microMPX: A UNIX based window manager for microcomputers

J. Scott Mulligan

The University of Montana

Recommended Citation
https://scholarworks.umt.edu/etd/5508
COPYRIGHT ACT OF 1976

THIS IS AN UNPUBLISHED MANUSCRIPT IN WHICH COPYRIGHT SUB-SISTS. ANY FURTHER REPRINTING OF ITS CONTENTS MUST BE APPROVED BY THE AUTHOR.

MANSFIELD LIBRARY
UNIVERSITY OF MONTANA
DATE: 1986
μMPX: A UNIX Based Window Manager for Microcomputers.

By

J. Scott Mulligan

B.A., University of Montana, 1982

Presented in partial fulfillment of the requirements
for the degree of
Master of Science
University of Montana
1986

Approved by

[Signatures]

Chairman, Board of Examiners

Dean, Graduate School

[Signature]

Dec. 11, 1986
Improving the ways that people utilize computers has been a topic of interest within the computing community in recent years. One improvement is the multiwindow environment which allows the user to manipulate several windows, each providing a separate terminal session on a multiprocessing computer.

μMPX is both a communications protocol and a window server committed to supplying a multiwindow environment to users with microcomputers. By using microcomputers as client machines in this system, the multiwindow environment becomes more affordable and available than current commercial offerings.

This document describes the development of the μMPX protocol as well as the design and implementation of a μMPX window server under the UNIX operating system.

*UNIX is a trademark of AT&T
Table of Contents

Abstract ii
Table of Contents iii
List of Figures v
Acknowledgements vi
1. Project Formulation 1
   1.1. Background 1
   1.2. An Introduction to the Project 3
   1.3. Proposed Work Plan 5
2. A Protocol for Multiplexed Communications 8
   2.1. The User Session 8
   2.2. List of Protocol Requirements 11
   2.3. Hardware Constraints 12
   2.4. Software Constraints 14
3. Formulating the Communications Protocol 17
   3.1. Building a Foundation 17
   3.2. Basic Packet Formats 17
      3.2.1. The Header Byte 18
   3.3. Error Detection and Correction 19
      3.3.1. Error Detection 19
      3.3.2. Error Correction 20
         3.3.2.1. The ACK Packet 20
         3.3.2.2. Negative Acknowledgements 22
         3.3.2.3. Sequencing Errors 23
   3.4. Formal Packet Types and Formats 24
      3.4.1. Packet Structures 26
   3.5. Startup Handshaking Sequence 27
   3.6. Flow Control and Hardware Buffer Overflow 27
   3.7. Final Amendments to the Protocol 28
      3.7.1. New Packet Structures 30
4. Designing and Implementing the Server Process 31
   4.1. An Initial Design 31
      4.1.1. The Send Process 32
      4.1.2. The Receive Process 35
      4.1.3. The Manager Process 36
      4.1.4. Flow Control 39
List of Figures

Figure 2-1: A Fully Encapsulated Packet 12
Figure 2-2: A Reasonable Solution 16
Figure 2-3: An Unreasonable Solution 16
Figure 3-1: Initial Header Byte 18
Figure 3-2: Header Byte with ACK bit 21
Figure 3-3: Final Header Byte Format 24
Acknowledgements

I would like to offer special thanks to the members of my thesis committee. The time they have spent advising me and proofing drafts could have been spent in more pleasant ways. Thank you for your gifts, Alden, Spencer and Dick.

My thanks, also, to all of those that have helped me along the way to where I am now. Parents, professors, instructors, friends and fellow students have all influenced my education and are all due this thanks; without them I would be a very different person.
Chapter 1

Project Formulation

1.1. Background

In recent years increased attention has been given to improving the ways in which humans interface with computers. An increase in non-technical users as well as a desire to increase user productivity has made this one of the most important issues in computer science.

One of the most widely developed improvements in this area is the multiwindow environment. Principally, multiwindow environments have been proven to increase productivity for non-technical users as well as sophisticated users. The basic premise of a multiwindow environment is to provide the user with the ability to simulate multiple terminals on a single video display while allowing that user to interactively switch his keyboard between those simulated terminals. Most commercially offered multiwindow systems use large high resolution bit-mapped graphic displays to present the various windows; allow overlapping of windows; permit the user to vary the size of the windows; provide graphics capabilities; and provide a mouse device to aid the user in opening, closing, selecting and resizing windows. In some cases color displays are even offered. In most cases, the hardware required to run these systems tends to be rather expensive.
Currently available window systems include Rob Pike's window system for the Teletype 5620 terminal, Jack Test's system for the MIT Nu machine, Sun's Sunview for Sun Workstations, and two systems that are not tightly coupled to proprietary hardware: the Andrew System, from Carnegie-Mellon; and the X Window System from MIT. Only the latter two window systems enjoy freedom from expensive hardware and no mandatory operating system modifications. Of these two, only the X Window System from MIT is widely available and in the public domain. The Andrew System is owned by IBM and available only to universities [Rosenthal, 1986].

So what of the potentials for the X Window System? At the time of this writing, the X Window System seems to have the greatest chance of becoming the de facto standard for all window systems. Its server processes require no modifications to the operating systems of the hosts they run on. Servers are commercially available for UNIX and VAX/VMS operating systems. A public domain server for UNIX is available for a minimal charge. Its communication protocol is in the public domain. It is very flexible, providing its users a wide range of functions.

Unfortunately, the X Window System has yet to fully prove itself to the computing community. It is not yet available on a wide range of client machines. This may be due to the fact that the X Window System provides a wide spectrum of functionality, thus requiring a considerable amount of effort to develop compatible client processes for various potential client machines.
1.2. An Introduction to the Project

One generally available resource for improving the the human/computer interface is the personal computer. These microprocessor-based machines have become more available, more powerful, and less expensive in recent years. While not providing as much flexibility or computing power as a graphics workstation, many of them do provide enough to support a windowing system with graphics support. It is on this premise that I suggest the development of a minimal window system specifically developed for use with microcomputers.

To be valuable, this window system should provide the user with a capability of opening several separate text windows. Full asynchronous communications should be supported to these windows, thus requiring the development of a communications protocol. Within the limits of the specific microcomputer both text and graphics terminals could be emulated by the client processes. Additionally, to provide future flexibility, a user should be able to execute processes on the server machine that may exploit specific features of the client microcomputer. The introduction of a simple library transport packet to the already needed communication protocol should accommodate this need easily. Most importantly, the entire system must be designed to execute as efficiently as possible.

By streamlining the functionality of this window system it is expected that a functional, usable and useful window system will be able to be designed and implemented to run efficiently on lower grade microcomputer hardware. This should make a better, more useful human/computer interface more attainable to
the public.

For my thesis project I propose the design and implementation of a window server protocol and a window server, as described above, that will run on any BSD4.2 UNIX compatible host. The server will be referred to as $\mu$MPX, indicating that it is a micro (window) multiplexer. The symbol $\mu$ stands for the mathematical symbol $\mu$ which is usually associated with the word micro. While the $\mu$MPX server will be designed to run on Berkeley compatible operating systems, an effort will be made to use only standard UNIX system calls with the hope of creating a server that will be easily ported to other versions of UNIX.

The $\mu$MPX protocol will be a packet driven, error-correcting protocol that can support all required communication functions over a standard RS-232 compatible serial line. The development of this protocol will be in conjunction with Sidney Wang, a graduate student at the University of Montana, who proposes to design and implement a client window system for the Apple Macintosh computer. We decided to use a packet driven protocol in order to provide error detection and error correction while minimizing computational overhead. Accurate data transmissions, we felt, are necessary when developing a windowing system in order to assure accurate results when opening and closing windows as well as terminating the window session. Transmission errors in these types of data would be catastrophic. Although other methods could be employed to accomplish error detection and error correction, we felt that a packet driven protocol would require a minimal amount of computational overhead. A packet driven protocol, while reducing computational overhead, may reduce transmission speeds. This tradeoff
seemed reasonable considering that microcomputers with limited computational abilities will be used as client machines. We chose to sacrifice potential transmission speeds for computational efficiency.

1.3. Proposed Work Plan

The first step to completing this project will be to define the basic foundation of the project; the communication protocol. To complete this first step I will be collaborating with Sidney Wang. Together, we will attempt to create a simple, efficient communications protocol that will transport data, allow specific remote procedure calls and remote library procedure calls, and maintain communication integrity through error detection. I expect that the design of this protocol will be the most difficult and time consuming aspect of the development of this project. The results of this design will strictly dictate the degree of success the remainder of the project will achieve.

Once an initial protocol design has been created, a preliminary design for the host service process will be made. From this point an incremental approach will be used to develop the window server. A minimal prototype, loosely based on this design, will be developed to test the communications protocol and to test basic design features of the server process. No attempt will be made to actually accomplish any windowing activities. Instead, this initial prototype will simply be used to transport various amounts of data in an attempt to test the multiple features of the communications protocol. Depending upon the results of this testing, the communications protocol may be refined or reworked and then tested
again until a suitable solution is found.

Once we have resolved the protocol issue, each of us will begin to refine and complete our own individual projects. For me, this will mean the development of a new prototype server process. As with the prototype that was developed to test the communications protocol, this prototype will not fully accomplish any windowing activities. Instead, this next prototype will attempt to resolve communication and queuing issues. Cycles of refinement will follow until proper solutions are established.

The next prototype will ignore all of the issues that were addressed by the first two prototypes. Management of subprocesses will be its thrust. Eventually it will be necessary for the server process to create shell processes, with appropriate environments, that will be associated with each window the user opens. This prototype will attempt to resolve all issues pertaining to the management of these subprocesses. Once again, cycles of refinement will follow until proper solutions are established.

Once the third prototype is complete, the final server process will be implemented using as much of the previous prototypes as possible. This phase of the project should proceed quickly since all of the major issues should already be resolved. Hopefully, the completion of this phase will coincide with the completion of Sidney Wang's project, allowing thorough testing of the basic window system.

Finally, the implementation of a minimal set of UNIX library routines will complete the final phase of this project. It will be through these routines that a UNIX programmer will be able to exploit Macintosh specific functions from his
UNIX programs. For example, a UNIX programmer utilizing this library will be able to sample the mouse input device on the Macintosh from his UNIX program.
Chapter 2

A Protocol for Multiplexed Communications

2.1. The User Session

To best understand what requirements this window system will impose upon its communication protocol, one needs to understand exactly how the user might utilize this window system. The following text will describe some interactions a user might go through when utilizing this product. In this way, the reader should gain a better understanding of how the client and server processes will need to interact as well as how various types of data will be employed by both processes.

The user will start his client process running on a microcomputer that is connected to a UNIX host machine. A standard RS-232 serial interface connection or a modem will accomplish this connection. Initially, the client process will be running in a non-windowing terminal emulation mode. This will provide the user the opportunity to login to the UNIX host machine. Once logged in, the user could opt to remain in this initial mode, merely using his microcomputer as a standard terminal. If the user decides to take advantage of the windowing capabilities at his disposal he will simply run the \( \mu \)MPX program residing on the UNIX host. Upon issuing the command to execute the \( \mu \)MPX program, the user's screen will blank and he will be allowed to open several individual windows on his screen. As each window is opened, a special message is sent from the client process to the
host process (μMPX) indicating that a new window has been opened and that a new shell, or command line interpreter process, should be started on the host for that window.

In each of the windows the user opens, a standard input prompt will appear just as if that window were a separate terminal. By selecting a window to be "current", the user may use his keyboard to enter data to that particular window. Typing on the keyboard will only affect the current window while still allowing the user to view incoming data on all open windows. Incoming data will be displayed on the appropriate window as it received. At high transmission speeds this should give the user a feeling that several things are being done at once.

Let's assume that the user has just opened his first window. Since there are no other windows open it follows that this first window must be his "current" window. Thus, anything the user types on his keyboard will be given as input to the shell process, running on the host computer, that is associated with this window. The user could use this window for editing files, executing programs or any other function he would normally accomplish using a terminal. Similarly, the user may open several more windows. Now, we will consider a hypothetical situation involving a user of this windowing system.

After having logged in to his host computer and having entered multiwindow mode, the user assesses the work he wishes to accomplish during this session. He quickly decides that he only has two major tasks he would like to accomplish. The first task involves the execution of a processor intensive program that creates a great deal of output to data files as well as providing continual status output to
the terminal. This program usually requires about 20 minutes of user time for execution. Unfortunately, this particular program cannot simply be run in the background or batch since its interactive design requires user input every few minutes, and the user input is generally based upon the status reports generated by the program itself.

The second task our hypothetical user wishes to accomplish involves the replacement of a sorting algorithm in a program he has written himself. After having written this particular program, a friend has pointed out a more efficient sort method for the type of data he is processing. This same friend had been gracious enough to electronically mail an implementation of the algorithm to our user; although it would need to be tailored for his specific application.

To begin, the user opens a window in the top half of his screen and begins his first program. With that process started, the user opens a second window within which he invokes an editor for the source file he wishes to modify. With this second window selected as his "current" window, the user removes the obsolete sorting code and begins to insert the new sorting code from memory. Glancing up, the user realizes that his first program requires some attention. After scanning the status reports generated by the program he selects the first window to be "current" and enters some appropriate data using his keyboard.

Having satisfied the first program for the time being, the user finds that he has lost his train of thought and cannot recall the next step of the new sorting method. Opening a third window, he invokes the system mail program in order to reread the note from his friend. With the actual algorithm in front of him the user
may continue updating his program, but reading through it again he finds that he
does not understand the final step of the algorithm and decides he must contact
his friend before continuing.

By opening a fourth window, the user is able to determine that his friend is
also logged in to the host computer and he is able to request an interactive “talk”
session with his friend. Through this session the two of them are able to
determine that the mailed algorithm contained a typographical error which was the
cause of confusion. It should be noted that during their entire discussion all
windows were still accessible to the user, allowing him to view the original mailed
algorithm, keep track of his processor intensive program, and discuss the problem
with his friend.

2.2. List of Protocol Requirements

By isolating the activities the window system will be responsible for, it can
be deduced that the window system will require its protocol to:

1. Transport user input data from a specific window on the
   microcomputer to the appropriate process on the UNIX system
2. Transport output data from UNIX processes to the appropriate window
   on the user’s microcomputer
3. Transport remote procedure calls specific to windowing operations
4. Transport remote library procedure calls that are specific to the client
   machine
5. Detect transmission errors
6. Recover from transmission errors
7. Operate at high transmission rates over serial lines without loss of data
   or hardware overflow problems

In addition, it would be desirable to accomplish all of these with a minimal
amount of overhead.
2.3. Hardware Constraints

When considering rudimentary factors that will shape the design of a computer communications protocol, it is important to consider the constraints imposed by the hardware that will be used. In this case, there are four major factors that need to be considered:

1. All data will travel over a serial data line.
2. Processor speeds of both the client and host machines must be adequate.
3. It may be desirable to use this windowing system at low baud rates.
4. Hardware input buffers may risk overflow at high baud rates.

The fact that all data will be traveling over a serial data line requires the implementation of fully encapsulated packets as shown in Figure 2-1.

|-----|-----|-----|--......--|-----|-----|-----|
|      |      |      |             |      |      |      |
|<-header bytes->|<data bytes>|<-trailer bytes->|
|      |      |      |             |      |      |      |
|-----|-----|-----|--......--|-----|-----|-----|

Figure 2-1: A Fully Encapsulated Packet

Fully encapsulated packets will be required in order to define the boundaries of individual packets in the case of lost data. In other words, by fully
encapsulating packets with header and trailer bytes, the communicating processes will be able to resynchronize their communications in the event that a data packet is corrupted through some transmission error.

The fact that the client machine may possess a processor of limited power and the fact that the host machine may be oversubscribed dictates an obvious requirement for efficiency. Similarly, the possible desire to operate this system at a low baud rate, as when using a modem, also dictates a need for an efficient, streamlined protocol. For this reason it will be important to minimize the number of bytes needed in the packet header by packing data into bit fields.

Since the overflow of hardware buffers is a generally a common problem at high baud rates it will be necessary to find a solution to this problem. Two potential solutions standout as likely candidates. First, it may be possible to allow the hardware or operating system to avoid this problem through some method of hardware handshaking. The use of modem control signals, for example, would provide a workable, if not plausible, solution to the problem. Second, it may be possible to incorporate a special handshaking packet into the communications protocol which would allow the client and server processes to detect the possibility of a hardware buffer overflow. Of these two solutions, the first would be most efficient but might restrict the possible contents of our packets. The second possibility, while solving the drawback of the first, would be difficult to implement on a timesharing host and poses many timing problems.
2.4. Software Constraints

It is also important to consider the limitations that will be imposed on the protocol due to software considerations. The major detail to be weighed here is the fact that our primary unit of data is a standard eight bit byte. While this may seem insignificant at first, it will effect the design of the protocol by dictating the maximum amount of data that can be transmitted in a single byte.

It is common for packet headers to include a field within them that indicates the length of the data field. When deciding what the maximum length of the data field should be, one needs to consider the number of bits to be allocated to the length field. An eight bit field would allow a maximum value of 255. If a larger value is required, two or more bytes would be required to contain the length field. Conversely, if a smaller value is needed, the length field could require seven bits or less, leaving part of the byte free for some other use.

In cases where field boundaries fall within a byte, rather than on a byte boundary, it will be necessary for the client and server processes to pack fields into bytes to be transmitted, and then unpack the fields from the corresponding bytes on receipt. Fields that are completely contained within the boundaries of a given byte are easily extracted through bit masking and bit shifting techniques. However, when fields cross the boundaries of the bytes that contain them, it may be necessary to extract the separate parts of the field and then reassemble those parts into a structure that is meaningful to the client or server process. This reassembly can introduce avoidable overhead in some cases. Thus, it is truly important to consider the exact placement of data fields when designing an
efficient communications protocol.

If we assume, for example, that there are seven fields that require seven, five, four, three, three, one and one bits respectively, Figure 2-2 and Figure 2-3 illustrate extreme examples of reasonable and unreasonable solutions. Figure 2-2 shows an example where all field boundaries fall within or on the boundaries of a byte. Figure 2-3, on the other hand shows the fields arranged in a haphazard fashion requiring the fields to be split across bytes when sent, and reassembled when received.
Figure 2-2: A Reasonable Solution

Figure 2-3: An Unreasonable Solution
Chapter 3

Formulating the Communications Protocol

3.1. Building a Foundation

The basis for this project is the communication protocol through which the client and server processes will communicate. Since all other aspects of the project will rely upon the structure and design of the communication protocol, the protocol is the most prominent part. With this distinction in mind, Sidney Wang and I resolved to spend a significant amount of effort on this stage of our projects. In this way, we hoped to develop a useful, efficient protocol that could be expanded if need dictated. With this as our goal we aspired to create a communications protocol that would satisfy the requirements outlined in the previous chapter.

3.2. Basic Packet Formats

As stated in the previous chapter, fully encapsulated packets will be used due to the fact that all data must travel across a single asynchronous serial data line and also for the purpose of achieving efficient error detection and correction within the protocol. Assuming a need for several different packet types, Sidney and I determined that a special "header" byte will begin each packet. Within this byte, information will be kept to indicate the type of the packet as well as other characteristics of the packet. In order to fully encapsulate the packet we decided
that two “trailer” bytes, each containing a unique eight bit value, would mark the end of each packet.

3.2.1. The Header Byte

After examining our needs carefully, we concluded that all packets would either be associated with one of seven possible text windows or, else would be maintenance packets used to relay commands such as needed for opening or closing a text window. Thus we decided that three bits of the header byte would be reserved for indicating which window the packet is destined for.

Deciding that eight possible packet types should fulfill all of our potential needs, the initial header byte could be designed as illustrated in Figure 3-1.

![Figure 3-1: Initial Header Byte]

At this point, two bits remain unused in the header byte. One of these, we determined, would be used as a sequencer bit and the other as an
acknowledgement bit. Both of these play an important role in matters of error detection and error correction. A detailed explanation of these two bit fields follows in the next section.

3.3. Error Detection and Correction

One of our chief concerns in the development of this protocol was the issue of error detection and error correction. It was our goal that the protocol must attempt to guarantee uncorrupt transmissions. We felt that this was of particular importance in the cases where the client process was issuing a maintenance command such as is needed for opening or closing a window. The result of such a transmission becoming garbled could be catastrophic.

3.3.1. Error Detection

There are two ways in which a packet could become corrupt. At some point during the transmission of a packet one or more bytes could be dropped. Similarly, bytes could become corrupt during the transmission resulting in changed data.

In order to solve the problem of lost data, we chose to rely upon the fact that all packets will be fully encapsulated. Hence, by predetermining the packet size, either by packet type or a specific length field in the packet, the recipient of the packet will know when to expect the encapsulating trailer bytes. If the trailer bytes are missing, the recipient can assume that a transmission error has occurred and must take steps to resynchronize with the sender by discarding all incoming bytes until such time that two trailer bytes are received.
To resolve the latter issue we considered several options. Finally we determined, that in order to minimize computational overhead, the best solution was to incorporate a checksum scheme into our protocol design. We felt that minimizing computational overhead was a major goal since we expected microcomputers to serve as client machines in the window system. Eventually, we decided that one entire byte in each packet would be reserved for a checksum value and that value would reflect a checksum based on all packet bytes except bytes used for the checksum itself or trailer bytes used for packet synchronization. We also agreed that a simple algorithm based on modulo arithmetic would fulfill our needs for generating an eight bit checksum value. In this way, all packets could be verified with a minimal amount of overhead.

3.3.2. Error Correction

3.3.2.1. The ACK Packet

Since transmission errors should occur very infrequently and because client machines may possess meager processing abilities, we decided that retransmission would best meet our needs for error correction. To accomplish this retransmission scheme, the sender of a packet must not continue to send packets until it receives an acknowledgement indicating that the sent packet was properly received. For this reason, a special packet type was created for the sole purpose of acknowledging the receipt of packets. An ACK packet, as we came to call it, would consist of a header byte and a checksum byte and would be followed by two standard trailer bytes as is the case with all packets in the μMPX protocol. It was
apparent to us that the ACK packet was one packet type that must never be acknowledged in order to avoid an endless amount of acknowledgement communication traffic.

This prompted us to briefly reevaluate our needs pertaining to error correction. We concluded that error correction was mandatory in all transmission cases with the possible exception of text data. It was at this point that we decided to make error correction on data packets a user controllable option. This decision brought a new change to our, previously discussed, header byte. A new field, consisting of a single bit, was introduced into the header byte. This new field would indicate whether the sending process would be expecting an acknowledgement on a given packet. For obvious reasons we called this the ACK bit of the header byte, (See Figure 3-2).

+---------+---------+--------+---------+--------+---------+---------+--------+
| I  I  A  I  I  I  I  | packet type | C | destination | X |
+----------+-----------+---------+----------+---------+--------+----------+---------+-------+

Figure 3-2: Header Byte with ACK bit
In all cases where the ACK bit was set, the receiver would have to return an ACK packet to the sender before another packet would be sent.

The fact that the header byte, in all packets, contained the destination of the packet enabled us to enforce this method on a channel by channel basis. That is to say, even though the sending process may be waiting for an acknowledgement for a packet sent to window one, it could still send pending packets to other windows or it could send maintenance packets. The ACK packet can be used to specifically acknowledge a packet that was bound for a particular destination by passing back that destination in its header byte. In this way, a reasonable bandwidth of transmission is preserved.

3.3.2.2. Negative Acknowledgements

It occurred to us that since ACK packets would never be acknowledged we might be able to make use of that bit to indicate positive or negative acknowledgement. By using the ACK bit this way, the ACK packet can be used to indicate the receipt of corrupt packets as well as sound packets. The receiver, upon detecting a checksum error, would still transmit an ACK packet back to the sender but would leave the ACK bit clear to indicate that the sender should retransmit the packet. If the receiver were to determine that the packet was sound it would simply send an ACK packet with the ACK bit set. For those cases where the receiver would be unable to send a negative acknowledgement, due to lost data, we decided that the sender would retransmit the packet automatically if it did not receive an acknowledgement within a given time period.

In cases where packets are not to be acknowledged, the sender simply sends
the packet and marks it acknowledged immediately. The receiver never sends an ACK packet and discards those packets that are perceived to have errors.

3.3.2.3. Sequencing Errors

Although it may appear that we have resolved the issues of error detection and error correction, it is a fact that our solutions have created a new obstacle. As stated thus far, our solutions to error detection and correction leave the possibility that a packet may be retransmitted unnecessarily. The simplest scenario that illustrates this is one where a packet is sent and acknowledged, but, due to a transmission error, the acknowledgement is not received by the sender and the packet is resent. In this way, the proper sequence of received packets has been impaired. This dilemma lead us to the notion of sequencing the packets from within the protocol, allowing the recipient to detect a sequencing error.

Since our acknowledgement scheme restricts the transmission of a packet until the previous packet has been acknowledged, a simple binary sequence should suffice our needs. To accomplish this, we utilized the last unused bit of the header byte. The final format of the header byte is as shown in Figure 3-3.

By giving each packet from a given channel a specific sequence number, a one or a zero, the receiver can detect and discard duplicate packets. It is still important for the receiver to acknowledge those packets, however, to prevent the sender from endlessly retransmitting the same packet. To safeguard against duplicate ACK packets the receiver must include the sequence number of the packet it is acknowledging within the returned ACK packet. This allows the sender to detect duplicate ACK packets which can be safely ignored.
3.4. Formal Packet Types and Formats

Having resolved the major issues surrounding the development of the \( \mu \)MPX protocol we proceeded to define the various packet types we would need. Initially we defined three more types in addition to the one ACK packet type previously discussed.

The first packet type deals with communicating text data. The sole purpose of this packet type is to transfer keyboard input from a particular window on the client machine to the appropriate process on the server machine and vice versa. This packet type, referred to as the data packet, consists of a header byte, a length byte, a checksum byte and a variable number of data bytes not to exceed 255 in number. The length byte in this packet is used to indicate the number of data
bytes the packet contains.

The second packet type is used exclusively for communicating special information related directly to the operation of the window system. These packets, referred to as maintenance packets, are used to initialize communications on startup, open windows, close windows, turn acknowledgements on or off for data packets, and terminate the multiplexed communication session. The format of this packet type consists of a header byte, followed by a command byte and a checksum byte. The command byte of this packet type is broken into two fields, a five bit field indicating the command to be executed and a three bit field indicating which window the command might pertain to. These commands will allow the user to: open a window; close a window; terminate the μMPX session; control whether or not data packets will require acknowledgments.

The third and final packet type defined is the library packet. Having the exact same structure as the data packet, the library packet was defined in order to allow processes on the host machine to make client-specific function calls to the client machine. In this way, host processes would be able to access specific client functions such as line drawing routines. We decided that the host processes should issue a special escape sequence, in preface to some textual command, in order to alert the window server process that the following data was to be sent in a library packet rather than a data packet.

These packets types along with the previously defined ACK packet type formed the four initial packet types we built our windowing system around.
3.4.1. Packet Structures

The structures of our initial packet types are portrayed by the following pseudo-code definitions:

Structure DATA_PACKET {
  byte: header;
  byte: length;
  byte: checksum;
  byte: data[255];
}

Structure LIBRARY_PACKET {
  byte: header;
  byte: length;
  byte: checksum;
  byte: data[255];
}

Structure MAINTENANCE_PACKET {
  byte: header;
  byte: command;
  byte: checksum;
}

Structure ACK_PACKET {
  byte: header;
  byte: checksum;
}
3.5. Startup Handshaking Sequence

Since the client process will initially need to be in a basic terminal emulation mode, the server process will have to inform the client process that it is running. This could be accomplished by simply having the server send a special escape sequence which, when received by the client, would cause the client process to enter its multiplex mode. Once in multiplex mode, the client would send a maintenance packet to the server process indicating that it is indeed in multiplex mode. If the server process does not receive this maintenance packet within a certain amount of time, the server process will terminate. This should be of particular benefit in the event that the server process is started by someone not working from a μMPX client machine.

3.6. Flow Control and Hardware Buffer Overflow

Flow control and avoiding hardware buffer overflow are two issues that must be carefully considered when undertaking a project such as this. While avoiding hardware buffer overflow is really an implementation issue rather than a protocol issue, it is necessary to consider potential side effects the protocol may impose.

The standard ASCII characters used for flow control are CTRL-Q and CTRL-S which have the octal values 021 and 023 respectively. Since these values could be contained within a packet it is possible that the protocol definition could obstruct the basic flow control mechanism between the client and server machines. For this reason, we decided to include an implementation decree within the definition of the protocol. During the actual transmission of any packet, the sender must
examine each byte before it is transmitted and must split all bytes with an ascii octal value of 021 or 023 into two bytes. Each byte must have the high order bit set to indicate this special condition. The lower four bits will contain half of the original byte's value. In order to avoid conflicts with other bytes that may already have the high order bit set, the same procedure will be used on all those bytes that already have the high order bit set with the exception of trailer bytes. In this way, the only bytes that will have the high order bit set will be either trailer bytes, which can and will have other bits set in the high half of the byte, or special byte pairs that together contain a single eight bit value and have only the high bit set in the high half of the byte.

As each character is received it must be examined to determine whether it is part of a special byte pair that will need to be reassembled in order to create the actual byte value before inserting it into the packet being created. In all cases the sender must send the low order bits in the first byte of a special byte pair and the high order bits in the second byte. By adhering to this scheme both client and server processes should be able to successfully transmit all kinds of data without disrupting the data flow between the two pieces of hardware.

3.7. Final Amendments to the Protocol

After implementing working prototypes of the client and server processes we decided that there were two ways we could improve the efficiency of the \(\mu\)MPX protocol. First, the two trailer bytes used to encapsulate each packet could be reduced to a single trailer byte. Since the definition of the protocol guarantees
that no other bytes transmitted will have the same value as a trailer byte, we are assured that the trailer byte value is unique. For this reason, we can make this modification without worrying about side effects or trade-off. This change saves one byte per packet; a significant savings considering that most packets tend to be small in size.

The second improvement consisted of creating a new packet type, called a data1 packet, which, like the data packet, would be used for transmitting text data. Unlike the standard data packet, the data1 packet carries only one data character and therefore does not require a length byte. This new packet type transports single character text data much more efficiently than the standard data packet. The need for this type of packet comes from the fact that typed text is transmitted and echoed in packets containing only single data characters due to the relatively slow speeds at which people type.

The final change to the protocol involved a functional enhancement that allows the client process to inform the server process that a window has been resized. To achieve this, a new packet type, the resize packet, was defined. This new packet type consists of a header byte, a checksum byte, a row byte, a column byte, two bytes indicating the number of pixels in the x direction, and two bytes indicating the number of pixels in the y direction. Since this information may not be of use on all possible hosts we decided that while the server process must be able to receive this packet type it could optionally discard this packet type if the information was not useful.
3.7.1. New Packet Structures

The structures of the new packet types could be portrayed by the following pseudo-code definitions:

```
Structure DATA1_PACKET {
    byte: header;
    byte: checksum;
    byte: data;
}

Structure RESIZE_PACKET {
    byte: header;
    byte: checksum;
    byte: rows;
    byte: columns;
    byte: x_pixels_1;
    byte: x_pixels_2;
    byte: y_pixels_1;
    byte: y_pixels_2;
}
```
Chapter 4

Designing and Implementing the Server Process

4.1. An Initial Design

The first step in the design of the μMPX server process was to define the various tasks it would be responsible to accomplish. These tasks were easily divided into three categories: those tasks related to the reception of communication packets, tasks related to the management of window processes, and tasks related to the transmission of packets to the client machine. Having this done, it appeared that implementing these as separate processes, each executing in a somewhat asynchronous fashion, might produce a more efficient implementation than a single process implementation. By implementing the server process in this way, I hoped to simplify the overall design by restricting the responsibilities of each process. For example, by dedicating one process to the task of receiving incoming packets, that process could enter a blocked state when waiting for incoming data. In a single process design, it would be necessary for the process to fulfill other processing needs when data was not being received. Thus, the first working implementation of a μMPX server was designed as three separate processes.

The next step was to determine the best way to initiate these processes. To accomplish this, I decided that a total of four programs would be created. Three
of these relate to each of the three processes previously discussed. The fourth program would be the one the user executes and the one that actually initiates the other three processes. This fourth process, referred to as the μMPX process, initiates the other processes by creating duplicate, running, copies of itself in memory using the UNIX “fork” system call. Prior to creating these processes, some means for allowing these processes to communicate with one another must be secured by the μMPX process. The μMPX process does this by utilizing the UNIX “pipe” system call in order to set up special communication channels called pipes. By initializing these channels first, the μMPX process guarantees that the new children processes will be able to inherit these channels when they are created. It is in this way that the send, receive and manager processes will be able to communicate with each other. Once created, the individual process copies transform themselves into the appropriate send, receive or manager processes by invoking UNIX “exec” system call.

Next the responsibilities of the three processes had to be defined.

4.1.1. The Send Process

First, it was determined that the send process would save all of the user's current terminal settings and reset them in a way that would be more suitable for packet driven communications. For example, by default, all characters received by the host are usually echoed back to the terminal device. This is particularly inappropriate for packet driven communications and, therefore, must be deactivated before packet information can be reliably received. Similarly, when the μMPX session is terminated, the send process must be certain to reset all terminal
parameters to their original states so the user may continue to utilize the host machine.

Next, the send process would have the duty of informing the client machine that \(\mu\)MPX is running. This would be easily accomplished by sending a special escape sequence which would be acknowledged by a special startup maintenance packet. To guard against a user accidentally starting the \(\mu\)MPX server, the send process must receive this packet, via the receive process, within a given time period. Otherwise, the send process, assuming there is no client process, will terminate the manager and receive processes, reset the terminal parameters and terminate.

If the startup maintenance packet is successfully received, the send process begins its standard tasks. They include checking for data from the receive process, checking for data from the manager process, and sending packet information to the client process through the terminal device. To accomplish these functions the send process maintains a set of eight queues. One queue, the service queue, is reserved for service-oriented packets, such as maintenance, acknowledgement and resize packets, that must be sent to the client. The other seven queues are used for storing packets that are associated with the seven possible text windows the user may open.

The send process may receive three types of information from the receive process. First, the receive process may request that the send process acknowledge or negatively acknowledge the receipt of a packet. In this event, the send process would place an appropriate acknowledgement packet on the service
queue for transmission to the client. Second, the send process could receive an
acknowledgement packet that had been sent by the client process. In this event,
the send process removes the current packet from the appropriate queue and
marks the next packet for transmission. Lastly, the receiver passes all
maintenance packets to the send process in order to inform it when windows are
opened and closed as well as when the client requests that the session be
terminated.

From the manager, the send process only receives data or library packets
from processes associated with windows. As each packet is received from the
manager process it is added to the appropriate queue for eventual transmission.

While not receiving data from either the receive or manager processes, the
send process must check the status of the top packet on each queue. If the
packet is marked for transmission the send process transmits it, marks the packet
not to be transmitted and records the time at which it was sent. If the packet is
not marked for transmission, the send process checks to see how long ago the
packet was sent. If a specific period of time has elapsed since the packet was
sent, the packet is retransmitted and the new transmission time is recorded. This
behavior could be repeated indefinitely if the packet is never acknowledged by the
client process.
4.1.2. The Receive Process

The receive process is wholly responsible for the reception and initial verification of all incoming packets. To accomplish this, the receive process reads an eight bit character from the standard input channel and, assuming this character is the header byte of an incoming packet, determines the format of the incoming packet. Once the entire packet is supposedly received, the receive process expects two trailer bytes* to positively mark the end of the packet. If these trailer bytes are missing, the receive process assumes that there has been a transmission error and tries to resynchronize with the client machine by discarding all incoming data until two consecutive trailer bytes are found. Once this is done, the send process expects that the next byte it reads will be the header byte of an incoming packet.

If, on the other hand, the packet is properly terminated by trailer bytes, the receive process performs a simple verification test by computing a checksum for the packet and comparing it to the checksum that was sent within the packet. This value is computed by totaling the eight bit values of each packet byte, with the exception of the checksum byte and trailer bytes, and performing modulo division on the total with some predetermined value. If the computed checksum does not match the checksum found within the packet, the receive process discards the packet. Before discarding the packet, however, the receive process examines what it believes may be the packet header byte. If the ACK bit of this

*In the revised version of the protocol, only one trailer byte is expected.
byte is set, it will create an ACK packet to negatively acknowledge the reception of the packet and will pass this packet to the send process for transmission to the client machine. If the header byte of the received packet was not corrupt, this ACK packet will trigger the retransmission of the corrupt packet. In the event that the header byte was corrupt, the bogus ACK packet should be ignored by the client process or, at worse, will cause the unnecessary retransmission of some other packet by the client process. In any case, no harm can be caused by attempting to inform the client process that a corrupt packet was received.

In all cases where the receive process is able to verify a packet successfully, the ACK bit of the header byte is examined to determine whether client requires an acknowledgement or not. If the ACK bit is set, an ACK packet will be created and passed to the send process for transmission to the client. Next, the header byte is examined in order to obtain the packet type. By keying on the packet type, the receive process is able to ascertain the proper destination of the packet. If the packet type indicates that it is a maintenance packet the receive process will pass the packet on to both the send process and the manager process; all other packet types are passed only to the manager process.

4.1.3. The Manager Process

More complex than either the send or receive processes, the manager process must coordinate all window communications to and from UNIX processes. Additionally, when the client process requests that a new window be opened, the manager process must create the necessary processes on the UNIX host. In this way, the manager process also manages the creation of UNIX processes that are
associated with open windows on the client machine.

To accomplish all of this, the manager process is compelled to rely upon special UNIX devices called pty's (pronounced pities). Pty’s are actually pseudo devices that are implemented in the kernel of the UNIX operating system. The name pty comes from pseudo terminal. Each pty is made up of two special files referred to as a master/slave pair. Although they are utilized in the same way as normal files, when opened for reading and writing, data that is written to the master file is available for reading on the slave file. Similarly, data written to the slave file can be had by reading the master file. Moreover, pty’s have the same characteristics and parameters that actual terminal devices have. Hence, input editing characters, such as used for character deletion or line deletion, can be set as well as many other terminal parameters. It is by setting these parameters a pty is made to function as a terminal would. Consequently, this is what enables a μMPX window to emulate a terminal.

Initially, the manager obtains a maintenance packet from the receive process which indicates that a new window should be opened. Within the header byte of this packet is a field that tells the manager which window is being opened. This field always contains a three bit value between one and seven. Having received this packet, the manager process obtains a pty from a pool of publically available devices and opens both the slave and master files with exclusive access to inhibit their use by other users of the host computer. File descriptors which act as indices to the pty are kept in a special data structure which allows the manager process to associate them with a particular window. At this point the manager
process must set the terminal parameters of the pty to those that would normally be associated with a terminal. This is done by simply copying the already saved parameters of the true terminal device to the pty's parameter tables.

Next, the manager process creates a duplicate process image of itself which, having detected that it is a duplicate rather than an original process, sets the slave side of the pty to be its standard input and output channels, and transforms itself into a UNIX shell process. A shell process is basically a command line interpreter that is usually started when a user logs in. It is through this shell that a user communicates with the host computer. At this point the manager may communicate to the shell through the master side of the pty. The shell process receives data through the slave side of the pty just as if the pty were an actual terminal. In the same way, the shell process writes all of its output to the slave side of the pty, allowing the manager process to read that data from the master side of the pty.

Whenever there is data to be read from a pty, the manager process must create a packet, put the data into the packet and pass the packet to the send process for transmission to the client process. All data read from pty's must be scanned for a special escape sequence indicating that a special library request is being made. In these cases, a UNIX process is requesting that a specific function in a library on the client machine be executed. This data is placed in a library packet rather than a data packet before being passed to the send process.

When packets arrive from the receive process they are processed according to their packet type. If the packet is a data or library packet its contents are
written to the appropriate pty master file. If the packet is a maintenance packet it could request that a new window be opened, an old window be closed, or it could request a general shutdown.

To close a window, the manager must send a signal to the shell process that is associated with the window to be closed. This signal is trapped by the shell and causes the process to terminate. The pty files are then surrendered, making them available to the public once more. When a shutdown request is received, the same procedure is repeated for all open windows before the manager process exits.

If the manager process receives a resize packet from the receiver, it simply updates a special parameter table associated with the pty with the new display size values.

Refer to Appendix B for a diagram depicting the three process implementation of the μMPX server.

4.1.4. Flow Control

Initially, the issue of flow control was resolved by simply turning on automatic flow control for the controlling terminal. This was done by the send process at the same time that it set the terminal to eight bit mode and half-duplex for packet communications. Setting automatic flow required the send process to make two system calls: one to set the flow control characters; a second to activate automatic flow control.
4.1.5. Design Flaws

A thorough design phase allowed the μMPX process to be easily implemented, however, rigorous testing uncovered two major faults in the initial design. The first of these faults, surprisingly, took a great deal of time to verify. After using the system for some time, it was discovered that a high number of transmission errors were occurring, particularly when large amounts of data were being transmitted. It appeared that the flow control mechanism was not operating as expected although it was not immediately apparent whether the problem lay with the client process or the server process. Further investigation proved that the server process was to blame. The unfortunate conclusion was that although automatic flow control was turned on for the controlling terminal by the send process, it was also nullified by the fact that the terminal was in eight bit mode. Since the μMPX protocol is based on eight bit transmissions it was imperative that we have the controlling terminal in eight bit mode. Thus, a new solution to flow control had to be found. It was, however, heartening to realize that the μMPX protocol worked so well that we were unaware of the problem until statistics were taken on packet transmissions.

The second, and more devastating flaw in the design, was found even later. Throughout our initial testing, we simply opened windows and started processes in each that would produce output. These tests proved our protocol to be sound and our client and server processes to be working. They did not, however, emulate an actual user session. It was not until after performing several tests such as these that we found cause to type a CTRL-S to stop the output of a process or a CTRL-
C to abort a process. On programs that output small amounts of data there seemed to be no effect when CTRL-C or CTRL-S were typed.

On programs that output large amounts of data, the CTRL-C or CTRL-S did perform as expected but not as quickly as one would desire. The cause of this problem was easily found. When a process on a pty would generate output data, the manager process would read that data and create packets that it would then give to the send process. The send process would take these packets and place them on a queue for transmission to the client process. Unfortunately, the manager process would create these packets much faster than the send process could send them to the client. The queue maintained by the send process would grow at a rapid rate while actual transmission of packets proceeded at a relatively slow pace. At the time that a CTRL-S or CTRL-C character was actually sent through the manager to the running process the send process already had a vast amount of data queued for the client. The send process, unaware of any transmissions to the running process, would continue to drain the queue until all data was sent. This flaw proved to be intolerably annoying and prompted a complete redesign of the server process.

4.2. A Revised Design

Leaving the issue of flow control for later, I undertook to redesign the μMPX server almost entirely. My goal was to create a single process which could fulfill all of the functions required of the μMPX server. Careful thought resulted in a new design that would not only resolve the issue at hand but would also increase
the efficiency of the server process and reduce the amount of source code.

4.2.1. A New Foundation

The new design utilizes the "select" UNIX system call. This system call examines a given set of I/O channels and returns a list of those channels that have data to be read. This system call will block the calling process until some data becomes available or until some time period has expired. By using this in the main loop of the μMPX process, the process can determine its processing needs without creating a large amount of processing overhead.

When the "select" system call returns, it will either indicate that data can be read on some channel or it will indicate that the function timed out. If the function timed out, there is little for the μMPX process to do other than to re-send packets it has already sent if they have not been properly acknowledged. Once this is done the μMPX process loops back to the "select" system call.

If there is data to be read, the μMPX process will determine whether the available data is packet data, window data or both. Whenever packet data is available it will be processed first by calling a special function designed to read packets from the standard input device (which corresponds to the client machine). If there is pending window data, the μMPX process determines which ptys are to be read and invokes a function designed to create and send packets containing that data. A more detailed explanation follows in the next two sections.
4.2.2. Window Data

If "select" indicates that there is data to be read, the μMPX process will determine whether the data is to be read from the standard input device or from a pty device. Each time there is data to be read from a pty device, the μMPX process calls a routine that determines which pty is to be examined and whether the last packet sent from that pty has been acknowledged. If the last packet has been acknowledged, a new packet is created with data read from the pty device. The new packet is then sent and marked as unacknowledged if acknowledgements are activated. If acknowledgements are not activated the packet is automatically marked as acknowledged.

4.2.3. Packet Data

Whenever packet data can be read on the standard input device, it is given top priority over all else. This is accomplished by making the host generate a software signal, similar to an interrupt, whenever data is available on the standard input device. By trapping this signal and branching into the packet-reading function, the μMPX process is able to read the incoming data as soon as it is available. The packet-reading function has been designed as a finite state machine in order to properly assemble the various packet types. By utilizing global variables this finite state machine may receive a single packet in several pieces, as when flow control interrupts a transmission. By preserving its context it can relinquish process control allowing processing in the interim. In this way, it is able to continue the assembly of the incoming packet as the data becomes available.
Each time a complete packet is perceived to exist, the packet reading function verifies the packet and, if requested, sends an appropriate ACK packet to the client machine. It then examines the packet, determines the packet type, and processes the packet according to the packet's type and destination. When this is done, the packet reading function returns control to the calling function. This usually results in returning process control back to the main program loop.

4.2.4. Flow Control Solved

With the single process implementation, the issue of flow control was easily solved. Since our protocol guarantees that packets shall not contain actual XON or XOFF characters, our μMPX process can assume that any such character received has been generated by the client machine for reasons pertaining to flow control. With this being the case, the μMPX process can voluntarily block all transmissions whenever an XOFF is received and resume transmissions any time an XON is received. By enabling a special signal facility the server process can be interrupted any time there is incoming data. In this way, we can insure that data sent by the client process, including flow control characters, will be processed as soon as possible.
5.1. Future Enhancements and Additional Applications

The initial definition of the µMPX protocol, as described in this paper, includes definitions for six separate packet types, while allowing for a total of eight. The two unused packet types could be used in the future to implement protocol enhancements that are not known at this time.

The existing library packet type can be used to transport a wide variety of remote procedure calls. Since the contents of the library packet do not affect the transmission of the protocol, those contents will remain transparent to the server process. This allows the implementation of a wide variety of libraries, each of which would correspond to a particular type of client machine. The possibilities for enhancement in this area are limitless.

Additional applications may include simple terminal use over exceptionally noisy data lines. By using the µMPX protocol, noisy lines, such as some long distance telephone lines during business days, become usable because of the error detecting and error correcting characteristics of the protocol.
5.2. Critique of Efforts and Results

The first steps in beginning this project were to define the scope of the project and to design an efficient protocol to fulfill the communication needs of the project. Defining the scope of the project was fairly easy. From the start we had one simple goal which essentially defined the entire project: we wanted to develop a simple, efficient, and reliable windowing system that could run on inexpensive microcomputers.

5.2.1. Protocol Definition

Realizing that the development of an efficient, error detecting and error correcting protocol would determine the success of the project we began a rigorous routine of design, inspection and refinement that would account for the bulk of our efforts. By critically analyzing each new development in the protocol, and analyzing its effects on the rest of the protocol, we were able to create a protocol that fit our needs. When tested, our new protocol performed better than we had hoped. The initial version of the protocol proved to be completely workable. Only one minor modification was made to improve efficiency; the two trailer bytes were reduced to one. Other improvements consisted of two enhancements that did not alter the existing protocol in any way.

Our initial decision to create a packet driven protocol proved to be correct. Although the packet scheme does require transmission overhead that may not have been required with other schemes, it functions well within our needs. Packets with the greatest amount of overhead are always very small in size.
Generally these packets contain single characters of text data generated by the user typing on the keyboard. Since typing is a relatively slow process, the overhead is essentially unnoticeable. This is particularly true if the server process prioritizes the active window in such a way that packets are returned to that window whenever possible.

5.2.2. Project Implementation

Our success in creating the µMPX protocol prompted the eventual success of the entire µMPX project. Starting with a reliable protocol as the project's foundation, only implementation details remained to be resolved. As in the development of the µMPX protocol, I set out to rigorously design the µMPX server process. Aware that my design was much different from other similar undertakings, I strove to uncover its hidden flaws. In time, satisfied that I had a workable design, I implemented the µMPX server process. Due to the impeccable design of the µMPX protocol, the first µMPX server worked quite well even though its flow control code was badly flawed.

Two unresolvable problems required the µMPX server to be completely redesigned. At first it seemed that a large amount of effort would be needed to redesign and reimplement the µMPX server. However, after carefully redesigning the server process, however, I realized that much of the original implementation could be reused as a result of its modular design. Thus, I was able to reimplement the µMPX server very quickly, finally achieving my original goal.

The end result of this project was the confirmation that inexpensive
microcomputers can be used to support multi-window environments. This fact, coupled with the newly developed μMPX protocol, will hopefully bring windowing environments, and their benefits, to users who previously could not afford them.
A.1. Packet types

**Data1:** Used for transmitting single characters of text data between UNIX processes and the client process. It is represented by a zero value in the highest three bits of the header byte.

**Data:** Used for transmitting multiple characters of text data between UNIX processes and the client process. It is represented by a value of one in the highest three bits of the header byte.

**Library:** Used for transmitting special function calls from UNIX processes to the client process. It is also used in those cases where the called function returns some value to the UNIX process that made the call. It is represented by a value of two in the highest three bits of the header byte.

**Ack:** Used to acknowledge the receipt of other packets. It is represented by a value of three in the highest three bits of the header byte.

**Maintenance:** Used to transport special instructions from the client process to the server process. It is represented by a value of four in the highest three bits of the header byte.

**Resize:** Used to inform the host when a window has been resized. Optionally, this packet type can be ignored by the host. It is represented by a value of five in the highest three bits of the header byte.
A.2. Packet Formats

Data1: 
  header byte
  checksum byte
  data byte

Data: 
  header byte
  length byte
  checksum byte
  up to 255 data bytes

Library: 
  header byte
  length byte
  checksum byte
  up to 255 data bytes

Ack: 
  header byte
  checksum byte

Maintenance: 
  header byte
  command byte
  checksum byte

Resize: 
  header byte
  checksum byte
  rows byte
  columns byte
  x pixels byte one
  x pixels byte two
  y pixels byte one
  y pixels byte two

A.3. Maintenance Commands

Quit: 
  Terminate μMPX session. The high five bits of the command byte contain a zero value.

Started: 
  Sent by the client process to inform the server process that it has received the startup escape sequence and has entered multiplex mode. The high five bits of the command byte contain the value one.
Open: Open a new window. The high five bits of the command byte contain the value two. The low three bits indicate the window to be opened.

Close: Close a new window. The high five bits of the command byte contain the value three. The low three bits indicate the window to be closed.

Ack_On: Data packets are to be acknowledged. The high five bits of the command byte contain the value four.

Ack_Off: Data packets are not to be acknowledged. The high five bits of the command byte contain the value five.

A.4. Miscellaneous

Trailer The trailer byte has an octal value of 365.

Checksum The checksum for packets is computed by totally all packet bytes, except the trailer and checksum bytes, and dividing the sum by 251. The remainder is the checksum value.

Startup When started, the µMPX server sends an escape, (33 octal), followed by the string "STARTUP" to the client process.

Library Unix processes making function calls to the client machine output an escape followed by an asterisk to inform the µMPX server that a library packet is needed. This escape sequence is followed by a single byte indicating the total number of bytes the packet will transport and the bytes themselves.
Appendix B

A Three Process Implementation of the Server
Bibliography


