placing the partition lower, one can simplify the protocol as well as avoid performing the work of the higher layers on the terminal. Section 2.2.1 explains why avoiding the work of the higher layers is particularly important in maintaining low power consumption by the terminal.

On one extreme, the partition could be placed on top of the Transport layer. Effectively, this means running the complete X-server on the terminal. This is the approach that was used by the Xerox PARC Pads. This would make available to the protocol all the information that could possibly be available. The bandwidth required for the downlink would be pretty small since all requests would be using higher level elements, and would have available to them all the X-server state. On the other hand,

this would place very heavy demands on the terminal hardware.

On the other extreme, we could place the partition right below the CFB sub-layer and intercept the communication between it and the frame-buffer. Since the communication between the CFB sub-layer and the frame buffer consists of the values of the individual pixels, this is effectively the same as the first generation InfoPad protocol. It has very minimal terminal hardware requirements but suffers from high bandwidth requirements.

For the purposes of the current protocol we decided to place the partition at an intermediate point. While it is true that levels higher than the CFB sub-layer have access to more elements than the CFB layer[6], the lower levels of the CFB sub-layer have access to most elements that take part in the actual drawing operations. We decided to place the partition within the CFB layer, immediately above the routines that actually perform the drawing operations in the frame buffer (see Figure 2-2). This lower portion of the CFB sub-layer has all the basic notions of geometric objects that might be drawn on the screen, but lacks the notion of all other elements (such as the notion of windows, for example). For most drawing operations, this means that various pieces of information about higher level elements such as windows, has to be transmitted every time (for example the position and clip-region[7] of a window). Generally, this overhead is not significant enough to warrant the extra complexity of a higher level. For some parts of the code (mainly the portions of the CFB code that handle text) we had to place the partition in the higher levels of the CFB sub-layer, since it would provide us with a substantial bandwidth advantage.

2.2.1 Low Power Issues

The main tradeoff in designing the protocol is between bandwidth and complexity. The main reason why complexity needs to be kept under control is that it affects the

[6]The Screen information is such an element.

[7]The clip-region is the portion of the window that is actually visible at any given time. Parts of the window might be obscured by other windows.

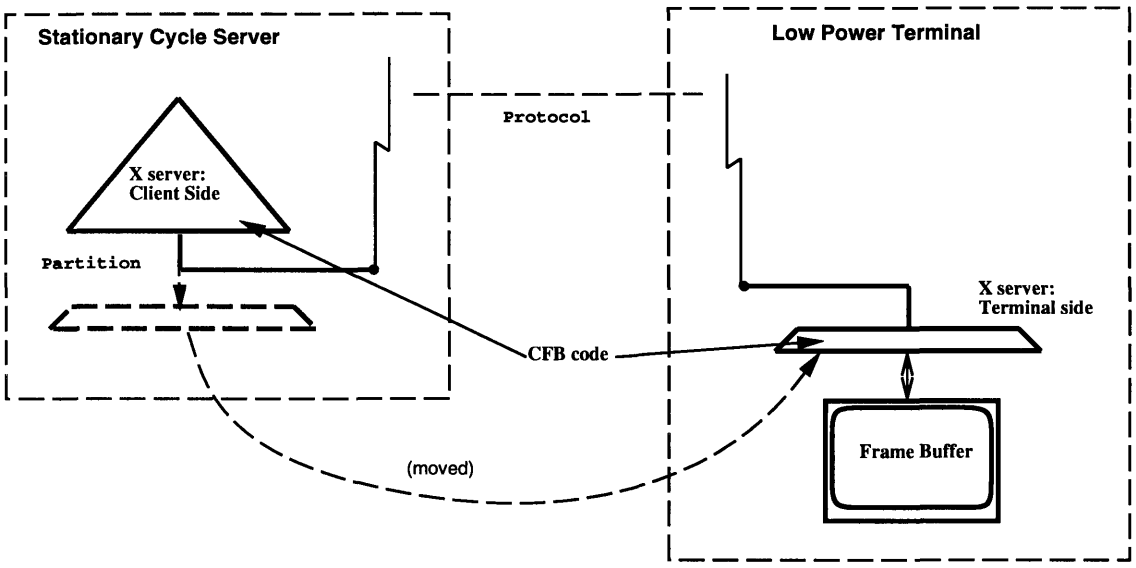

Figure 2-2: The Partitioning of the X-server to Obtain the Protocol.

final power consumption of the terminal hardware. There are three major parameters that are dictated by the protocol complexity which affect the terminal hardware power consumption:

Number of Tasks The more complex the protocol, the more it will have to do in order to service each request. This means that more throughput will be required from the terminal hardware. More throughput will either require more processing units working in parallel or a higher clocking frequency, both of which increase the power consumption.

Memory Requirement We would like to limit the amount of memory that is necessary on the terminal hardware to process the protocol. DRAM is particularly expensive in terms of power because it needs to be refreshed and it is therefore constantly switching. SRAM is not as expensive in terms of power consumption, but it is particularly expensive in terms of silicon area occupied. That silicon area could be used for architecture-level optimizations that would make the hardware more efficient and thus allow a reduction in clock frequency and

therefore in power consumption. Therefore, both the size of the code that will need to run on the terminal hardware, and the amount of state it needs to save, must be kept minimal. Both sizes tend to be smaller for simpler protocols.

Better Mapping to Hardware If the protocol is particularly simple, with most operations being of the same type, then there is a greater opportunity for architectural optimizations to the hardware. For example, if the protocol is limited to calculating rectangles on the screen and filling them in with a specified value, the terminal hardware would benefit a lot from an address generator (see section 4.3). Such architectural optimizations allow a reduction in clock rate (which allows a reduction in voltage supply as well) thus providing very attractive power savings.

2.2.2 The Element Types and Operations

Once the location of the partition in the X-server was determined, deriving the protocol requests (and hence the element types and operations) was a simple matter of determining what information needed to be relayed to the terminal to reproduce the drawing actions of the X-server. Effectively, all the drawing routines had to be intercepted to make sure that any operations that affect the display are reproduced on the terminal side. Since the partition was placed right above the drawing routines, it was generally unnecessary to save any state for longer than a single request. All the information needed by each drawing routine could be relayed to the terminal for each request. The only exception to the above rule was the caching of fonts which is explained later in this section.

An extensive list of the individual requests and their formats would be too long to describe here and, at any rate is unnecessary. If the reader would like to gain a better understanding of the individual requests and their formats, he/she is advised to refer to Appendix A for instructions to unpack the source code and then look at the individual code files. The file `Xserver/cfb/cfbrequests.h` contains a listing of

all the request identifiers as well as the file in which the corresponding routines can be found.

Here is a listing of the basic elements that the protocol is aware of:

Boxes These are rectangles and form the basis of many of the drawing operations. They are used both as drawing primitives (e.g. when the background of a window is painted, it is drawn as a rectangle of a given color) and as control information for the processing of other elements (e.g. regions are made up of a list of boxes).

Lines The X protocol defines lines of varying widths and styles. The drawing routines (and, subsequently, our communication protocol) follow the X protocol definitions of these blindly. The only exception is zero-width lines[8], which can be drawn by a more efficient algorithm and therefore have their own request identifier. Lines are only used as drawing primitives.

Arcs The same comments that apply to lines also apply to arcs.

Ellipses This includes circles. The same comments that apply to lines and arcs also apply to ellipses.

Chords These are portions of an ellipse. Again, the communication protocol follows the definitions of the X protocol for chords.

Pixmaps These are pictures in a standard format. They can have varying depths (roughly equivalent to bits-per pixel). The MIT X11 R6 server can handle pixmaps of any depth but the only ones actually used by applications are of depth 1 and depth 8. Depth 1 pixmaps are also called bitmaps for convenience. Pixmaps can be used both as drawing primitives (e.g. by copying the pixmap

[8]Zero-width lines are defined as lines that are drawn with the minimum width possible – in this case one pixel-wide.

to a portion of the screen) and as control information for other requests (e.g. as tiles and stipples).

Regions These are lists of boxes that define a certain portion of the screen. They are generally used as clip regions to restrict other drawing operations to the visible portions of a window. Regions are only used as control information.

Text Strings Again, the X protocol specifications are followed blindly for these except in the case of image text. For optimization reasons, an image text request is broken down to a draw box request and a normal text operation. Text strings are only used as primitives. The drawing routines for text strings take as input the pointers to the actual character bitmaps (also known as glyphs) rather than the string and the required font. It was therefore necessary to intercept text operations at a higher level where the string and font information was available.

Fonts These are the actual font types that the text drawing routines use. The X-server actually creates the fonts – they are simply transmitted to the terminal when necessary. This allows the terminal to use any fonts that the X-server has access to. Since fonts are particularly large structures and their use exhibits a lot of temporal and spatial locality, they are cached. It would be prohibitively expensive in terms of bandwidth to retransmit fonts for every text request. The font cache is the only state that transcends request boundaries (i.e. lasts for more than a single request). The cache size is variable and the protocol can accommodate cache sizes as large as 256 fonts. A much smaller size (on the order of 6 entries) is generally adequate for most types of use. When the cache is full, cache entries are overwritten using a Least-Recently Used (LRU) criterion.

Colormap Entries These are the entries in the table (colormap) that translate the 8-bit value of a pixel to an actual RGB value that can be displayed by the hardware. Note that the protocol is not aware of the colormaps themselves

(even though the X protocol allows for multiple colormaps) since it intercepts these requests at the hardware level and there is only one colormap at that level.

The following operations are available to the communication protocol:

Basic X protocol operations [11] The X protocol defines 16 operations by which the new drawing primitive can be combined with the existing contents of the frame-buffer to yield the final image. All 16 are supported by the communication protocol. Some of them are optimized and have their own request identifiers (e.g. the Copy request which overwrites the old contents of the frame-buffer and therefore does not need to read the frame-buffer – only write to it).

Stippling Operations These are operations that use a bitmap as a mask in order to only affect certain portions of the image. They can be used to create effects such as a transparent pixmap (an image overlaid over an existing display so that the old display can be seen through the uncolored portions of the image).

Clipping Operations These are operations that restrict the basic drawing operation to a particular region (that might contain multiple boxes). They are generally used to limit drawing operations to only the visible portions of a window.

Input Operations Five requests are reserved to allow the transmission of user input (e.g. from a mouse or keyboard) from the terminal to the cycle server. These are:

Pointer Motion This request informs the cycle server that the user has moved the main pointer (e.g. a mouse or pen) to the new location mentioned in the request.

Button Press This informs the cycle server that one of the buttons on the main pointer was pressed. The button identifier is part of the request.

Button Release This informs the cycle server that one of the buttons on the main pointer was released. The button identifier is part of the request.

Key Press This informs the cycle server that one of the keys on the keyboard was pressed. The key identifier is part of the request.

Key Release This informs the cycle server that one of the keys on the keyboard was released. The key identifier is part of the request.

The above set of elements and operations forms what we call the Raw Graphics Protocol (the fully functional protocol before error correction, detection and retransmission).

2.2.3 The Emulation Environment

In order to test the raw graphics protocol (as well as the final protocol) and make some basic measurements on it, an emulation environment was built. This environment consists of:

- A workstation connected to a network that will act as the cycle server. The workstation has to be X-compatible and capable of running the modified X-server.
- Another computer connected to the same network so that it can communicate with the first workstation. This computer has to have an unmodified X-server running, in order to run the terminal emulation application.
- A modified version of the X-server that has been partitioned in accordance with the raw graphics protocol. This X-server has to be identical to the original X-server except for the fact that the information intercepted at the partition in accordance with the raw graphics protocol is sent via the network to the terminal emulation application running on the second computer[9].

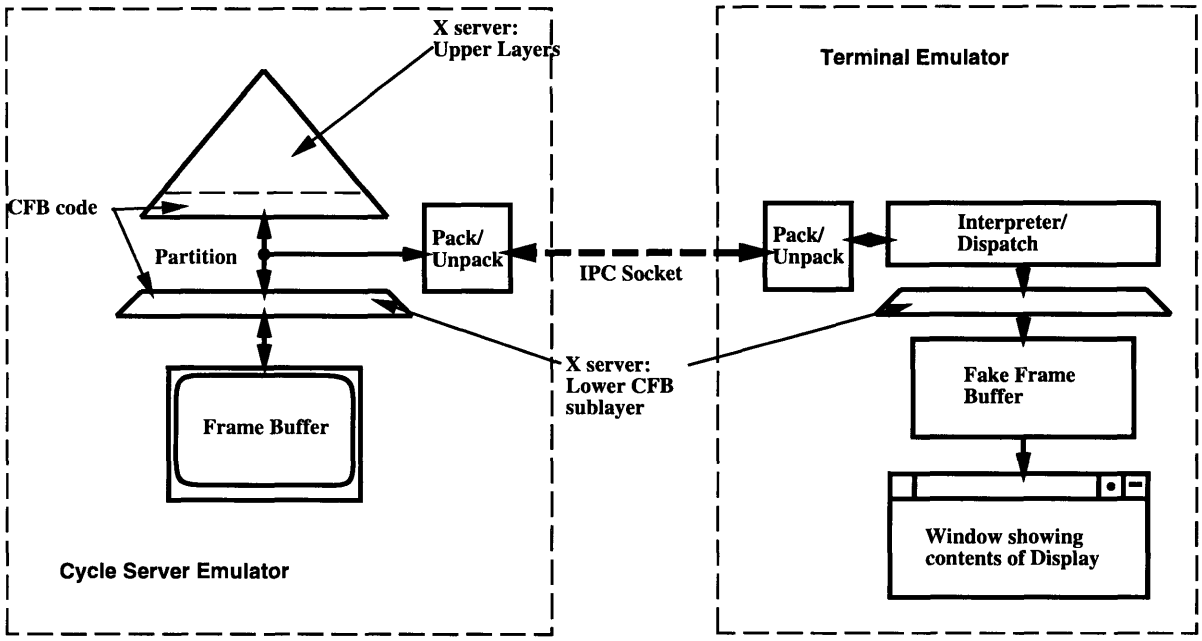

Figure 2-3: The Structure of the Emulation Environment.

- A terminal emulation application that runs on the second computer. This consists mainly of the software that will interpret the protocol on the final terminal hardware, except for the fact that it draws into an area of memory instead of a real frame buffer. It displays an X-window which, upon receipt of an X expose event, loads the contents of the fake frame buffer into the window so that it can be viewed.

Figure 2-3 shows the final structure of the emulation environment.

The cycle server emulator runs on a Sun 4/260 running Unix. The modified X-server actually displays the contents of the frame buffer on the original screen so that they can be compared with the output of the terminal emulator. It communicates

[9]Strictly speaking, this is not absolutely true. Since our final prototype will run with a 640×480 display, it was necessary to modify the X-server to assume this resolution instead of 1150×900 that was standard on our hardware. This was the only way to guarantee accurate measurements. Furthermore, it was necessary to modify the X-server to receive input from the terminal emulation application rather than its own keyboard and mouse.

with the terminal emulator using TCP/IP sockets (the main UNIX IPC substrate). The terminal emulator contains the code that interprets the protocol and draws to the fake frame buffer. It is actually an X-application itself (which is why it should not be allowed to connect back to the modified X-server). It pops up a window in which it displays the contents of the fake frame buffer and from which it receives keyboard and mouse input. For efficiency reasons, it only updates the display when the window receives an expose event[10]. Assuming that a window manager is running on the same machine. such events can be generated at will by iconifying the window and de-iconifying it again. We ran our terminal emulator on a number of Sun platforms running UNIX.

The terminal emulator also contains code which dumps data about the requests to two files, allowing measurements to be made. In the final version it also included noise injection code (to simulate a noisy wireless channel) and error correction, detection and retransmission code.

2.3 Error Correction

The raw graphics protocol is all that is necessary to allow the terminal to work in the absence of any noise that can corrupt the data. Wireless communication channels, however, are notorious for containing strong noise sources. An unfortunate consequence of making a protocol out of higher level primitives is that its noise tolerance decreases dramatically. In the InfoPad protocol, most of the data represents pixels on the screen. If any of that data gets corrupted, the end result will be that some pixels on the screen will have the wrong color. For our raw graphics protocol, however, a very substantial fraction of the data is control information that dictates the request type, the clip regions, the locations of the primitives, etc. If any of that data gets corrupted, the results could be disastrous. The primitive might be drawn to the

[10]This type of event occurs when a part of the window that was previously obscured is exposed because of window rearrangement.

wrong place on the screen, it might be clipped severely or even the wrong request might be detected at the terminal side with completely unpredictable results. The raw graphics protocol is therefore particularly ill-equipped to deal with noise in the communication channel.

We, therefore, decided that it was absolutely necessary to use an error correcting code to protect the data from noise. One possible approach would be to wrap the data in an error detecting code (e.g. a cyclic redundancy check or CRC) and then retransmit any requests that were corrupted. However, one of our design goals was to make the protocol work relatively well[11] with Bit Error Rates (BER for short) as high as 10^{-3}.[12] Given the size of some of the requests, the above scheme was likely to result in an unacceptably large number of retransmissions since a large number of requests would have at least one bit corrupted. A better approach would be to use a code that can actually correct many of the errors, thus avoiding the need for retransmission in most cases.

2.3.1 The Reasons for Choosing the RS(15,9,7) code

A number of error correcting codes were explored for use in the protocol. The final choice was a Reed-Solomon code that works on 4-bit symbols, encodes 9 symbols to 15, and detects and corrects 3 simultaneous errors[6, 8]. The main advantages of this code that lead to its final use are listed below:

Immunity to Burst Mode Noise With the capability of correcting up to 3 simultaneous errors in any of the 15 symbols that make up the encoded codeword, it will require bursts of noise at least as long as 10 bits before the code will fail to

[11]It is not possible to guarantee how well any wireless system will operate in the presence of noise. In the worst possible case, a large object could be obscuring the transmitter, in which case, irrespective of the protocol, it will be impossible for the terminal to receive any requests whatsoever.

[12]This means that, on average, one bit in every 1000 is corrupted. Both the expected noise characteristics and the noise produced by our emulator environment have a large content of burst noise which is a lot harder to overcome than simple random noise.

correct the errors. After two-way interleaving (the interleaving that was used for our protocol), the shortest burst that will be uncorrectable is 22 bits.

Large Error Detection Capacity After two-way interleaving, the shortest burst error that will be undetectable by this code is 22 bits. As is described in the next section, error detection capability proved to be particularly important and it was in fact necessary to enhance the error detection process by the use of a CRC.

Small Size The code is relatively small, each codeword having 15 symbols (60 bits). This is important for two reasons:

1. The whole encoding process can be cast into hardware making it much more efficient both in terms of speed and in terms of power. Had the code been bigger, it would have been prohibitively expensive in terms of silicon area to cast it to hardware.
2. When the number of bytes does not exactly fit into an integer number of codewords, it has to be rounded up. This means that some bytes are sent which are unnecessary and increase the overhead of the code. This overhead is bigger for bigger codes.

2.3.2 The Two-Level Encoding Scheme

Our preliminary results from the raw graphics protocol (see section 3.1) indicated that the largest amount of bandwidth was consumed by pixmaps being sent as raw data for copying onto the display (e.g. little icons on applications or large images in image viewers). Since these consisted of individual pixel data rather than control information, the worst possible outcome of such data being corrupted would be some pixels being displayed in the wrong color. Experience with the InfoPad showed that, while such errors create a less pleasing display to the user, it is nonetheless acceptable.

There was no reason, therefore, to incur the large overhead of error correction (over 66% for our error correcting code) just to protect the pixel data.

We decided to use the same, two-level encoding scheme that the InfoPad project used. We arranged the packing of information so that all the control information comes first and the pixel data (if any)[13] comes last. The error correction system only corrects the first portion of the message which contains the control information. The number of bytes that are encoded is stored in the header of each request so that the other side knows how to re-assemble the message. Note that the unencoded data is never retransmitted unless the header could not be read (see section 2.4).

2.4 Error Detection and Retransmission

Despite the large error correcting capacity of the Reed-Solomon code, measurements with our emulation environment showed that there were a large number of errors that the code could not correct (see table 4.1)[14]. The code would still detect most of these errors though. Furthermore, most of the requests had only small amounts of encoded control information (generally less than 100 bytes), but fonts were all encoded and resulted in particularly large requests (from 4Kbytes for the small fonts to 12Kbytes for the larger ones)[15]. In such cases there was a substantial probability that the request would suffer uncorrectable errors. In order to deal with such errors, an acknowledgment and retransmission scheme was developed.

The system operates as follows:

- Each request has associated with it a 16-bit ID. IDs are incremented for each request and are re-cycled after the counter reaches the count of 65535 (which

[13]In fact, only four of the requests have any pixel data associated with them – all others consist solely of control information.

[14]In our noise model, this was generally independent of the actual noise rate and was due to lengthy burst errors.

[15]The actual font requests would be that big only when there was a cache miss and the full font structure had to be relayed to the terminal. For cache hits, only the cache slot number has to be sent which is only one byte.

is equal to $2^{16} - 1$). After the X-server sends a request to the terminal it stops and waits for an acknowledgment from the terminal.

- If the terminal cannot decode the header, then it has no idea how to re-assemble the message since it does not know what portion of it is encoded. That information, along with the total size of the message is stored in the header. It therefore flushes its input[16] since it is not possible for it to determine how many bytes to read to remove the whole message. Note that it can only flush the current message since it has not sent an acknowledgment yet and the X-server will not send any more messages until it receives an acknowledgment. After it has flushed the input, it sends an acknowledgment to the X-server requesting a full retransmission (ACK_RETX_COMP).

- If the terminal decoded the header properly, it proceeds to read the rest of the message (it now knows the total message size) and to decode the encoded portion of the message. If it cannot decode the encoded portion properly, it retains the unencoded portion and throws the encoded portion away[17]. It then sends to the X-server an acknowledgment requesting retransmission of only the encoded portion (ACK_RETX_PART).

- If the terminal managed to decode both the header and the encoded portion properly, it sends to the X-server an acknowledgment that asks it to proceed with the next request (ACK_OK) and then reassembles the message and dispatches to the appropriate drawing routine.

- If the X-server receives an ACK_RETX_COMP it retransmits the whole request and waits for another acknowledgment.

[16] Flushing the input means removing all data at the input queue that is still waiting there.

[17] It is not considered a significant error if the unencoded portion is corrupt which is why we did not encode it in the first place.

- If the X-server receives an ACK_RETX_PART it retransmits only the encoded portion (including a new header that reflects the fact that there is no unencoded data) and waits for another acknowledgment.

- If the X-server receives an ACK_OK it proceeds with the next request.

Acknowledgments are carried on the same channel that carries all the other requests. The above procedure only describes the process for acknowledging requests sent from the X-server to the terminal. In the other direction things have been purposely kept a lot simpler. If the X-server receives any request it cannot properly decode, it simply forgets about it and proceeds to the next one. The reason for this is that if one were to acknowledge requests in both directions, both directions would need timeouts in case the ACKs get corrupted. Using symmetric acknowledgments is particularly complex[12], while the consequences of asymmetric acknowledgments were not serious enough to justify the extra complexity. In fact, the reason why we did not use finer grain acknowledgments is that packet level acknowledgments require a symmetric acknowledgment scheme[18]. There are, however, some downsides to asymmetric acknowledgments:

- If an ACK gets corrupted, it is ignored instead of being passed to the code that is waiting for an ACK. It is possible therefore for the system to deadlock. In order to avoid that, a timeout was built into the terminal software that retransmits the last ACK if it does not hear from the X-server for more than a fraction of a second. Occasionally though, the X-server will not be sending any requests until the user sends it some input. In this case, the timeout still causes ACKs to be retransmitted causing a background level of activity as an overhead. This is not a serious concern since it only happens when there is no other activity between the X-server and the terminal, and it does not therefore cause resource congestion.

[18]Another reason was that it would make the overhead higher by transmitting a lot more acknowledgments.

- It is possible for input events from the user to be corrupted and lost. In the case of pointer motion events, this is completely insignificant since it only causes an irregular jump in the motion of the cursor. In the case of button and key events, no incorrect behavior results; the user simply has to perform the action again. These are considered minor irritations rather than problems.

Again, despite the high error detection capacity of the Reed-Solomon code, it is still possible for some errors to remain undetected (see section 3.2). Such errors will result in unpredictable behavior. It was, therefore, deemed necessary to provide an additional system of error detection. It was decided that a 16-bit CRC would provide adequate protection against otherwise undetectable errors. We decided to use a CRC that is widely used in modems (since the noise found on modem lines is very similar to the noise found on wireless channels). The CRC is based on the polynomial $x^{16} + x^{12} + x^5 + 1$. Before the encoded portion of the message is actually encoded, a CRC for it is calculated and inserted into the message header[19]. On the terminal side, when the message is decoded, the CRC is calculated again and compared to the value found in the header. If the two do not match, an ACK_RETX_PART is sent to the X-server[20].

[19]Note that the CRC cannot be performed after the message is encoded. If it was, it would declare the data corrupt even after the Reed-Solomon code had corrected it (if correctable), and ask for an unnecessary retransmission.

[20]Both the Reed-Solomon encoding and the CRC calculations were described for the X-server requests to the terminal. Note, however, that the requests going from the terminal to the X-server (including ACKs) are also encoded with both the CRC and Reed-Solomon codes. They are just not acknowledged in any way.

Chapter 3

Results and Measurements

Our emulator environment allowed us to make several sets of measurements to evaluate the overall performance of the protocol. Based on these results, we modified some of our design decisions and set targets for the future development of both the protocol and the hardware for the terminal. These measurements also allowed us to evaluate the feasibility of the concept of higher level protocols.

Two sets of measurements were taken. The first set of measurements was taken before the inclusion of error correction, detection and retransmission (on the raw graphics protocol alone). The second set of measurements was taken on the overall final protocol and was used to evaluate the error correction and retransmission protocols. The reason why we performed two sets of measurements is threefold:

- A preliminary set of measurements was needed to refine the raw graphics protocol before we moved over to the more complex final protocol.
- The preliminary set of measurements gave us a relatively accurate picture of what would be necessary or unnecessary in the error correction and retransmission system in order to get good protection against noise without significant overhead. For example, the preliminary measurements showed that most of the bandwidth was consumed by pixmaps being sent across for copying on the final

screen. This is why we decided to use a two-level encoding scheme as opposed to encoding all the data.

- Although the emulator environment had an adequate response time before we included the error correcting, CRC and retransmission systems, response time degraded drastically in the final system. One of the reasons was the added computation that needed to be performed for ECC and CRC – another was the fact that the X-server would now stop and wait for acknowledgments so network latencies suddenly became important. Both of these reasons will be eliminated in the final system by using custom hardware for ECC and CRC, and a dedicated radio link, so we expect the response time to be much shorter in the final system. Since response time affects the way a user reacts to the system and the eventual utilization characteristics, it affects the measurements we make. For example, if the response time is really slow, the user spends most of his/her time waiting for the system to complete its processing and prepare for input. The number of actions that he/she takes per unit time decreases so the number of requests the X-server has to service per unit time decreases. This causes an artificially low bandwidth among other things. We decided, therefore, to use the measurements from the raw graphics protocol system and extrapolate from those using statistics obtained from the final system to calculate the final bandwidth and reliability.

The primary goals of the measurements made on the raw graphics protocol was to allow us to see which requests were being used most frequently so that we can optimize them further, and to give us some idea of what to expect for a bandwidth requirement. We knew that this would depend on the type of application used, so we classified applications into three models and made measurements for each. These models are:

High graphics content These are applications that have a high graphics compo-

nent, with pictures and elements other than lines, rectangles etc. Such applications tend to send large pixmaps to the X-server to be drawn on the screen. These pixmaps have to be sent over the radio link by the protocol. The end result is that such applications have high bandwidth requirements and, by far, most of that bandwidth is consumed by sending pixmaps. As an example of such an application, we used sessions with the Mosaic World Wide Web navigator.

Medium graphics content These are generally applications that have graphical user interfaces and use some pixmaps for their displays, but do not perform most of their operations using pixmaps to draw to the screen. They have intermediate bandwidth requirements and most of the bandwidth is equally divided between text operations and pixmap operations. As a representative application of this type, we used FrameMaker, a popular word processing application.

Low graphics content These are generally applications that are text-based but have a minimal X windows interface. They have low bandwidth requirements and a large proportion of that bandwidth is consumed by text operations. As representative applications of this type we used xterms displaying various kinds of information, and the GNU editor, Emacs, used to edit a file.

3.1 The Raw Graphics Protocol Measurements

Tables 3.1 to 3.3 show the global results of the first set of measurements. Tables 3.5 to 3.7 show the volume of traffic divided by request type[1]. Note that most applications do not use many of the esoteric X operations, but they are included in the protocol for completeness.

The tables clearly show a decrease in the bandwidth required when moving from high graphics content to medium graphics content to low graphics content applica-

[1]Requests that were never used during the session are not listed.

Total Number of Requests	14902
Total Duration in Seconds	460
Total Number of Bytes Transferred	6706669
Average Request Size (Bytes)	450
Average Transfer Rate (bytes/sec)	14611
Maximum Request Size	68779
Average number of requests per sec	32.3
Maximum Number of Bytes in 1 sec period	101406

Table 3.1: Global results for raw graphics protocol (Mosaic session)

tions. They also show that most of this reduction comes from a reduction in the number of bytes transferred as pixmaps (mainly under the request DoBitBltCopy which copies pixmaps to the screen and CopyPlane1to8 which converts bitmaps to pixmaps and then copies them to the screen). As one moves to applications that have a lower graphics content, a higher percentage of the traffic arises from the text printing requests (such as TEGlyphBlt8, PGlyphBlt8CLPD and PGlyphBlt8). Even for those types of applications though, a major amount of bandwidth is consumed by DoBitBltCopy. The main reason for this is the way the cursor updates work. Before the cursor is printed, the area below it is copied into a pixmap. Then the cursor is painted. When the cursor finally moves out of that area, the old contents are copied from the pixmap to the screen again. This is handled by request DoBitBltCopy and causes considerable pixmap traffic even in the absence of graphics.

From the tables, the average bandwidth requirement is 15 Kbytes/sec or 120 Kbits/sec and the maximum bandwidth required for a one second period is 100 Kbytes/sec or 800 Kbits/sec.[2] Table 3.4 shows a summary of the important results for all three sessions.

[2]These values were calculated using the results from the Mosaic sessions at a BER of 10^{-3} since these displayed the worst case behavior.

Total Number of Requests	29675
Total Duration in Seconds	950
Total Number of Bytes Transferred	6079939
Average Request Size (Bytes)	204
Average Transfer Rate (bytes/sec)	6406
Maximum Request Size	10061
Average number of requests per sec	31.2
Maximum Number of Bytes in 1 sec period	44875

Table 3.2: Global results for raw graphics protocol (FrameMaker session)

Total Number of Requests	7448
Total Duration in Seconds	649
Total Number of Bytes Transferred	812601
Average Request Size (Bytes)	109
Average Transfer Rate (bytes/sec)	1254
Maximum Request Size	11214
Average number of requests per sec	11.4
Maximum Number of Bytes in 1 sec period	31497

Table 3.3: Global results for raw graphics protocol (Emacs session)

Session	Av. Transfer rate	1 sec. Peak	Req. per sec.
Mosaic	116888 (bits/sec)	811240 (bits)	32.3
FrameMaker	51248 (bits/sec)	359000 (bits)	31.2
Emacs	10032 (bits/sec)	251976 (bits)	11.4

Table 3.4: Global results for raw graphics protocol (Summary)

Request	Sum of Bytes	Percentage
ScreenInit	13	0.00
Push8	130830	1.95
FRSolidCopy	87501	1.30
FSolidSCopy	54860	0.82
FillBoxSolid	45870	0.68
FillBoxTile32	1834	0.03
DoBitBltCopy	5872887	87.57
DoBitBltXor	2608	0.04
CopyPlane1to8	2832	0.04
FST32Copy	15101	0.23
SegmentSS	3760	0.06
LineSS	17628	0.26
SegmentSS1Rect	50342	0.75
LineSS1Rect	7269	0.11
FPoly1RCopy	910	0.01
PolyPoint	1386	0.02
PFASCopy	507	0.01
PGlyphBlt8	267030	3.98
PGlyphBlt8CLPD	45818	0.68
TEGlyphBlt8	59901	0.89
UpdateClrmap	5565	0.08
KeyPressedCode	312	0.00
KeyReleasedCode	572	0.01
ButPressedCode	1599	0.02
ButReleasedCode	1599	0.02
MotionNotifyCode	28135	0.42

Table 3.5: Per-request results for raw graphics protocol (Mosaic session)

Request	Sum of Bytes	Percentage
ScreenInit	13	0.00
Push8	284172	4.67
FillRTS32	99676	1.64
FillRTSU	271010	4.46
FRSolidCopy	54324	0.89
FRSolidXor	1129	0.02
FSolidSCopy	12882	0.21
FillBoxSolid	45638	0.75
FillBoxTile32	3063	0.05
DoBitBltCopy	3482444	57.28
DoBitBltXor	8240	0.14
CopyPlane1to8	1042435	17.15
FST32Copy	18697	0.31
SegmentSS	35158	0.58
LineSS	19978	0.33
SegmentSD	374596	6.16
SegmentSS1Rect	75782	1.25
LineSS1Rect	17652	0.29
FPoly1RCopy	1024	0.02
PolyPoint	67248	1.11
PGlyphBlt8	40437	0.67
PGlyphBlt8CLPD	10057	0.17
TEGlyphBlt8	16970	0.28
UpdateClrmap	933	0.02
KeyPressedCode	18057	0.30
KeyReleasedCode	18174	0.30
ButPressedCode	767	0.01
ButReleasedCode	767	0.01
MotionNotifyCode	58616	0.96

Table 3.6: Per-request results for raw graphics protocol (FrameMaker session)

Request	Sum of Bytes	Percentage
ScreenInit	402	0.05
Push8	53130	6.54
FRSolidCopy	1014	0.12
FSolidSCopy	9615	1.18
FillBoxSolid	7609	0.94
FillBoxTile32	1363	0.17
DoBitBltCopy	484273	59.60
CopyPlane1to8	1634	0.20
Stipple32FS	2426	0.30
OpaqueSt32FS	27980	3.44
SegmentSS	3252	0.40
LineSS	6154	0.76
SegmentSS1Rect	7932	0.98
LineSS1Rect	33044	4.07
FPoly1RCopy	390	0.05
PGlyphBlt8	12949	1.59
PGlyphBlt8CLPD	15814	1.95
TEGlyphBlt8	110311	13.58
UpdateClrmap	885	0.11
KeyPressedCode	12181	1.50
KeyReleasedCode	12792	1.57
ButPressedCode	130	0.02
ButReleasedCode	130	0.02
MotionNotifyCode	7191	0.88

Table 3.7: Per-request results for raw graphics protocol (Emacs session)

Total Number of Requests	24317
Total Duration in Seconds	998
Total Number of Bytes Transferred	4738738
Average Request Size (Bytes)	194
Average Transfer Rate (bytes/sec)	4752
Maximum Request Size	38101
Maximum Number of Bytes in 1 sec period	40216
Total number of requests (not including acks)	11918
Total Reed-Solomon encoding overhead (%)	17.982656

Table 3.8: Global ECC results for full graphics protocol (Mosaic session)

3.2 The Final Protocol Measurements

3.2.1 The Effect of the Error Correction Code

Tables 3.8 to 3.10 show the same information as tables 3.1 to 3.3 but are updated to reflect the results obtained for the full protocol. All of the measurements were taken with a burst mode noise source and a BER of 10^{-3}. The tables have an additional line that states the percentage of overhead due to the Reed-Solomon error correction code. The first thing to note is that this overhead is substantially below the theoretical 66% because most of the data is not encoded. The second thing to note is that as one moves to applications that have a lower graphics content, the percentage of traffic that is due to unencoded data decreases, and therefore, the effective overhead increases. The maximum overhead occurs during the Emacs session which has about 46% overhead.

3.2.2 The Effect of Retransmissions

Tables 3.11 to 3.13 show information relevant to the retransmission overheads. All measurements were taken with a burst-mode noise system at a BER of 10^{-3}. Similar measurements were made at lower BERs but the results are not included here. All measurements at lower BERs showed reduced retransmission overhead. The overhead

Total Number of Requests	39009
Total Duration in Seconds	1462
Total Number of Bytes Transferred	5120239
Average Request Size (Bytes)	131
Average Transfer Rate (bytes/sec)	3504
Maximum Request Size	20423
Maximum Number of Bytes in 1 sec period	26110
Total number of requests (not including acks)	21957
Total Reed-Solomon encoding overhead (%)	23.557444

Table 3.9: Global ECC results for full graphics protocol (FrameMaker session)

Total Number of Requests	15866
Total Duration in Seconds	862
Total Number of Bytes Transferred	1123082
Average Request Size (Bytes)	70
Average Transfer Rate (bytes/sec)	1304
Maximum Request Size	18555
Maximum Number of Bytes in 1 sec period	18795
Total number of requests (not including acks)	8457
Total Reed-Solomon encoding overhead (%)	45.746915

Table 3.10: Global ECC results for full graphics protocol (Emacs session)

Total Number of bytes transferred	4397493
Total Number of bytes due to timeouts	68250
Timeout traffic percentage	1.55 %
Total number of bytes due to successful requests	4289259
Total number of bytes due to retransmissions	108234
Retransmission overhead (percentage)	2.52 %
Total number of messages	12453
Total number of requests	10154
Total number of retransmissions	24
Percentage of requests due to retransmissions	0.235803 %
Average number of bytes per request	433

Table 3.11: Global retransmission results for full graphics protocol (Mosaic session)

due to timeout traffic is also listed. The main thing to note is that both the timeout traffic and the retransmission traffic are relatively small even at a BER of 10^{-3} (which is the highest BER that the system was designed to handle). Another thing to note is an interesting deviation in the results of the Emacs session. Because the user spent some amount of time reading pieces of text, the X-server would remain idle and a large number of timeouts would occur. This explains the disproportionate percentage of timeout traffic for that session.

3.2.3 The Effect of Noise on the Final Display

Figure 3-1 shows the display on the cycle server at a particular instant in time. This is unaffected by noise. For comparison, Figure 3-2 shows the corresponding display on the terminal with some of the unencoded data (such as the main color image) corrupted by noise[3]. The system was operating at a BER of 10^{-3} at the time. Obviously, the display is very clear at such error rates even in the presence of burst-mode noise[4]. Because pixmaps are now transmitted as 8 bits per pixel, long

[3]Note that the missing cursor is an artifact of the method that was used to obtain the screen-shots and is not in any way related to the effects of noise.

[4]In fact, there is a visible error caused by a large burst of noise - try to find it.

Total Number of bytes transferred	4323324
Total Number of bytes due to timeouts	53910
Timeout traffic percentage	1.25 %
Total number of bytes due to successful requests	4305514
Total number of bytes due to retransmissions	17810
Retransmission overhead (percentage)	0.41 %
Total number of messages	17049
Total number of requests	15220
Total number of retransmissions	32
Percentage of requests due to retransmissions	0.209809 %
Average number of bytes per request	284

Table 3.12: Global retransmission results for full graphics protocol (FrameMaker session)

Total Number of bytes transferred	757052
Total Number of bytes due to timeouts	72720
Timeout traffic percentage	9.61 %
Total number of bytes due to successful requests	743987
Total number of bytes due to retransmissions	13065
Retransmission overhead (percentage)	1.76 %
Total number of messages	7407
Total number of requests	4972
Total number of retransmissions	11
Percentage of requests due to retransmissions	0.220751 %
Average number of bytes per request	152

Table 3.13: Global retransmission results for full graphics protocol (Emacs session)

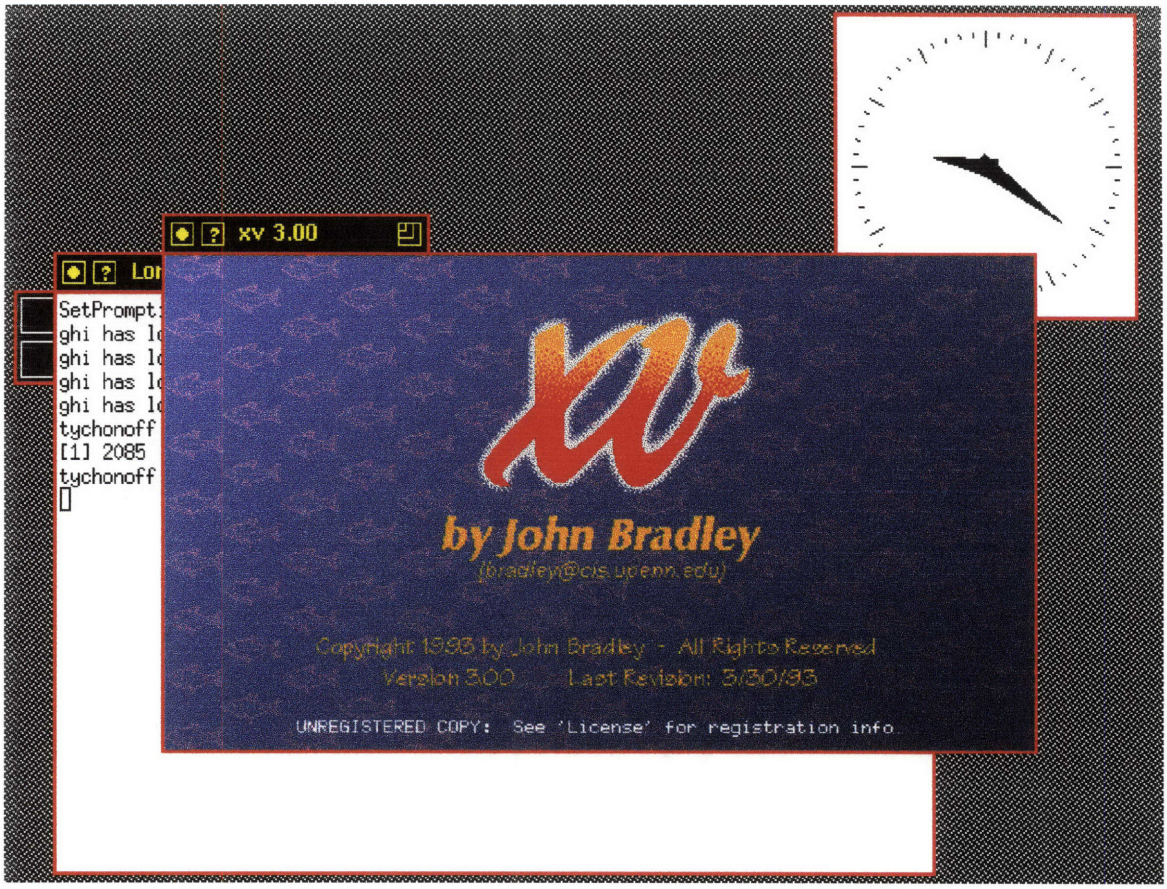

Figure 3-1: The Uncorrupted Display.

bursts of errors that affect consecutive bits now affect 8 times less pixels than in the monochrome case. In this respect, the system is more immune to noise than protocols that are based on monochrome displays.

3.2.4 Overall Performance

As it was stated at the beginning of this chapter, the response rate in the final system was not sufficient to make accurate bandwidth measurements. Using the results of subsections 3.1 and 3.2, we can extrapolate to estimate what the bandwidth would be if the response time was the same as that of the raw graphics protocol. We used the equation below to calculate the effective bandwidth :

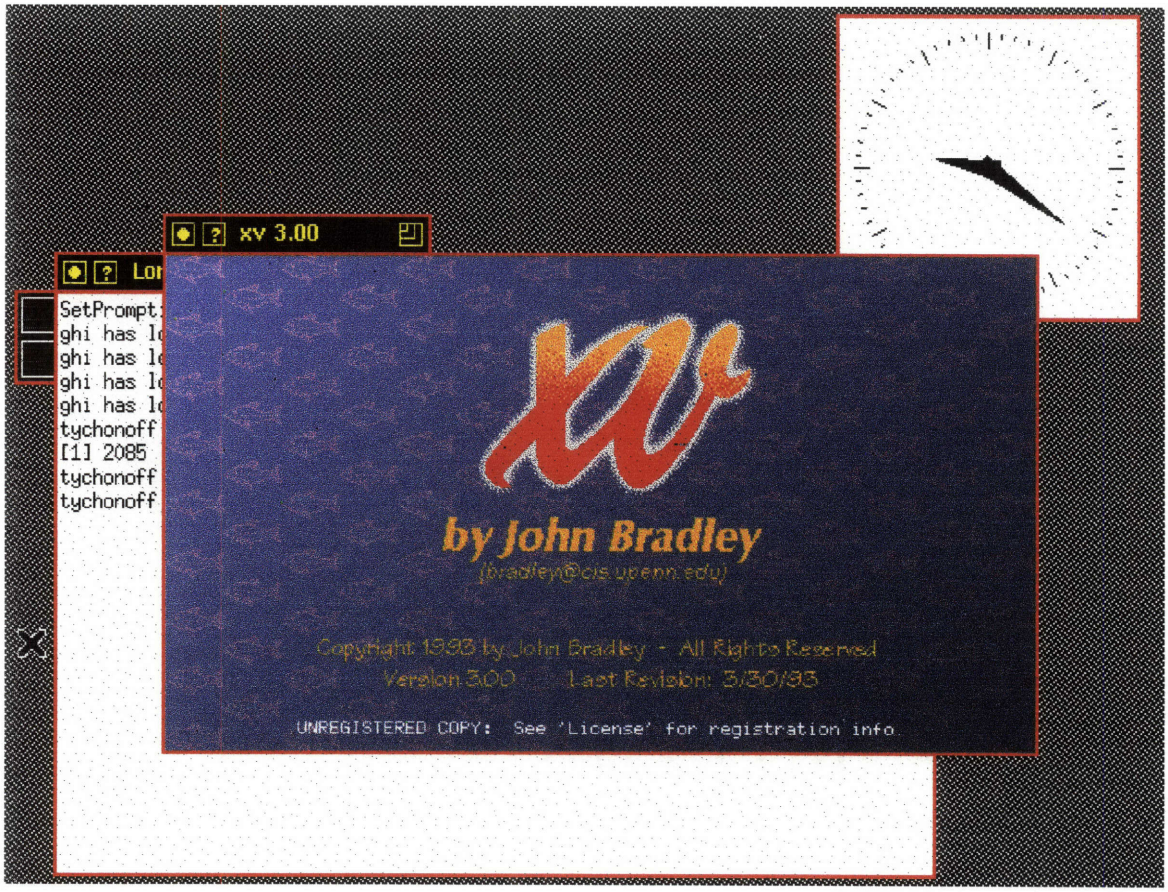

Figure 3-2: The Corrupted Display.

$$B_{eff} = (B_{raw} \times (1 + ECC_{ohd})) \times (1 + T_{ohd} + R_{ohd})$$

where

B_{eff} is the effective bandwidth requirement

B_{raw} is the corresponding bandwidth requirement for the raw protocol

ECC_{ohd} is the overhead due to the Reed-Solomon code

T_{ohd} is the overhead due to timeout traffic [5]

R_{ohd} is the overhead due to retransmissions

Using the above equation, we found the average bandwidth requirement to be 17Kbytes/sec or 140Kbits/sec. Similarly, the maximum bandwidth required to satisfy a one-second peak was calculated to be 970Kbits/sec.[6]

[5]Timeout traffic is the traffic caused by acknowledgments sent because the terminal did not receive any requests from the X server for a while. In this case, it sends the last acknowledgment again assuming that the first one was lost. Such acknowledgments are redundant a lot of the time and are considered to be overhead.

[6]These values were calculated using the results from the Mosaic sessions at a BER of 10^{-3} since these displayed the worst case behavior.

Chapter 4

Conclusions

This chapter focuses on what we have learned from our emulator environment and the measurements it allowed us to make. It describes how well the protocol performs within the original design specifications as well as most of the problems we know of, and some protocol properties that might help us design the terminal hardware.

4.1 Evaluation of the Protocol

One of the most important characteristics of the protocol that has not been discussed so far is its flexibility. The protocol does not define standard formats for the requests - only the headers and the two-level encoding scheme. The format of each individual request is completely flexible as long as all control information that needs to be protected by ECC is placed before all other information that needs no protection. While this was not originally one of our design targets, we soon realized that it was a particularly useful way of working around the disparate structures of the drawing routines. The structural variation was caused by the fact that those routines were heavily optimized, each one for a different task. Furthermore, this flexibility allows substantial room for improvement. To begin with, as better ways of performing current tasks are discovered, we can change the individual requests affected without

having to change anything else. Secondly, the fact that the X-server itself is so expandable would not be very useful if our protocol was not flexible. If that was the case, we would have to redo the partitioning and transfer of code, for every modification to the X-server.

One of the most important design goals was to keep the required bandwidth low. The protocol was designed with a 1Mbit/sec radio link in mind operating over a very short range. Given that rate, the average bandwidth requirements are easily satisfied. The peak rates, however, approach the available bandwidth. This means that generally, the system will operate very much within its capabilities and the bandwidth of the radio link is not likely to be a bottleneck. On the other hand, it points to the fact that when large transfers occur, latencies caused by the time it takes to get the data through will be on the order of a second. These latencies will slow the response time of the system. Hopefully, this will not happen often enough to be annoying to the user. Unfortunately, our emulator environment could not be easily modified to allow us to see how latencies would affect response time.

The second goal was to maintain the low power properties described in section 2.2.1. Generally, if any work could be performed on the cycle server without incurring a large bandwidth penalty, we opted to partition it out to the cycle server. There are some notable exceptions, but these are so rare under normal use that the overall overhead in computation would be small in comparison[1]. We also tried to keep the size of both the code and the memory required for various operations small. The code itself is 279 Kbytes when compiled for the SPARC processor under UNIX. This figure is somewhat misleading because it includes the code that handles the X-interface of the emulator, the error correction code and CRC code[2], none of which will be included

[1]Drawing arcs is one of these exceptions. It requires geometric mathematical functions which we decided not to build as a table but rather to have as code. Building tables would be prohibitively expensive in terms of required memory and the requests are so infrequent that it is not a favorable tradeoff.

[2]Both error correction and CRC will be performed in hardware – in software they took about 10Kbytes of code.

in the final program, but it does not include the operating system facilities which will be part of the final program. Since no state other than the fonts transcends request boundaries, the major amount of global data memory is used for the font cache. At its current size of 6 entries and an average font size of 8 Kbytes, the whole font cache would take about 48 Kbytes. The only other large portion of memory that might be required would be a buffer to hold unencoded data in case of a partial retransmission (ACK_RETX_PART). The maximum request size is set at 300 Kbytes, so at most, this would be the amount of memory required. Finally, some minimal amount of memory will be required to handle the process stacks. Not counting operating system services, the stack only grows to a depth of seven so the stack requirements are actually relatively small. Finally, a random pool of memory is required for short-lived memory allocation such as pixmaps and regions. Generally such elements are no bigger than 2 Kbytes and are usually much smaller than that. All in all, about 1 MByte of RAM seems to be sufficient for all uses other than the frame buffer. See section 4.3 for a suggested layout to save power.

The final target was reliability in the presence of noise. Unfortunately, the protocol needs some improvement in this particular area. Table 4.1 shows the nature of the noise model we used. It displays the distribution of the number of errors suffered by each packet for 2×10^6 packets at a BER of 10^{-3}. As the reader can see, a significant portion of them have more than the three errors that can be detected by the protocol. The probability that any packet will have undetectable errors is then 2.20×10^{-4} at that BER. The probability that such an error, undetectable by the ECC, will also escape the CRC is 2^{-16} or 1.53×10^{-5}[14]. The overall probability that an encoded packet will have an error that will not be detected (and therefore no retransmission will occur) is 3.36×10^{-9}. From the extrapolated bandwidth and the encoding overhead we can calculate how many packets are being transmitted per second:

$$P = B_{eff} \times ECC_{ohd}/((15 - 9) \times 4)$$

Number of errors	Number of instances
0	1990327
1	8228
2	612
3	393
4	208
5	82
6	79
7	47
8	7
9	7
10	6
11	3
12	1
13	0
14	0
15	0

Table 4.1: Noise analysis for the burst mode noise generator.

where P is the number of packets per second, B_{eff} is the average bandwidth in bits per second, 15 is the length of the encoded word in symbols (nibbles) and 9 is the number of encoded symbols. This comes out to be 1050 packets per second. The mean time between failures because of this mode of error then is over 280000 seconds or close to 80 hours.

There is also another mode of failure. Despite the fact that the headers are covered by CRC, they are particularly sensitive to undetectable errors since the CRC cannot be calculated until all the packets have been decoded. If the header is corrupt, we have no idea how many packets to decode and the CRC is useless. That means that if we receive an undetectable error in the header, it is likely that the system will fail before the CRC can even be calculated. The probability that at least one header packet will be received that is corrupt in any given second can be calculated by

$$P_f = 1 - (1 - P_p)^{2 \times H}$$

where

P_f is the probability that there is a failure (at least one undetectable error in a header packet)

P_p is the probability that a given packet will contain an undetectable error

H is the number of headers per second

2 is the number of packets per header

This comes out to 1.40×10^{-2} giving a mean time to failure on the order of minutes. However, the actual probability is substantially smaller due to some sanity checks performed on the values encoded in the header. The two main sanity checks involve the "`message size`" field which is four bytes long and denotes the total size of a request or retransmission, and the "`encoded size`" field which is two bytes long and denotes the size of the encoded portion of the request. The request size limit has been

set to 300000 bytes which is the size of a pixmap that covers the whole screen (the only way a request could ever get so large). Therefore, the first sanity check rejects all headers that have a "`message size`" field of more than 300000. The "`encoded size`" has to be a multiple of the packet size (in our case 15), so the second sanity check will reject as corrupt all headers that have an "`encoded size`" field that is not a multiple of the packet size. While it is hard to analytically determine the effect of these sanity checks, we have reason to believe that these checks significantly reduce the probability that a corrupt header passes undetected. Our observations support this belief. The observed mean time to failure without the sanity checks in place was on the order of minutes. When the sanity checks are in place, the observed mean time to failure is on the order of hours.

4.2 Limitations and Future Research

The protocol as described here, while complete and adequate by most counts, is by no means finished. There are a number of things that we would like to improve upon.

By far the most important is reliability. We consider the MTBF to be too high at a BER of 10^{-3}. In order to improve upon this we are considering the addition of a CRC for just the header, as well as replacing the 16-bit CRC by a 32 CRC, or 2 CRCs of 16 bits. Those two modifications alone would raise the MTBF to the order of years giving much more reliable operation.

Another improvement would be to extend the protocol to handle the painting of the cursor in a more intelligent way. We could, for example, arrange a special request for this, that copies the area under the cursor to a local pixmap (on the terminal) before the cursor is drawn, and then restores it from there. This would cause most of the pixmap traffic in text-based applications to disappear, thus reducing bandwidth requirements. It would also cause some reduction in bandwidth requirements in high graphics content applications that make heavy use of the pointer input device (mouse

or pen) and thus cause lots of cursor movement.

Finally, during our measurement sessions, we noticed that the protocol can cause traffic even when nothing gets drawn to the display. The main reason for this is elements (usually text) being "drawn" outside the clip region. While the display never gets changed for such a request, the routines that perform the normal drawing operations are still called so they create protocol traffic. We would like to change the individual requests involved so that this does not happen.

4.3 Suggestions for the Hardware

In developing the protocol, we gained a much better understanding of what the terminal hardware would need to do and thus came up with a number of possible optimizations for hardware.

To begin with, most operations on the terminal side involve painting scan-line segments. This means that sequential pixels are being accessed most of the time. Most frame buffers are arranged so that consecutive pixels are mapped to consecutive memory addresses. This has a number of important consequences:

Address Generators Rather than perform these indexing operations on the general purpose ALU using standard registers, it would make the operations much faster and make them consume less power if we created an on-chip address generator. To begin with, an address generator can perform a full indexing operation in one clock cycle. At the same time, the main ALU would be available for other operations. While this would increase the silicon area consumed by the CPU, the additional speed will allow operation at a reduced clock frequency that will allow a voltage scaling technique to be used to reduce the power consumption. Also, with the exception of floating point operations, indexing operations are the only type of operations that require multiplication. Since in indexing operations the multiplier is always constant (the width of the display in pixels), we

could hardwire that operation into the address generators and avoid including multiplication hardware in the ALU[3]. Finally, when not needed, the address generator can be shut down and so would consume negligible power.

Wider paths to the Frame Buffer By doubling the path to the frame buffer, one doubles the capacitance being switched but can halve the clock cycle. Thus, the capacitance switched per unit time remains constant. However, the slower clock cycle now allows the use of a voltage scaling technique at the same time, which reduces power dramatically. The fact that most accesses are sequential allows such wide busses to be used efficiently. In any case, the portions of the bus that should not be written to can be shut down[9, 10] thus preventing wasted power.

Another interesting optimization possibility arises from the use of ECC and CRC codes. The ECC and CRC operations are the type of operations that would be a lot faster (and a lot more economical in terms of switching activity) if performed on custom hardware. They are also completely orthogonal to the actual requests of the protocol, so they can be cast into hardware without affecting the flexibility of the protocol. As long as the same noise model is employed, there is no need to change the ECC or CRC operations.

Furthermore, an interesting possibility arises in the way memory is used. Because of the large memory size, it is very likely that DRAM would be used in the final hardware. However, DRAM is expensive in terms of power because it needs to be refreshed constantly. Most of the memory usage on the terminal is particularly transient. In fact most data that is stored, other than the code, is not needed for longer than a request. One could arrange memory layout so that any unused memory remains as a large continuous block. By keeping pointers to the beginning and end of this block, it would then be possible to avoid refreshing that block thus saving a substantial amount of power. Another possibility would be to avoid refreshing data

[3]Floating point operations are rare so we can afford to do them in software – they are mostly used for calculating sines and cosines for drawing ellipses and chords.

that only lasts for one request, if the request can be serviced in less than the required refresh time of the DRAM. These two techniques can significantly reduce the amount of power expended for data storage.

Finally, most operations take place as large loops. This means that a low associativity cache with large sets would be particularly effective as an instruction cache. Data caches are less critical since most data that is taken from memory is only visited once (except for bitmaps used as tiles, and global variables). Thus, the data cache can be made with smaller sets and a somewhat larger associativity. Note that a data cache is of no use when accessing the frame buffer because all frame buffer locations are accessed at most once during the duration of a request. Both caches would reduce power consumption by

1. increasing performance thus allowing the use of voltage scaling techniques.

2. reducing off-chip traffic which tends to be orders of magnitude more expensive in terms of power because of the larger capacitances involved.

Appendix A

Unpacking and Building the Emulator Environment

On the back of this thesis are attached three diskettes that contain the source code and the executables for the emulation environment. The diskettes are formatted for UFS (UNIX File System) and can only be used on UNIX machines. The same source code and executables are available by anonymous ftp from "`rle-vlsi.mit.edu`" under "`/pub/ghi/project`".

A.1 Unpacking the Source Code for the Emulation Environment

Go to the directory under which you want to place the environment and make sure it is writable by your user ID ("`chmod 755 .`" should set the right mode – if you get an error it probably means that you do not have permission to change modes on that directory and you should contact your system administrator). Insert each diskette in order and type "`tar -xvf <floppy_device>`". If you do not know which floppy device to use, ask your system administrator. You should now have a set of three files called "`source1.tar.gz`" to "`source3.tar.gz`". Uncompress the files with "`gunzip`

`*.gz`". Then unpack the sources by running "`tar -xvf <filename>`" on each file. If all went well, you should now have a directory called "`Xserver`" which contains all the sources.

A.2 Obtaining The Source by Anonymous FTP

At the command prompt (assuming you have a UNIX type machine) type "`ftp rle-vlsi.mit.edu`". You will be asked for a login name – type "`anonymous`". Then you will be prompted for a password. Please type your full electronic mail address, e.g.. "`ghi@lcs.mit.edu`". Go to the right directory using the command "`cd /pub/ghi/project`". Set the transfer mode to binary by typing "`bin`". Retrieve the packed source file by issuing the command "`get source.tar.gz`". Move the file to the directory under which you wish to unpack the source and make sure it is writable by your user ID as above. Then uncompress the source file with "`gunzip source.tar.gz`". Finally, extract all the sources with "`tar -xvf source.tar`". If all went well, you should have the "`Xserver`" directory with all the sources.

A.3 Building the Source

Generally, you should not need to perform this step. The code has only been tested on SPARC workstations and contains a number of references that are specific to those machines. It is therefore unlikely that you will be able to successfully build the code for any other platforms. Furthermore, the code has only been tested with the SunOS 4.1 operating system. Use of any other operating system might cause the code not to compile properly. Having said the above, it is unlikely that you will need to compile the sources anyway, since we have provided pre-compiled binaries with the above distribution.

At any rate, if you still need to recompile the sources (e.g. because you made modifications to the code) you will need to do a bit more work. All directories are provided with "`Imakefiles`". You will need to run "`imake`" on them to produce Makefiles which you can then use to build the distribution. Note that the "`Imakefiles`" provided require some of the original templates that were distributed with the X11 R6 release. Read your manual page on "`imake`" for further information on how to go about making "`Makefiles`". Once you have the "`Makefiles`" simply type "`make`" in the "`Xserver`" directory to make the Xserver. Then go to "`Xserver/infopad`" and type "`make`" to make the terminal emulator.

Appendix B

Using the Emulator Environment

The first thing to note is that unless you have successfully built the emulator environment for another platform, you can only use it on SPARC machines running SunOS 4.1.

The emulator environment is designed to operate on two different machines. One machine emulates the cycle server and the other the terminal. There must be a network connecting the two that does not filter TCP/IP traffic. Two machines on the same Ethernet backbone are the best configuration.

The first step is to disable "`xdm`" on the machine that will emulate the cycle server. This is necessary since "`xdm`" starts it's own version of the X server which would prevent you from running the modified one. Since your setup might be specific to your site, contact your system administrator to help you disable "`xdm`" if it is running.

Having done that, log into the terminal emulator machine and go to the directory under which you unpacked the source (see appendix A). You will find the terminal emulator program (called "`monitor`" under the directory "`Xserver/bin`"). If you want to simulate the system with noise type "`setenv IPD_NOISE_RATE 17000`" [1].

[1] As a rough guide, the number you need to insert here is $17 \times (1/BER)$ were BER is the target noise rate.

This will initialize the noise injection system at a BER of 10^{-3}. Experiment with other numbers if you wish. Note that this is only a target rate – the actual rate might deviate from this value because of the randomness involved in creating the noise. The real (measured) noise rate will be printed once the terminal emulator initializes. After you have set the noise rate, you may start the emulator by typing "`Xserver/bin/monitor`". If you have set a reasonable noise rate, the program will run for a few minutes trying to initialize the noise injection system. If not, it will warn you about the noise rate being set to zero. After that, a window will pop up on the screen. This is the window that will display the terminal "screen". Moving the mouse in this window or pressing any of the buttons or keys while the cursor is in the emulator window, will cause the corresponding actions to be sent to the modified X server that will be running on the cycle server machine.

Once you have the window, it is time to start the modified X server. Before you start the program, find out the name of the machine that is running the terminal emulator by typing "`hostname`". Then type "`setenv IPD_HOST <name>`" on the cycle server machine, where name is the host name of the terminal emulator machine. This allows the modified X server to contact the terminal emulator on the right machine. Then you can start the X server by typing "`cd Xserver/bin; xinit`" in the directory in which the source was unpacked. [2]

If all went well, you will see the X server initialize a portion of the screen in the familiar tiled background. If you now send an expose event to the window on the terminal emulator you will see the screen reproduced in the window. Again, since the way you can send an event depends on your window manager settings, you might want to contact your system administrator for help. If all else fails, you can iconify and de-iconify the window[3]. You can use the system by typing in the window at the

[2]Do not put the bin directory in your path and then use "`xinit`" to start the X server. "`xinit`" will probably start your unmodified X server instead of our modified one.

[3]The screen is actually being drawn to memory all the time. For efficiency reasons, we do not copy the contents of the memory to the window until an expose event occurs.

terminal side as if you were on the real display.

To shut down the system, exit from all your applications running under the modified X server (including the window manager). Both the X server and the terminal emulator will exit automatically.

After you exit the system normally, the terminal emulator will leave behind two statistics files named "`/tmp/IPD_STATS_FILE`" and "`/tmp/IPD_NOISE_STATS_FILE`". You can run the analyzer programs on these files to obtained condensed statistics. Use "`analyze`" on "`IPD_STATS_FILE`" and "`retx`" on "`IPD_NOISE_STATS_FILE`". The output of the two analyzer programs is self explanatory.

Appendix C

The Request Types and Functions

Below we list and describe the function of all 64 requests that currently form the protocol. For the exact formats of each request, the reader is advised to consult the source code. The file "`Xserver/infopad/cfbrequests.h`" contains a listing of all the request identifiers and the files to which they correspond. For more information on the meaning of the various objects and X protocol operations refer to [11].

ScreenInit This notifies the terminal that the Xserver has initialized its screen. Currently this is not used but it is reserved for future use.

Push8 This uses a pixmap (usually a bitmap actually) as a stencil to paint a certain section of the screen to a given color. It is mainly used to paint the cursor.

FillOpStp32 Fill Opaque Stipple with a 32-bit wide stipple. This is an optimized version of the Fill Opaque Stipple request of the X-protocol.

FillRTS32 Fill Rectangle Transparent Stipple with a 32-bit wide stipple. The same as above but for transparent operations.

FillRTSU Fill Rectangle Transparent Stipple Unnatural. Transparent stipple operations for stipples of other sizes.

FRT32Copy Fill Rectangle Tile with a 32-bit wide tile using the copy X protocol operation.

FRT32General Same as above but for all operations other than copy.

T32FSCopy Tile with 32-bit tile using Fill Spans technique and the X protocol copy operation.

T32FSGeneral Same as above but for all operations other than copy.

FRSolidCopy Fill Rectangle Solid using the X protocol copy operation.

FRSolidXor Same as above for exclusive OR operation.

FRSolidGeneral Fill Rectangle Solid for all other operations.

FSolidSCopy Fill Solid using Spans technique and the X protocol copy operation.

FSolidSXor As above but for the exclusive OR operation.

FSolidSGeneral Fill Solid using Spans technique for all other operations.

FillBoxSolid Fill Box Solid.

FillBoxTile32 Fill Box with a Tile that is 32 bits wide.

DoBitBltCopy Copy a pixmap (or another portion of the screen) to the screen using the X protocol copy operation.

DoBitBltXor Same as above but for the exclusive OR operation.

DoBitBltOr Same as above but for the OR operation.

DoBitBltGeneral Copy a pixmap (or another portion of the screen) to the screen for all other operations.

SetSpans Set a portion of the screen to a known value using the Spans technique.

CopyPlane1to8 Expand a bitmap to an 8-bit per pixel pixmap and copy to the screen.

FBTOCopy Fill Box Tile Odd (not 32-bit wide tile) using the X protocol copy operation.

FBTOGeneral Same as above but for all other operations.

FSTOCopy Tile with Odd (not 32-bit wide) tile using the Spans technique and the X protocol copy operation.

FSTOGeneral Same as above but for all other operations.

FBT32Copy Fill Box Tile with a 32-bit wide tile for the X protocol copy operation.

FBT32General Same as above but for all other operations.

FST32Copy Fill with a 32-bit wide Tile using the Spans technique and the X protocol copy operation.

FST32General Same as above but for all other operations.

UnaturalSFS8 Unnatural (not a standard size) Stipple using Fill Spans technique optimized for 8-bit per pixel displays.

UnaturalSFS Same as above for all other displays.

Stipple32FS Stipple with 32-bit wide stipple using the Fill Spans technique.

OpaqueSt32FS Same as above for opaque stipples.

SegmentSS Draw a line segment.

LineSS Draw a line.

SegmentSD Draw a line segment with dashed lines.

LineSD Draw a line with dashed lines.

SegmentSS1Rect Draw a line segment bounded by rectangle.

LineSS1Rect Draw a line bounded by rectangle.

FPoly1RCopy Draw a polygon using the X protocol copy operation.

FPoly1RGeneral Same as above for all other operations.

ZeroArcCopy Draw a Zero width (one pixel wide) arc using the X protocol copy operation.

ZeroArcXor Same as above for X protocol exclusive OR operation.

ZeroArcGeneral Same as above for all other operations.

PolyPoint Draw a number of points.

PFASCopy Fill multiple Arcs Solid using the X protocol copy operation.

PFASGeneral Same as above for all other operations.

TEGlyphBlt Draw Image text

PGlyphBlt8 Draw normal text optimized for 8 bit displays if it does not need to be clipped.

PGlyphBlt8CLPD Draw normal text optimized for 8 bit displays if it needs to be clipped.

PGlyphRop8 Draw normal text for 8 bit displays for all X operations other than copy if it does not need to be clipped.

PGlyphRop8CLPD Draw normal text for 8 bit displays for all X operations other than copy if it needs to be clipped.

TEGlyphBlt8 Draw Image Text optimized for 8-bit displays.

UpdateClrmap Update one or more colormap cells.

KeyPressedCode Notify X server that the user has pressed a key.

KeyReleasedCode Notify X server that the user has released a key.

ButPressedCode Notify X server that the user has pressed a button.

ButReleasedCode Notify X server that the user has released a button.

MotionNotifyCode Notify X server that the user has moved the main pointer.

ACK_RETX_COMP Acknowledge given ID requesting full retransmission.

ACK_RETX_PART Acknowledge given ID requesting partial retransmission.

ACK_OK Acknowledge proper receipt of given ID.

Bibliography

[1] S. Angebranndt, et al. *Definition of the Porting Layer for the X v11 Sample Server.* April 1994.

[2] A. Chandrakasan. *Low Power CMOS Design.* PhD. Thesis, UC Berkeley, 1994.

[3] A. Chandrakasan, et al. *A Low-Power Chipset for a Portable Multimedia I/O Terminal* Journal of Solid State Circuits, December 1994.

[4] D. Dobberpuhl, et al. *A 200 Mhz 64-bit Dual-Issue CMOS Microprocessor.* IEEE Journal of Solid State Circuits, pp106-107, 1992.

[5] S. Gildea. *X Window System, Version 11, Release 6 Release Notes.* March 1994.

[6] H. Imai. *Essentials of Error-Control Coding Techniques.* Academic Press, 1990.

[7] E. Israel, E. Fortune. *The X Window System Server* Digital Press, 1992.

[8] J. F. MacWilliams. *The Theory of Error Correcting Codes.* North-Holland Pub., 1978.

[9] J. McCormack. *Writing Fast X Servers for Dumb Color Frame Buffers.* WRL Research Report 91/1.

[10] J. McCormack. *Smart Code, Stupid Memory: A Fast Xserver for a Dumb Color Frame Buffer.* WRL Technical Note TN-9.

[11] R. W. Scheifler. *X Window System Protocol, X Consortium Standard, X Version 11, Release 6.* March 1994.

[12] Symbolics Documentation. *Networks and I/O* February 1984.

[13] M. Weiser *Some Computer Science Issues in Ubiquitous Computing.* Communications of the ACM, July 1993.

[14] S. B. Wicker. *Error Control Systems for Digital Communication and Storage.* Prentice Hall, 1995.