MacBlit: A microcomputer based multiple window terminal emulator for UNIX operating system

Sidney Wang
The University of Montana
COPYRIGHT ACT OF 1976

This is an unpublished manuscript in which copyright subsists. Any further reprinting of its contents must be approved by the author.

Mansfield Library
University of Montana
Date: 1987
MACBLIT
A Microcomputer Based Multiple Window Terminal Emulator
for UNIX Operating System

By
Sidney Wang
B.S., University Of Montana, 1984

Presented in partial fulfillment of the requirements
for the degree of
Master of Science
University of Montana
1987

Approved by

[Signature]
Chairman, Board of Examiners

[Signature]
Dean, Graduate School

September 14, 1987
Date
MacBlit is a multi-window graphic terminal emulator which utilizes the Apple Macintosh personal computer to provide a friendly user interface for hosts which run UNIX BSD 4.2/4.3 and compatible operating systems.

UNIX operating system’s multitasking capability allows the user to run many programs at once. Unfortunately, the effectiveness of a multitasking environment has long been limited by conventional terminals. Because the concurrently running user programs all share the same terminal, outputs from different programs are sent to the same screen asynchronously, intermix together and become unreadable.

MacBlit provides a solution by allowing the user to open many process windows (up to 7) on the Macintosh screen. Each window will be assigned to a different program on the host so that output from that program will appear only in its own window. Since windows can be overlapped and the content of a window is preserved, the user may switch between programs with just a click on the mouse.

Windowing systems have been proven capable of increasing one’s productivity manyfold. MacBlit can offer such technology to mainframe users at a very low cost — by using a low cost microcomputer.
Table of Contents

Abstract ................................................................................................................................. ii
Table of Contents ......................................................................................................... iii
List of Tables and Figures ........................................................................................ v
1 Introduction ........................................................................................................ 1
   1.1 What is this paper all about? ............................................................................. 1
   1.2 The problem ..................................................................................................... 2
   1.3 Solutions ......................................................................................................... 3
   1.4 Project overview .............................................................................................. 4
2 Development Methodology ....................................................................................... 5
3 The Macintosh ........................................................................................................ 7
   3.1 Hardware .......................................................................................................... 7
   3.2 Operating System and User Interface Toolbox ............................................... 7
   3.3 User Interface .................................................................................................. 8
      3.3.1 Windows ................................................................................................... 8
      3.3.2 Menus ..................................................................................................... 10
      3.3.3 Controls .................................................................................................. 11
      3.3.4 Dialog box .............................................................................................. 11
4 The MacBlit Prototype ......................................................................................... 13
   4.1 MacBlit and µMPX - how they work together .................................................. 13
   4.2 MacBlit internal structure ................................................................................ 15
      4.2.2 Window Table .......................................................................................... 16
      4.2.1 Message Router ..................................................................................... 17
      4.2.3 DoEvent .................................................................................................. 18
5 The Communication Protocol .............................................................................. 19
   5.1 About the protocol ........................................................................................... 19
   5.2 Physical Layer .................................................................................................. 20
   5.3 Data Link Layer ............................................................................................... 21
      5.3.1 Protocol Design ...................................................................................... 21
         5.3.1.1 General error and the checksum .................................................... 22
         5.3.1.2 Sequencing error and the sequence number ................................. 22
         5.3.1.3 Out-of-Sync error and the SYNC character ............................... 23
5.3.1.4 XOn/XOff conflict and resolving method ........................24
5.3.2 Formats of different packets .......................................................  25
  5.3.2.1 Header .................................................................26
    5.3.2.1.1 PType ...............................................................26
    5.3.2.1.2 AckReq .................................................................27
    5.3.2.1.3 SeqNo .................................................................27
  5.3.2.2 Command Byte .........................................................28
  5.3.2.3 Length............................................................................28
  5.3.2.4 Data............................................................................29
  5.3.2.5 Checksum .....................  29
  5.3.2.6 Sync............................................................................29
  5.4 Implementation.................................................................29
    5.4.1 First Try - Interrupt .................................................................31
    5.4.2 Alternative - Finite State Machine .................................  32
  6 VT100 Emulator ........................................................................................................33
    6.1 Display Record .............................................................................33
    6.2 CharMap .............................................................................35
    6.3 Scrolling .............................................................................38
    6.4 Top and bottom margins .........................................................38
  7 Conclusion......................................................................................40
    7.1 What did we accomplish? ..........................................................40
    7.2 Looking into the future ...................................................................40
References..............................................................................................42
Glossaries............................................................................................44
List of Tables and Figures

Table A. Values of PType and their meanings
Figure A. Standard document window
Figure B. MacBlit’s System Set up dialog box
Figure C. MacBlit and μMPX
Figure D. MacBlit internal structure
Figure E. Packet Formats
Figure F. PacMan internal structure
Figure G. CharMap data structure
1 Introduction

1.1 What is this paper all about?

This paper presents the design of a multi-window terminal emulator called MacBlit. It utilizes the newest generation of microcomputer technology to provide UNIX-based computers a friendly user interface that, before, was only available on expensive graphic terminals costing many times as much.

The UNIX operating system is well known for its multitasking environment, richness of software tools and powerful command interpreters. Its multitasking capability is available at the user level, allowing the user to run many programs\(^1\) at once.

A UNIX process has at least three input/output channels opened. They are named, by UNIX convention, stdin (standard input), stdout (standard output) and stderr (standard error). Normally, stdin is associated with the terminal's keyboard whereas stdout and stderr are associated with the terminal's screen. A process can be either in the foreground or background. A foreground process is capable of accepting user inputs but a background process cannot — its stdin channel is disconnected from the keyboard. Both types of processes can output to the terminal. The UNIX command interpreter, called the Shell, has an operator ‘\&’ which runs a command in the background. For example,

```
cc prog1.c &
```

runs the C compiler on the file prog1.c and immediately returns control to the user without waiting for it to finish. Another command ‘fg’, brings a process to the foreground.

\(^1\) A running program is called a process. In this paper, “process” and “running program” will be used interchangeably.
Suppose Charlie, a very talented programmer, has just discovered a bug in his program. After about an hour of study, Charlie decides to rewrite a routine in one of the source files. He first enters a text editor to make some changes, saves the file and starts the compiler to rebuild the program. Since the compilation would take about 20 minutes, he puts the compilation process in the background and meanwhile enters the editor to document the changes. After 20 minutes, the program is ready to be tested and the design document is edited. User level multiprocessing can increase one’s productivity by allowing more than one task to be finished in the same amount of time.

1.2 The problem

The effectiveness of UNIX's multitasking environment has been limited by conventional single screen terminals. This is because all user created processes (background and foreground) share the same terminal, sending output to the same screen asynchronously, and intermixing together to become unreadable. For example, while Charlie was editing the design document, the compiler discovered some errors in the changes he made to a source file and sent error messages to the terminal. These were mixed with or even overwrote the design document on the screen.

Another problem associated with a conventional terminal is that when one program quits and a new one starts its execution, the output of the first program may be erased by the second. Some tasks, such as editing the reply message while reading the received mail, or editing more than one file at the same time by running multiple copies of a editor etc., are impossible. In general, any task that has a need to refer to the output of a previous program is difficult, if not impossible, to perform.
1.3 Solutions

To solve the above problems, computer scientists have invented the windowing concept. The main idea is to divide a terminal screen into partitions called windows, each acting as a separate terminal which may be assigned to a process so that the output of that process will be sent only to its own window.

There are many windowing systems being developed. The early ones, such as "window" utility on UNIX, were functionally too simple to be useful. They were primarily designed for text terminals. Each window was actually a split portion, either horizontally or vertically, of the screen; and so, the more windows there were, the smaller each one could be. In other words, one would not be able to edit in multiple full-size windows.

The later generation of windowing systems are more sophisticated. They support overlapping windows on large bit-map graphic terminals. Windows can also be resized and moved. In some cases an additional input device called a mouse is also available for easy window manipulations. Some well known examples include the XWindow package developed by MIT, SunView from Sun Microsystems [10] and Blit terminal from Bell Lab [3] [8] [9] [12]. Their major drawback is that the equipment they require tends to be quite expensive.

In January 1984, Apple Computer introduced the Macintosh personal computer which first brought a multi-window user interface to general users at an affordable price. Shortly after, other vendors released more hardware and software to mimic the Macintosh's user interface on other microcomputers. The most popular ones include Microsoft Window for IBM PCs, WorkBench for Amiga personal computers, and TopView for the Atari ST
series microcomputers. These windowing environments are either directly supported by
the operating system or by add-on library routines which can be called from user programs.

With the power of 16-bit microcomputers and pre-written windowing packages, it is
possible to develop a multi-window graphic interface for a UNIX host on one of these
microcomputers.

1.4 Project overview

The whole project consists of three main parts: a multi-window terminal emulator
running on the Apple Macintosh called MacBlit, a supporting program called \( \mu \)MPX
running on the UNIX host, and a communication protocol which provides a reliable
communication link between MacBlit and \( \mu \)MPX. This project had been implemented by
Scott Mulligan and myself. Scott Mulligan was responsible for the software on the UNIX
machine and I was responsible for the MacBlit program.

The Macintosh was chosen as the target machine because, 1) Its operating system
provides the most comprehensive set of routines to support the multi-window user
interface. 2) Low cost. 3) It is widely available; there are 10 Macintoshes installed in the
Department of Computer Science. 4) At the time the feasibility study was done (August,
1985), it was the only microcomputer being designed to fully support the windowing
environment.

In this paper, I will only describe the details of the MacBlit program. Discussion of the
\( \mu \)MPX program can be found in the paper written by Scott Mulligan [5].
2 Development Methodology

The project was basically experimental. At the time it was started, the Macintosh operating system contained bugs, the development system contained bugs, the Macintosh's documentation was incompletely, the Macintosh's windowing environment was new and the idea of a event-driven program structure was unfamiliar. With all these uncertainties involved, we felt that the traditional development methodology, such as the waterfall model, could not be applied. Instead, an incremental development approach was attempted. Following are the steps that we used:

(1) Specify requirement. A detailed requirements specification had been written as the proposal for the project. It described the scope of the project, its functionality, what the MacBlit's user interface should look like and how to use it.

(2) Design and implement the communication protocol. Since the communication protocol was crucial to the whole system and had to be well understood by both of us, we designed it together. We built two programs, one on the UNIX host and one on the Macintosh just for testing the correctness of the protocol.

(3) Build the MacBlit user interface. While the protocol was under development, I also started learning how to use the Macintosh toolbox library and I created a prototype for the MacBlit user interface. This prototype contained all the menu items and parameter setting dialog boxes. It could also open, close, resize, and move windows.
(4) Build the first working prototype. After obtaining a correct protocol implementation, Scott began to develop the first version of the μMPX and I started to develop a simple multi-window dumb terminal emulator.

(5) Build the second prototype. As we gained more knowledge about the project we learned what would work and what would not. The second prototype had two purposes: first, we corrected the mistakes made in the first one; second, I implemented the VT100 window. It was a difficult time for both of us because most of the previous work we had done had to be thrown away. μMPX was redesigned completely, everything except the user interface in MacBlit had to be thrown away, and although the definition of the protocol was only changed slightly, its implementation was totally different.

(6) Implement ASCII file transfer. My project advisor suggested that I add a file transfer feature in MacBlit, so I did.

(7) Test the MacBlit program. MacBlit was tested in two phases: testing by myself and testing by the public. After having tested the program as much as I could, I had my students, classmates, professors, and friends serve as beta testers. They received copies of the program, tested it, and sent me bug reports and suggestions. I then corrected the bugs, made enhancements, and sent them the new version to test again. The testing process has been continuing for more then six months and will continue so long as the program is still in use.
3 The Macintosh

3.1 Hardware

The Apple Macintosh is a small portable microcomputer weighing less than 20 lbs. It contains a Motorola MC68000 CPU running at 7.8 megahertz, 512K or more random access memory (RAM), 64K or more read-only memory (ROM), one or two disk drives, two RS232/422 compatible serial ports on the back panel for serial communication, a graphic input device called a mouse, and a nine-inch black and white bitmap video display. There is no special hardware to generate characters or graphic processor to draw lines and points. Everything that appears on the screen is drawn by software in the QuickDraw library.

QuickDraw is just one of the libraries residing in ROM; there are others which manage windows, menus, controls, memory, files etc.. All these libraries are collectively called "toolbox" which will be discussed in the next section.

3.2 Operating System and User Interface Toolbox

Macintosh ROM contains important components which differentiate the Macintosh from other microcomputers. It not only contains the operating system but also contains the User Interface Toolbox which enforces the "look and feel" of every Macintosh application. Toolbox routines are categorized into libraries called managers; each manages a distinct aspect of the user interface. For example, Window Manager allows the user to create, manipulate and dispose of windows; Menu Manager supports the use of menus; Control Manager deals with buttons, check boxes, and scroll bars.
Using the toolbox has many advantages: 1) It simplifies the task of creating a windowing environment for programs because most of the routines have already been written. 2) Different programs can have a consistent “look” so that once the user learns how to use one program he can also use the others. 3) Toolbox routines are highly optimized assembly codes. They are usually more efficient than high level language routines.

MacBlit makes heavy use of the toolbox and follows as close to a typical Macintosh application as possible.

3.3 User Interface

The Macintosh user interface consists of numerous elements and can be very complex. Due to the limited amount of available space in this paper, I will only describe the ones which are used in MacBlit.

3.3.1 Windows

A Window is a rectangular area on the Macintosh screen in which graphics and text can be displayed. Macintosh supports many types of windows, the one MacBlit uses is called “standard document window” as shown in Figure A.

---

2 A "typical" Macintosh application is the one that its user interface design follows the “Macintosh User Interface Guideline” [2] [4].
At the top of the window is the title bar, which displays the window's name at the center position. When the window is the front-most window, it is called the active window and its title bar will be highlighted by horizontal strips. When the window is inactive, the strips are removed. Inside the title bar, at the left, is the window's close box. Clicking inside the close box causes the window to disappear from the screen. The title bar also serves other purposes: clicking and dragging inside the title bar repositions the window; double clicking inside it enlarges the window to its full size, double clicking it again restores the window to its original size.

At the bottom-right corner of the window is the size box which is used to change the window's size. When the user presses and drags the mouse inside the size box, an outline of the window appears with its top-left corner “anchored” and its bottom-right following the mouse. When the mouse button is released, the window is redrawn to its new size.
Usually, a window only displays a small portion of a document. The invisible portion of the document is called the invisible contents of the window and can be scrolled into view by using two controls: the horizontal scroll bar—located at the bottom of a window, and the vertical scroll bar—located at the right of a window. Each scroll bar has three control parts: 1) up and down\(^3\) arrows which scroll the window's contents upward or downward a line, 2) page-up and page-down regions, which are the grey area inside a scroll bar, scroll the window's contents upward or downward a “page” at a time, 3) thumb box which can be dragged to show any portion within a document directly.

### 3.3.2 Menus

Menus are an integral part of the Macintosh user interface. They provide a common means of accepting commands from the user. At the top of the screen is the menu bar, which spans the screen and lists the titles of the available menus. Pressing and holding down the mouse button over one of these titles causes a list of menu items to be displayed under the title. Dragging the mouse down the menu causes one item after another to become highlighted on the screen. When the button is released, the currently highlighted item will be chosen, causing the designated action to take effect. If no item is highlighted when the mouse button is released, the menu will just vanish and the screen will be restored.

Pull-down menus are considered to be easier to use than the traditional command driven interface because users do not have to memorize commands. Also, each individual menu item can be disabled and enabled under program control. This reduces the chances of

---

\(^3\) To a horizontal scroll bar the up and down directions really mean left and right.
issuing wrong commands and simplifies the program by eliminating codes for checking illegal commands and thus increases the program's reliability.

3.3.3 Controls

A control is an object on the Macintosh screen with which the user, using the mouse, can cause instant action with visible results or change settings to modify a future action. The aforementioned scroll bar is a form of control. In addition, MacBlit also uses pushbuttons, checkboxes, and radio buttons.

Pushbuttons are used to make something happen immediately. MacBlit uses them to dismiss dialog boxes.

Checkboxes retain an on-or-off setting that affects the way something will happen at a later time. Clicking on a checkbox alternately turns it on and off, independently of any other control.

Radio buttons are like the checkboxes that are grouped together to offer a multiple choice. Turning any one button “on” makes all the others in the group “off”, so that only one button can be in the “on” position at a time. For example, MacBlit uses radio buttons to select baud rate from a set of possible values.

3.3.4 Dialog box

Dialog Boxes are temporary windows which are primarily used for displaying messages, warnings or implementing controls which are not frequently used. For example, in MacBlit, there is a dialog box which allows the user to change the communication parameters such as baud rate, number of data bits, number of stop bits etc. (Figure B).
### System Set Up

<table>
<thead>
<tr>
<th>Baud Rate:</th>
<th>300</th>
<th>2400</th>
<th>9600</th>
</tr>
</thead>
<tbody>
<tr>
<td>600</td>
<td>3600</td>
<td>19200</td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td>4800</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parity:</th>
<th>No</th>
<th>Odd</th>
<th>Even</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Bits Per Char:</th>
<th>7 Bits</th>
<th>8 Bits</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Stop Bits:</th>
<th>1 Stop Bit</th>
<th>2 Stop Bits</th>
</tr>
</thead>
</table>

- **Save Temporary**
- **Save Permanent**
- **Cancel**

Figure B. MacBlit's "System Set Up" dialog box.
4 The MacBlit Prototype

4.1 MacBlit and μMPX - how they work together

MacBlit has two operation modes: single-window mode and multi-window mode. When MacBlit first executes, it is in single-window mode and opens a window called “Login Window” to emulate the DEC's VT100 text terminal. The user can then login to the host and perform any task as he would with a normal VT100 terminal. MacBlit remains in single-window mode until the user executes the μMPX program to put it into multi-window mode.

μMPX will first initiate a three-way handshaking procedure to ensure that MacBlit is properly initialized. Any error occurring during the initialization procedure will terminate μMPX and bring MacBlit back to single-window mode. At the end of the three-way handshaking, the two programs are ready to communicate.

At this point, the user can open new client windows by choosing the “New” window command. There is a maximum of seven windows allowed to be opened at the same time. Due to the small size of a Macintosh screen, however, four will probably be the practical limit. For each newly opened window, MacBlit sends a command to μMPX, asking it to create a new client shell. There is a one-to-one relationship between a client window and a client shell, and this information is kept by μMPX internally.
A client shell is connected to μMPX by a pseudo terminal (ptty). A ptty has an input channel and an output channel. Inside μMPX there is a polling loop which repeatedly polls the ptty output channel of each client shell. If it gets data from a ptty, it first determines in which window the data should be displayed, and then sends the data along with the window id to MacBlit. When MacBlit receives the data, it looks at the window ID and routes the data to the correct window.

The Macintosh keyboard is always associated with the active window. When the user types a character, MacBlit will send the character along with the window id of the active window to the μMPX. When μMPX receives the character, it will route the character to the correct client shell through its ptty input channel. Since any window can be made active by clicking with the mouse inside any portion of the window, the user can switch between processes very easily.

Figure C. MacBlit and μMPX
In Figure C, windows 1, 2, and 3 are three client windows opened on the Macintosh; shells 1, 2, and 3 are three client shells created by \( \mu \)MPX, each corresponds to a client window. P1 and P2 are two client processes created by shell 1. Similarly, process p3 was created by shell 2 and p4 was created by shell 3.

When the user is completely done with a window, he can release the window by choosing the Kill Window command. MacBlit will erase the window from the screen and send a kill command to \( \mu \)MPX which will then kill the associated client shell and all processes created under the shell.

4.2 MacBlit internal structure

Figure D shows the internal structure of the MacBlit program. MacBlit is made up of six parts: a communication protocol (PacMan) which will be discussed in chapter 5, an event handler called DoEvent, a Message Router, a set of terminal emulators, a global Window Table, and a System Control routine.
Notice that although figure D shows three emulators, only the VT100 emulator was implemented in the second prototype and will be discussed in chapter 6.

4.2.2 Window Table

The Window Table is a global data structure; it has eight slots, numbered from 0 to 7. Slot 0 is unused, slots 1 to 7 are for the seven windows. Each slot is a record of three fields, in terms of C syntax the Window Table can be written as:
typedef struct {
    char    *dspPtr;
    int     wType;
    int     wStatus;
} window_slot;

window_slot window_table [8];

The dspPtr is a pointer to a display record. A display record is a data structure used by an emulator to store data which will be displayed on a window. Different emulators may have different structures for the display record. Therefore dspPtr is actually a generic pointer and should be type-cast to the correct display record type before being used.

The wType is the window type which identifies what terminal a window emulates.

The wStatus is the window status which tells whether a window is currently active, closed, behind another window, or has its window slot unused.

4.2.1 Message Router

The Message Router is responsible for routing the messages to the correct emulator. When it gets the message from the protocol, it uses the window id in the message as the index into the window table, examines the wType field to find out which window the message is for, and then calls the correct emulator routine with the message and the display record pointer in dspPtr field as parameters. A message may have a window id equal 0, meaning that the message should be used to alter MacBlit's operational status. In this case, the message router will call the System Control routine instead. In the second prototype,
System Control Messages\(^4\) are being sent from MacBlit to \(\mu\)MPX only. Therefore, System Control is not implemented.

### 4.2.3 DoEvent

The sole responsibility of DoEvent is to get the event from the operating system's event queue, find out which window is effected and call the appropriate routine of an emulator to handle the event.

A window can be opened, closed, moved, resized, and scrolled. These operations are all direct results of an event — namely, clicking the mouse button. There are many events that need to be handled by a Macintosh program; some are generated by the user and some are generated internally by the Macintosh operating system. The user-generated events are associated with the input devices such as clicking the mouse button, hitting a key, or inserting a disk. The operating system-generated events are usually a by-product of the result of a user event. For example, the activate event is the result of making a window active; the update event is the result of exposing an overlapped window.

Every emulator has its own way to deal with these events. For example, a click in the vertical scroll bar may scroll upward one line in a VT100 text window but may only scroll five pixels in a TEK4014 graphic window. Therefore, the event handling routines are an integral part of an emulator.

\(^4\) A System Control message is called a System Maintenance message when sent from MacBlit to \(\mu\)MPX.
5 The Communication Protocol

5.1 About the protocol

MacBlit uses an error-detecting-retransmitting communication protocol called Window System Protocol (WSP) to provide an error-free connection to a UNIX host. The decision to develop such a protocol was made at the very beginning of the project. This was because MacBlit exchanges control information with the host to create and destroy processes, and any error that alters this information may cause catastrophic results. For instance, an error which changes an open window command to a kill window command may mistakenly destroy all the results that have been accumulated by a process in many hours. High reliability can not be obtained at no cost; there is certainly some overhead associated with detecting errors and recovering data. More importantly, a carelessly designed protocol may degrade MacBlit's performance to an unacceptable level. However, we believed that reliability is more important than raw speed and were willing to put extra effort and time into the design to make sure that the implementation of the protocol would provide the best possible performance.

The design of WSP is based on the Reference Model of Open Systems Interconnection (OSI) proposed by the International Standards Organization (ISO). The ISO/OSI model has seven layers; each layer provides a well defined set of services to those above it. Since MacBlit is just a simple one-to-one communication system, there is no need to have any service beyond the first two layers — namely, the physical layer and the data link layer. The task of the physical layer is to transmit and receive raw bits over a communication channel. The data link layer is responsible for providing a virtually perfect channel — even when the physical connection is not error-free.
In this chapter, I will first discuss the physical layer of the protocol. Then I will focus
the discussion on the data link layer including its design and how MacBlit implemented it.
I will only discuss the implementation of WSP on the Macintosh. The UNIX side of the
same protocol was clearly described in the paper written by Scott Mulligan [2].

5.2 Physical Layer

The most popular way (as far as the non-IBM world is concerned) to connect a terminal
to a host is by a RS232 channel. RS232 is a physical protocol for transferring characters
serially beween a computer and peripheral devices (please note the differences between the
RS232 protocol and the WSP: the former is the protocol used by the physical layer, which
is a component of the WSP). There are two RS232-compatible serial ports on the
Macintosh's back panel, they can be connected either to a modem or a direct line into the
host.

The RS232 protocol is a character oriented asynchronous protocol. Every character
transferred is headed by a start bit to indicate the beginning, followed by seven or eight data
bits which make up the character body, then followed by an optional parity bit, and finally
ended with one or two stop bits. Except for the start bit, the number of bits in other fields
are changeable (they are called parameters). Moreover, flow control can also be a software
handshake or a hardware handshake. In the case of WSP's physical layer, a RS232
channel is defined to be: one start bit, eight data bits, no parity, and one stop bit. There is a
total of 10 bits in a character. Flow control is done by software handshaking with ASCII
2110 (Ctr-Q) and ASCII 2310 (Ctr-S) being designated as the XOn and XOff characters
respectively.
The physical layer is implemented by the *serial driver*, a set of routines provided by the Macintosh operating system to access a hardware device called the Serial Communication Controller (SCC) (Figure F). The serial driver is interrupt-driven. Whenever SCC receives a character or is ready to send one, it interrupts the current process and detours control to the serial driver which will then perform any necessary operation to move a character in or out of the SCC. There are many routines included in the serial driver, such as making read and write requests, inquiring about status information, and setting parameter values. The data link layer interfaces with the physical layer by calling these serial driver routines.

### 5.3 Data Link Layer

#### 5.3.1 Protocol Design

WSC's data link layer is a simple stop-and-wait protocol based on the 1-bit sliding window protocol described by Tanenbaum [1]. When one end wants to send data, it first breaks the data into smaller chunks called packets, then sends them to the other end in order. For every packet received, the receiver will check to see if the packet was affected by transmission errors. If everything is correct, the receiver will send an acknowledgement to signal the sender to send the next packet, otherwise, a negative acknowledgement is sent instead, to signal the sender to retransmit the previous packet. The stop-and-wait action is taken on the sender side. It will not send the next packet unless it knows the previous one was received correctly, as indicated by a received acknowledgement.

One goal that MacBlit must achieve is to support many opened windows at once and to have all of them updated asynchronously. Therefore, the original Tanenbaum protocol, which was designed for a single channel communication, must be modified to meet this
need. We extended it by simply adding a three bit channel number to each outgoing packet. This number defines eight virtual channels — one for the maintenance packets and seven for the seven windows. The stop-and-wait action is then applied to each channel so that when the sender is waiting for an acknowledgement from a window, it will still be able to send packets through other channels with the receiver on the other side knowing that they are different packets and displaying them on different windows.

5.3.1.1 General error and the checksum

The main task of the data link layer is to detect errors. It does so by inserting an integer number called checksum into every outgoing packet. Checksum is calculated by applying some arithmetic operations to the packet contents. The equation we chose is the following:

\[
\text{Checksum} = \text{Sum of all bytes in a packet} \mod 253_{10}
\]

This equation gives us an 8-bit number from 0 to 252. Statistically speaking, one out of 256 erroneous packets will be undetected. In other words, if the error rate is 1% and the average packet length is 30 bytes, then, on average, there will be one erroneous packet escaping from our error detection scheme in every 765,000 bytes. We think this is good enough for our purpose. A better encoding scheme such as a 16-bit CRC could have been used, but we decided not to use it because of heavy overhead problems.

5.3.1.2 Sequencing error and the sequence number

Consider the following scenario:

(1) MacBlit sends a packet to \(\mu\)MPX. The packet is correctly received. \(\mu\)MPX sends an acknowledgement back to MacBlit.
(2) The acknowledgement was damaged or lost, so that MacBlit thinks \muMPX did not receive the packet.

(3) MacBlit times out and sends the same packet again.

(4) The packet is also received correctly by \muMPX but it is duplicated.

What we need is a way to tell whether a packet has already been received. This is achieved by adding a one-bit sequence number to each packet. The sequence number will be toggled between 0 and 1 for each new packet. If the receiver receives two consecutive packets with the same sequence number, it immediately knows they are duplicated.

5.3.1.3 Out-of-Sync error and the SYNC character

Errors in transmissions may cause the receiver at either end to lose track of where the packet boundary is; these are called out-of-sync errors. Consider the following example: suppose the UNIX host sent a packet of 25 characters to MacBlit, but unfortunately, during transmission, a one-bit error changed the length field of the packet from 25 (binary 11001) to 9 (binary 01001). Now the MacBlit's receiver would read only 9 bytes instead of 25 and consider the 10th byte as the beginning of a new packet. The receiver should discover a checksum error and ask the host to resend the packet. But, since the receiver under-read 14 characters, all incoming packets will also be shifted by 14 bytes and none of them will ever be received correctly.

To recover from an out-of-sync error, WSP uses a scheme similar to the one used by the serial controller to recover from a framing error. In analogy to the stop bit, there is an 8-bit SYNC character appended at the end of each packet. The value of the SYNC character is defined to be the binary number 01010101, because it does not conflict with the
XOn/XOff characters. When the software detects an out-of-sync error, it will try to re-sync itself by discarding the currently read packet and skipping all incoming characters until a SYNC is found. The software then treats the next incoming character as the first character of a new packet.

5.3.1.4 XOn/XOff conflict and resolving method

Since the physical layer of the protocol is byte oriented, it is possible to have XOn and XOff characters stored as packet contents somewhere in a packet. Every time the receiver on one side receives an XOff character, it signals its counterpart sender to stop sending data until an XOn character is received. However, the receiver can not distinguish an XOn/XOff character which is meant to alter the transmission status from an XOn/XOff character which is meant to be part of a packet's contents. If the XOn/XOff character is indeed part of a packet, the result may be a deadlock situation and the loss of a packet - we call this situation XOn/XOff conflict.

One way to resolve the XOn/XOff conflict is to make sure that there are no XOn or XOff characters in the packet. We use the following scheme to accomplish this.

When the sender sends a packet, it examines and sends it character by character.

1. If the character is an XOn, an XOff or any character greater than ASCII 127 \text{(bit 7 is set)}, the sender splits the character into two bytes; otherwise, the character will be sent as is.

2. Put the bits 4 to 7 of the character into bits 0 to 3 of the first byte.

3. Put the bits 0 to 3 of the character into bits 0 to 3 of the second byte.
(4) Set bit 7 of both bytes to 1.

(5) Send the first byte followed by the second byte instead of the character itself.

When the receiver receives a packet, it receives and examines it character by character.

(1) If the bit 7 of the character is set, the receiver will read in an additional character.

(2) The receiver then creates a new character by putting the bits 0 to 3 of the first character into bits 4 to 7 of the new character, and bits 0 to 3 of the second character into bits 0 to 3 of the new character.

5.3.2 Formats of different packets

Figure E shows the structure of different packets. Although different packets may have different numbers of fields, they must have at least a header, a checksum and a sync field.

```
<table>
<thead>
<tr>
<th>Header</th>
<th>Command</th>
<th>Checksum</th>
<th>Sync</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Header</td>
<td>Length</td>
<td>Checksum</td>
<td>Data</td>
</tr>
<tr>
<td>(b)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Header</td>
<td>Checksum</td>
<td>Data</td>
<td>Sync</td>
</tr>
<tr>
<td>(c)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Header</td>
<td>Checksum</td>
<td></td>
<td>Sync</td>
</tr>
<tr>
<td>(d)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure E. Packet formats. (a) Maintenance packet, (b) Data packet and Library packet (c) Data1 packet (d) Acknowledgement packet
5.3.2.1 Header

The Header can be further divided into three subfields: PType, AckReq (or Ack/Nak for acknowledgement packets), and SeqNo.

5.3.2.1.1 PType

PType stands for Packet Type. It occupies bits 5, 6, and 7. It is one of the two fields which are common in all packets (the other one is the Checksum). When the software reads a packet, it first looks for the PType and then decides how to interpret the packet's contents.

Currently, there are five different types defined, as shown in Table A.

<table>
<thead>
<tr>
<th>PType Value</th>
<th>Meaning</th>
<th>ACK Required?</th>
<th>Window Dependent?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Maintenance packet</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>Library packet</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>Data packet</td>
<td>Optional</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>Data1 packet</td>
<td>Optional</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>Acknowledgement packet</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table A. Values of PType and their meanings

- Maintenance packets are used to send the window independent commands such as CreateWindow, KillWindow, Abort, etc.
- Library packets are used to send commands which access the Macintosh QuickDraw and other tool box routines.
- Data packets are mainly used to exchange character strings and escape sequences of the VT100 and TEK4014 windows.
• Data1 packets are used to send a single character. Every time the user pushes a key, a single character is sent. Keyboard events are interactive; they need to be handled as fast as possible. By defining a new type, Data1 packets can be assigned higher priority.

• Acknowledgement packets are used to signal the sender whether the last packet it sent has been received correctly. If the received packet has no error, an ACK packet is sent; otherwise, a NAK packet is sent.

The acknowledgement packet is the only kind which can never be acknowledged because the result would be an infinite loop. Since maintenance packets and library packets are sensitive to errors, they must be acknowledged.

5.3.2.1.2 AckReq

Following PType is the Acknowledgement Request bit (bit 4). It will be set to ‘1’ if acknowledgement for the outgoing packet is required by the sender. And it will be set to ‘0’ if no acknowledgement is needed. In an acknowledgement packet, since it will never be acknowledged, this field is used for specifying whether it is an acknowledgement packet or a negative acknowledgement packet. When this bit is set to ‘1’, it represents an ACK packet otherwise, it represents a NAK.

5.3.2.1.3 SeqNo

SeqNo (bit 0 to 3) is the two part sequence number. Its left most three bits is called the Channel Number. It has the same value as the window number the packet belongs to. The last remaining bit is called the Order Number which is used to keep track of the order of packets in a channel.
For an acknowledgement packet, the $SeqNo$ is the sequence number of the received packet for which the sender is sending an acknowledgement for. Whereas, for all other types of packets, the $SeqNo$ is the sequence number of the out-going packet.

### 5.3.2.2 Command Byte

*Command Byte* is used to direct $\mu$MPX to take different system maintenance actions. So far, five different commands have been defined.

**Create Window** — the user has just opened a new client window; $\mu$MPX should create a new client shell for it.

**Kill Window** — the user has just killed a client window; $\mu$MPX should destroy the corresponding client shell and kill all processes created under that shell.

**ACK On** — signal $\mu$MPX to turn on the AckReq bit on all DATA and DATA1 packets.

**ACK Off** — signal $\mu$MPX to turn off the AckReq bit on all DATA and DATA1 packets.

**Quit** — tell $\mu$MPX to kill all client shells and all child processes created under those shells, and then terminate.

### 5.3.2.3 Length

The *Length* field stores the number of bytes in the *Data* field.
5.3.2.4 Data

In a DATA1 packet the *Data* field is a single character. In a DATA and Library packet, the *Data* field may contain arbitrary information. The size of the *Data* field is limited by the maximum value the *Length* field can represent — 255.

5.3.2.5 Checksum

The *Checksum* field is an 8-bit unsigned integer which is calculated by adding together all bytes in a packet, excluding the SYNC character, and then mod the sum with the prime number 253.

5.3.2.6 Sync

The *Sync* is the sync character we mentioned before.

5.4 Implementation

The data link layer is implemented by a package called PacMan (Packet Manager). PacMan is made up of three major modules: PacRead, PacWrite, and PacMake, as illustrated in Figure F.
The module PacRead reads complete packets from the physical layer (serial driver), decodes them, and puts the contents of the DATA field in the message queue (MsgQueue) for an emulator to use. Some packets do not contain a DATA field, such as maintenance packets and acknowledgement packets; they are handled internally by PacRead. When PacRead needs to acknowledge a received packet it will send the acknowledgement itself to minimize delay. QueueInfo is a internal global data structure which connects PacRead and PacWrite together.

PacWrite is responsible for sending all kinds of packets. It repeatedly examines the ACK/NAK queue and the eight SendQueues, picks up pending packets, passes them on to the physical layer, and sends them to μMPX. There are eight SendQueues, numbered from 0 to 7, each corresponding to a virtual channel. The SendQueue[0] keeps maintenance
packets and does not correspond to any window. The other seven SendQueues correspond to the seven windows; each keeps the packets of the window with the same number.

The packets in the SendQueues are supplied by another module called PacMake. Its task is to construct packets out of the module's input parameters provided by the caller modules. PacMake is capable of making all kinds of packets except ACK/NAK packets, they are generated by PacRead.

To achieve higher efficiency, it would be necessary to have PacWrite, PacRead, and Message Router running concurrently. At first glance, this concurrence should be easy to implement, since the three modules are functionally independent of each other and they communicate only through shared memory. But unfortunately, the Macintosh was designed to be a single-user-single-tasking computer, the facilities we need to support multitasking—such as CPU scheduling and memory protection—do not exist. Multitasking must be simulated by our own efforts.

5.4.1 First Try - Interrupt

One way to simulate concurrence is to use interrupts. There is a special kind of interrupt called vertical retrace interrupt which is generated at the time when the electron beam has reached the bottom of the CRT and starts returning to the top. The time it takes to return is very short as far as human beings are concerned but it is long enough for a CPU to execute thousands of instructions. The Vertical retrace interrupt is generated 60 times per second. If we install an interrupt handler which executes the three modules, we can achieve the effect of concurrence. This was once the preferred method because a vertical retrace task uses the otherwise idle CPU time and therefore increases the utilization of the CPU.
However, a vertical retrace interrupt can occur at any moment, even in the middle of a memory compaction process. Therefore, a vertical retrace task can not allocate memory nor can it call any toolbox routine which allocates memory. Otherwise, the memory may be allocated on top of the half-moved memory and corrupt the application heap. The result of this drawback would be a great increase of complexity which conflicted our goal of having a simple design and therefore, the idea was abandoned. I hope, when MacBlit is redesigned in the future, this method will be given serious consideration again.

5.4.2 Alternative - Finite State Machine

The alternate method is very simple. We simply make PacRead into a finite state machine, whenever PacRead needs to read from the serial driver, it first tests to see whether there is any character ready for reading. If there is, it proceeds; if not, it just returns. The next time PacRead is called it will start from the point (state) where it last exited. As a result, PacRead can be called freely at any time, any place in the program, without being blocked by the delay of incoming data.
6 VT100 Emulator

This chapter addresses some of the key issues of the design and the implementation of the VT100 emulator. In general, the VT100 emulator contains two parts: a display handler and an event handler. The display handler is responsible for displaying the received messages on windows. Its tasks include parsing escape sequences, addressing cursors, updating text buffers, and displaying and erasing characters. The event handler is a set of routines which handle the user and system generated events such as mouse down in vertical and horizontal scroll bars, key down, auto key, window update, window activate, and window deactivate.

The VT100 emulator parses escape sequences by using a finite state machine. When the emulator is called, it first appends the received message to an internal buffer. The contents of the buffer are taken out by the emulator one character at a time and the finite state machine advances according to what character it reads. If an uncompleted escape sequence is reached, the parser returns and the current state is memorized. When the emulator is called again, it will start at the point where it left off but with more data in the buffer. Thus, the uncompleted escape sequence will have a better chance to be scanned successfully. As soon as a legal or illegal escape sequence is recognized, the appropriate action will be taken and the finite state machine will be reset.

6.1 Display Record

The display record is the most important data structure for the VT100 emulator. It stores not only the display text, but also all the information needed to emulate a VT100 terminal.
The following is a simplified version of the display record with most of the internal bookkeeping fields eliminated$^5$.

```c
typedef struct {
    WindowRecord window;
    char streamBuf[256];
    int streamHead;
    int streamTail;
    LineRecord *charMap;
    LineRecord *display[24];
    LineRecord *viewTop;
    LineRecord *viewBottom;
    int viewLeft;
    int viewRight;
    int state;
    int currentRow;
    int currentCol;
    int oldRow;
    int oldCol;
    int topMargin;
    int bottomMargin;
} VTDspRec;
```

Window is the window record used by the Window Manager. When the emulator calls the Window Manager routines, it will pass a pointer to the window record as a parameter.

StreamBuf is a circular queue for storing unprocessed messages.

StreamHead and streamTail are the two index pointers needed for manipulating the streamBuf.

---

$^5$ A complete display record has 42 fields.
CharMap, displayList, viewTop, viewBottom, viewLeft, and viewRight will be discussed in the next section.

State keeps the next state-number that the escape sequence parser will be in when it is called again.

CurrentRow and CurrentCol keep the current cursor position.

OldRow and OldCol are used for implementing the "save cursor" and "restore cursor" function.

TopMargin and bottomMargin store the line number of the first line and the last line of the scrollable region.

6.2 CharMap

A VT100 window has an apparent size of 24 lines by 80 characters. Internally, MacBlit keeps a much larger display buffer which can store as many as a hundred 80-character lines. To view the invisible portion of the display buffer, one must use the vertical and horizontal scroll bars. The internal buffer is a double circular linked list called CharMap (figure G). The main reason for using a double linked list is to support the following features:

(1) User scrolling - caused by clicking mouse button inside one of the scroll bars.

(2) Command scrolling - caused by receiving a line feed character or an escape sequence which scrolls the window upward or downward a line at a time.

(3) Window update.
(4) Top and bottom margin.

Figure G. CharMap data structure
A node in the linked list is a line record which has three fields:

typedef struct {
    LineRecord *next;
    LineRecord *last;
    char lineBody[80];
} LineRecord;

LineRecord *CharMap;

Each line record stores the text of one line in the lineBody field. If the line is shorter than 80 characters, the rest will be filled up by spaces. The next and last fields are pointers to the next and previous LineRecords. There is an array of 24 pointers called the DisplayList which points to 24 consecutive LineRecords in CharMap. The DisplayList defines the current drawing portion in the CharMap. For example, to draw a character at line 15 column 22, the emulator first inserts the character into

```
DisplayList[15]->lineBody[22]
```

and then draws the character at the appropriate location on the window.

Since the window is scrollable, the drawing portion of the CharMap may not be the same as the portion shown on the window. The viewing portion of the CharMap is defined by four pointers: viewTop, viewBottom, viewLeft, and viewRight; they point to the first line, the last line, the first column, and the last column which is currently visible to the user. In other words, we draw into the LineRecords pointed to by DisplayList, and display all the texts bounded by the four view pointers. Notice that viewTop and viewBottom are address pointers whereas viewLeft and viewRight are index pointers.
6.3 Scrolling

The VT100 emulator handles the user scrolling and command scrolling differently. User scrolling only changes the “viewing position” of a window over its CharMap; it does not alter the displayList pointers and only the view pointers need to be updated. When the user clicks inside the vertical scroll bar, depending on which direction the scrolling occurs, the viewTop and the viewBottom pointers are updated, one line at a time, by tracing either the next or the last pointer in the line records that they are currently pointing to. Horizontal scrolling is slightly different; since the viewLeft and viewRight are index pointers rather than address pointers, they can be updated by simple arithmetic calculations.

Command scrolling affects both the DisplayList pointers and the view pointers. When command scrolling occurs, depending on the scrolling direction, each pointer in the DisplayList, starting from the TopMargin to the BottomMargin, will point to its next or the last LineRecord. And then the viewTop and the viewBottom pointer will also be updated accordingly.

6.4 Top and bottom margins

The VT100 terminal has a feature which defines two fixed regions on the screen. The two fixed regions are defined by setting the top and bottom margins. When scrolling occurs, only the lines between the top margin and the bottom margin are affected. Since scrolling is done by shifting pointers through the next or the last link of LineRecords, if we disconnect those lines, which should not be scrolled, from the CharMap, these lines will remain at a fixed location when the window scrolls. For example, in figure G, the top and bottom margins are set at line 4 and 22 respectively, meaning that the lines 1 to 3 and
lines 23 and 24 should not be affected by scrolling and are disconnected from the CharMap. The disconnection is done by the following statements:

/* Take care of the top margin */
(1) DisplayList[1]->last->next = DisplayList[TopMargin];
(2) DisplayList[TopMargin]->last = DisplayList[1]->last;
(3) DisplayList[1]->last = NULL;
(4) DisplayList[TopMargin-1]->next = NULL;

/* Take care of the bottom margin */
(5) DisplayList[24]->next->last = Display[BottomMargin];
(6) DisplayList[BottomMargin]->next = DisplayList[24]->next;
(7) DisplayList[BottomMargin+1]->last = NULL;
(8) DisplayList[24]->next = NULL;

Notice that characters can still be drawn into the fixed regions because the DisplayList is still pointing to those lines, but the viewTop and viewBottom pointers will never point to them and thus they will not be scrolled.
7 Conclusion

7.1 What did we accomplish?

The goal of MacBlit emphasized testing the idea rather than producing an optimal product. It was designed with a view toward easy modification rather than raw speed. The throughput of the second prototype was about 300 characters (3000 baud) over a 9600 baud serial communication line. In terms of band-width utilization, it is a little over 30%. The speed is slow but not unusable and I believe that with the help of some optimization techniques, throughput over 60% can be achieved.

The Macintosh User Interface Toolbox was at first difficult to learn because of the lack of documentation, but once mastered, the result was a clean and yet flexible structure especially suitable for incremental program development. For example, the ability to enable and disable each menu item allowed me easily to ignore most of the secondary features in the first prototype and focus my attention on the main stream functions.

7.2 Looking into the future

The MacBlit prototype only implemented VT100 emulation, but since each emulator is self contained, adding others should not be too difficult. Planned future enhancements include Tek4014 vector graphic emulation, DEC's VT240 color graphic terminal emulation,\(^6\) Kermit file transfer protocol, and direct toolbox access.

---

\(^6\) This feature requires the newly introduced Macintosh II with color monitor.
With the direct toolbox access, a UNIX program could open other types of windows on the Macintosh, define its own menu and controls, draw graphics using the QuickDraw library, and give direct access to the mouse. Direct toolbox access is considered the most useful feature because it would allow UNIX programs with a complete Macintosh user interface, such as a mouse driven editor similar to MacWrite, to be developed on the UNIX host.
References


Glossaries

**Client Shell** - a UNIX shell created by μMPX upon receiving a Create Window command.

**Client Process** - a UNIX process running under a Client Shell.

**Client Window** - a window created by MacBlit on the Macintosh. It is assigned to a Client Shell and thus all output from that Client Shell will appear on the Client Window.

**Virtual Channel** - an imaginary connection between a Client Window and a Client Shell.