8-26-1986

MWTerm: a macintosh based multiple-window unix workstation

Deshler Armstrong Jr

MWTerm: A Macintosh™ Based Multiple-Window Unix™ Workstation

By

Deshler D. Armstrong Jr.

Submitted in partial fulfillment of the requirements for the Master of Science in Computer Science.

Approved by:

Guy Johnson

Name Illegible

Name Illegible

I, Deshler D. Armstrong, hereby grant permission to the Wallace Memorial Library of Rochester Institute of Technology, to reproduce my thesis entitled MWTerm: A Macintosh Based Multiple-Window Unix Workstation in whole or in part. Any reproduction will not be for commercial use or profit.

Deshler D. Armstrong

August 7, 1986

Abstract

MWTerm is a terminal emulator that runs on an Apple Macintosh computer. It provides a multiple window interface in which each window maps to a independent process on the host computer. It is designed to run in conjunction with a cooperating program on the host computer, under the Unix operating system.

This paper describes the ideas behind the user interface presented by MWTerm. The implementation of both the Macintosh and the Unix ends is described. A discussion of the lessons learned and future enhancements is provided.

Table of Contents

1.0 Introduction

The past several years have seen the coming of age of the "workstation" style of computing. This style of computing is characterized by its human interface. The user is presented with an integrated, visually oriented interface. Some of the characteristics of a workstation environment (to be discussed below) include:

- Multiple windows provide natural access to concurrency.
- Menus provide a quick, easy to learn interface to the command structure of the program or operating system.
- Advanced input devices, such as mice, provide the user with a powerful interface to windows and menus.

Windows

In the past, a concurrently executing program could not provide concurrent output. If more than one process tried to write to a conventional terminal, their outputs would become hopelessly mixed together. In addition there was no reasonable way to connect the keyboard to one process and then to another[6]. The effect of these and other problems was to relegate concurrency to various schema having one interactive process, and several background or batch processes. Alternative schemes were to "push" and "pop" processes, but these techniques suffered from many of the same limitations as the background process schema.

There was another, hardware intensive way to utilize the power of concurrency. Users could use multiple terminals, with each terminal running a separate process but on the same computer. By using this technique, a user could for example maintain an editing session, a debugger session, and any other useful utilities concurrently. The only flaw with this style of computing was its cost, both in terms of hardware and programer desk space.

In effect, a multiple window, workstation environment provides an interface similar to the multiple terminal schema mentioned above. The screen of a single terminal is divided into several regions. Each region, called a "window", corresponds to one of the terminals in the multiple terminal schema. The problem of mapping one physical keyboard into several virtual keyboards is handled by allowing only one window to be active at a time. The active window will receive any keyboard input.

By mapping the separate terminals of the multiple terminal schema into separate virtual terminals on one screen, several improvements can be made in the user interface:

- The cost is substantially lower.

- The desk space requirements are reduced.

- Having the output of all the processes in close physical proximity aids in concentration and comprehension.

- Having the I/O for several processes pass through one processor, the terminal, generates opportunities for new forms of integration.

Menus

Another problem with the conventional user environment was the use of textual commands to control programs. It has always been true that parsing natural language is a hard problem. If a user is allowed to type in commands using natural language, the command processor must be able to parse and deduce semantic content from an almost infinite variety of possible inputs. Because of this problem, users were usually required to provide input in a strongly constrained input language. Typically this input language was considered "unfriendly" and hard to learn by new users. Also input languages made it hard for new users to learn about new features in a program. One had to know the proper syntax before one could try a feature. Only by learning its complete command syntax could a user use the full functionality of a program. Finally, command interfaces tended to punish poor typing skills. If every command to the program requires a typed command, non-typists require substantially more time to accomplish a given task.

One solution to this class of problems was to provide a help command. Thus by learning the syntax of one command, a user could invoke a resource to help him/her learn about the syntax of the other commands. Often the help resource would allow the user to learn what the capabilities of the program were. There were problems with this type of solution:

- A help command does not remove the need to learn an obtuse command syntax. It simply serves as an aid in learning the command syntax.

- A help command typically generates pages of output which the user must sort through in order to find the kernel of information sought.

- A help command does not reduce the amount of typing required by command interfaces. So called hunt-and-peck typists are not aided by help commands.

A typical workstation solves these problems through the use of menus. A menu consists of a context sensitive listing of the available commands. The list of commands is context sensitive, in the sense that only commands currently appropriate are presented to the user. By moving a graphic indicator to the desired command in the listing, the user may invoke the requested functionality. Descriptive entries in the menu allow a novice user to quickly determine which commands are currently appropriate. Also a user can invoke functionality with a minimum of typing.

The Mouse

The mouse is the input device that provides a integrated interface to the user. A mouse is a device that rolls on the user's desk; usually it will have one or more buttons that the user can "click". As the user moves the mouse on his/her desk, a graphics character, called the *cursor*, moves on the screen of the terminal, controlled by the user's movements of the mouse. When the user clicks one of the mouse buttons, he/she is selecting the item on the screen which is covered by the cursor.

For example, if there is a menu on the screen, the user can select an item on the menu by moving the cursor over that item and then clicking a button on the mouse. A user can make a window on the screen the active window, the one that gets keyboard input, by moving the cursor until it lies over that window, and then clicking the mouse.

By always using the strategy of moving the cursor over some desired

object or action and then clicking the mouse button, the user is presented with a consistent, easy to learn interface.

It seems likely that a user interface based on multiple windows, menus and mice is easier to use and allows the user to make more productive use of his/her time.

1.1 The Problem

Current workstations are characterized by the power of their user interface, and by their cost per user. The cost per user of workstations is high, due to the fact that each workstation, typically being a single user machine, must duplicate the resources that are shared by several users on a conventional multi-user system. Due to this expense, it is still more cost effective to provide a central computing resource for several users, rather than giving each user a workstation.

If however, one could provide the user interface of a workstation for minimal cost, and allow users to still share a central computing resource, then a workstation style environment could be inexpensively obtained.

The goal of this thesis is to investigate one method for doing this. The technique used consists of using a inexpensive personal computer to provide a workstation style interface to a conventional computer. The personal computer chosen is the Apple Macintosh. The conventional computer is a Pyramid computer running the Unix timesharing system. A software system, entitled MWTerm, was written which ties these two systems together forming a workstation style environment.

The completed system consists of a program running on the Macintosh which manages the user interface. It accepts characters from multiple

independent processes running on the Unix system, and posts them to multiple windows on the screen of the Macintosh, each window having been mapped to one of the processes on the Unix system. It also provides a menu/mouse oriented front end, giving users access to menus to manipulate the processes on the host computer.

The Macintosh is in many respects the ideal machine for this application. It is relatively inexpensive, and just as importantly, it comes with an efficient set of routines for maintaining windows, menus, and handling mouse input. It is readily purchased and maintained, and fits comfortably on a user's desk. Its main disadvantage, as will be discussed later, is its small screen size.

1.2 The Work

The main goal of this thesis is to design and implement a special type of workstation environment. This environment is characterized by a separation of the user interface from the main computational resource. The motivation for this separation is the cost-per-user advantage already described. The questions this thesis hopes to answer are:

- Is such a distributed environment possible? Is it desirable?
- What compromises will be necessary to implement such an environment?
- Is the Macintosh well suited to this type of application? If not, what characteristics does it lack?
- Is the Unix operating system well suited to this

type of application. If not, why not?

- What sort of algorithms and protocols are needed?

1.3 Other Work

In the last few years there has been much work in the fields related to this thesis. The seminal work in a window/menu/mouse oriented user interface was done at the Xerox Palo Alto Research Center (PARC) in the early 1970's. The result of this work was the Smalltalk programing language and environment [5]. The Smalltalk environment was itself dependent on the bitmap oriented graphics algorithms, such as the BitBlt operator [4][3], that were developed at PARC. With time, the Smalltalk style environment has been implemented on other machines [6][7][16][11].

The idea of distributing a workstation environment between a large host and a local processor is not new either. The Blit graphics terminal supports a window style interface to the Unix operating system, as well as allowing interprocess communication between processes on the host computer and processes executing on the terminal [1][2].

More recently UW, a public domain terminal emulator with many of the same features as MWTerm, was written for the Macintosh computer[17]. UW provides a multiple window interface to multiple processes running on a computer under the Unix operating system, in a manner similar to MWTerm. In many ways UW is quite similar to MWTerm. Like MWTerm it is concerned wholely with providing a multiple window interface to Unix. In addition it provides different types of windows for editing and graphics. The main differences between UW and

MWTerm is that UW has a less ambitious interface, providing a rudimentary cut and paste, and no support for scrolling back through previous output in a window. Because of the simplifications in its interface, and its having been written in a compiled language, UW provides a much higher throughput than MWTerm.

Apple Computer has recently announced a software product, which implements an environment in which a Macintosh serves as a user interface front end to applications on a larger host computer, entitled MacWorkStation. MacWorkStation is distributed as a program which runs on the Macintosh, and a library that can be used to write applications on a host computer under either VMS or Unix. The program was originally developed for internal use in Apple, but is now available to the public for $1,500.

2.0 MWTerm

This chapter describes the MWTerm system. The environments in which the two parts of the system execute will be described. I then discuss the user interface and it's implementation, beginning with the communication protocol and concluding with the Unix and Macintosh ends.

MWTerm is best thought of as a system of two cooperating programs. One program executes on the Macintosh end of the system. The other program executes on the Unix end of the system. (See Figure 1.)

The Unix end of MWTerm creates children processes, each of which are mapped to a separate window on the Macintosh's screen. The standard input and output of the children processes are set up in such a way, that the processes think they are reading/writing directly to a terminal, but are, in reality reading and writing to the slave end of a pseudo tty. The parent process, controlling the master end of this pseudo tty, handles all the multiplexing between the children processes and the Macintosh end. Characters written by the children processes, are read by the parent process, and sent to the Macintosh, along with information needed to direct them to the appropriate window. Characters received from the Macintosh are written to the master end of the pseudo tty associated with the process that is currently attached to the keyboard. This process then reads them from the slave end of the same pseudo tty.

The Unix end also interprets commands to it that are embedded in the stream of characters from the Macintosh. These commands include requests to create or kill processes. Finally the Unix end embeds in the

MWTerm: System Organization

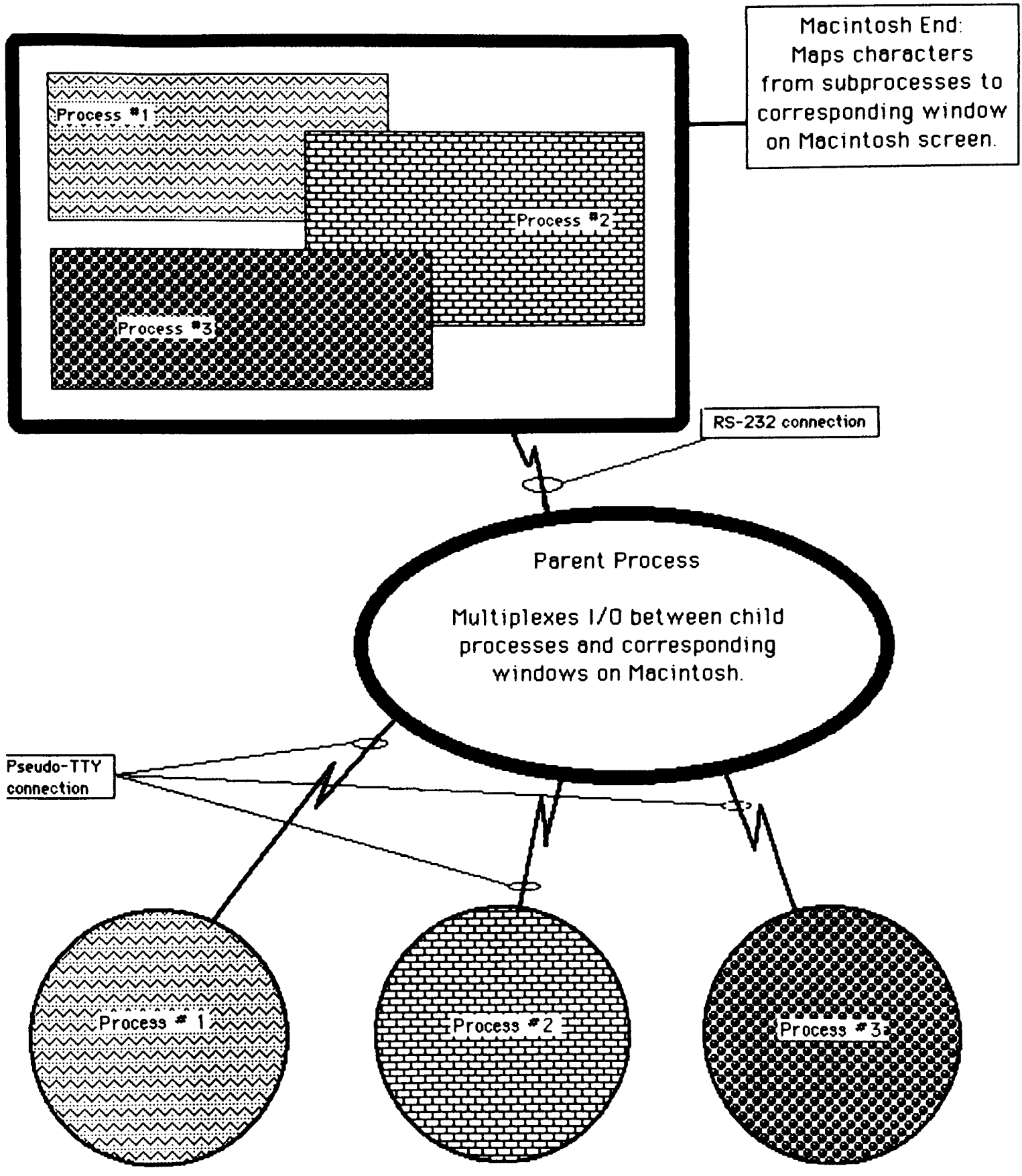

Figure 1

outgoing stream of characters to the Macintosh, status information about newly created processes.

The Macintosh end of the system is responsible for presenting the user interface. Displaying characters in the separate windows, accepting user input from the keyboard, and presenting menus are tasks handled by the Macintosh end.

The Macintosh end is presented with a stream of characters from the Unix host. The stream of characters consists mainly of characters from the children processes on the Unix host. The stream also contains status information inserted into the stream by the parent process. Mostly this status information identifies the source of the characters which follow it in the stream.

2.1 The Macintosh Environment

This section describes the environment of the Macintosh end of MWTerm, in particular the language used and a description of the support provided by the routines built into the Macintosh.

The Macintosh end of the system is implemented in the MacForth™ language. MacForth is a implementation of Forth-83, with numerous extensions and optimizations for the Macintosh[12].

Forth is a language which attempts to provide low level access to a computer's hardware, while allowing the programer to express algorithms in a high level, structured manner[14]. In general Forth systems successfully provide low level access to a computer, providing only the minimum tools needed to use a machine's features. Forth programs "compile" to a threaded dictionary of subroutines, where each subroutine is defined as a list of pointers to other subroutines, or pointers to machine

code. (See Figure 2.) The Forth kernel consists of a simple interpreter which interprets subroutines defined in the dictionary. Since interpretation consists solely of following a list of pointers, Forth interpreters are substantially faster than conventional interpreters. Forth interpreters can approach the speed of compiled languages, especially if the programs do not nest words too deeply, and if often repeated segments of code are programed in machine language.

MacForth is based on a multi-tasking Forth system, known as Multi-Forth™, however the multi-tasking features, contrary to initial advertising, were never enabled in MacForth. This results in a system which has some of the complexity of a multi-tasking system, without the advantages of multi-tasking.

Another characteristic of MacForth is its attempt to simplify the Macintosh user interface. MacForth provides a large number of routines which correspond to the Macintosh's built in user interface routines. However, these routines differ from the Macintosh's routines. They attempt to provide a much higher level interface than the Macintosh's built-in routines. Some of these differences are not clearly documented. Assumptions concerning these routines are embedded throughout the MacForth system. Finally, some of these routines deviate from the user interface provided by the corresponding Macintosh routines.

Underneath MacForth lies the "Macintosh toolbox" [7][9][10], which consists of a library of routines which are mostly embedded in the Macintosh's ROM. These routines are separated by functionality into different "managers". They include:

- Window Manager: These routines allow the

A simple FORTH definition

```
square-it    ( a simple word that squares the input number )
   dup  *

two-squared    ( a dumb program that returns the value 4 )
   2   square-it
```

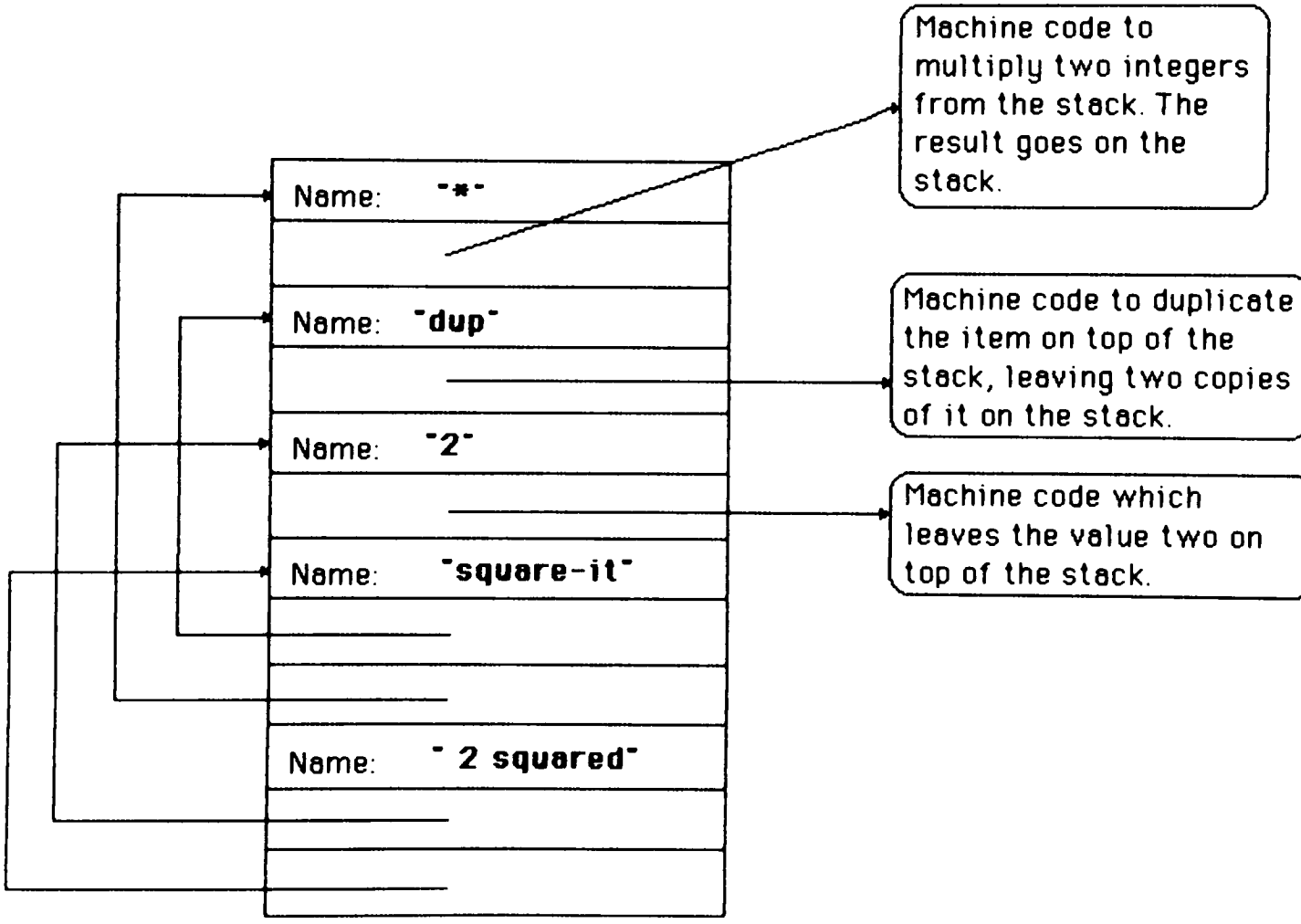

Figure 2

programer to create and manipulate pre-defined windows on the screen. When creating windows, the programmer can specify whether the user will be allowed to change the size of the windows, move the windows around on the screen, or delete the windows. In addition many other characteristics of the windows can be set (size, title, other windows to overlap, etc.) The window manager also provides support for dealing with changes in the windows. When the user causes text in a window to become covered or uncovered, the window manager helps notify the program, and provides tools for dealing with these events.

- Menu Manager: These routines allow the programer to determine when the user has selected a menu item. Dropping down the menu and highlighting the user's selection are handled automatically by the menu manager. The programmer can dynamically change the appearance of the menus.

- Event Manager: This is the core of the programer's interface to the Macintosh's toolbox. Whenever an external event occurs, the event manager puts an event record in a queue of important events, known as the "event queue." Some of the external events handled by the event manger include user keystrokes, user selection of menu items, user manipulation of windows, etc. In general the event manager is the programer's interface to the user's interactions with the Macintosh user interface. One of the strengths of the event manager is that is allows the programer to deal gracefully with multiple, asynchronous events, such as those that occur in a multiple window environment.

- TEdit: This is a library of routines for editing text. These routines allow a program to store text, and combined with the Control and Window managers, scroll through text. TEdit also provides some the

support needed to cut and paste text.

- Other managers include the Font Manager, Memory manager, Control manager (handles the scroll bars, buttons, etc.), Device manager, and the Vertical Retrace manager (manages code to be executed during the vertical retrace.)

2.2 The Unix Environment

This section describes the aspects of the Unix environment which are relevent to the Unix end of MWTerm, in particular, the ability to allow a single process to control I/O to and from several independent children process, and the ability of the parent process to create and destroy the children processes.

In particular, the BSD 4.2 mechanism known as a "pseudo tty" is used. Pseudo ttys are virtual I/O devices that provide an interface identical to that of a terminal. A process can open the "master" side of a pseudo tty, and then read and write to it. Any process which opens the "slave" side, will upon reading from it, read the characters written into the master side. Any characters written into the slave side, will appear on the master side for reading. Unlike a Unix pipe, a process reading and writing the slave side of a pseudo tty, can use the same I/O operations that are available when reading and writing to a terminal. A pseudo tty simulates a real tty to the point of allowing the slave side to use RAW mode I/O and do other terminal specific operations. A Unix pipe will not support the types of I/O specific to terminals; a Unix pipe is just another type of sequential device supported by Unix.

Use is also made of the BSD 4.2 Unix "select" function. This function takes as arguments a list of I/O descriptors and returns a list of those

descriptors which have an I/O operation pending. In effect the select function does a poll of all the I/O descriptors of interest, providing a convenient way for a program to respond to multiple asyncronous I/O streams.

The Unix end of MWTerm was developed on the Graduate Computer Science department Pyramid minicomputer. The Pyramid operating system (called OSx) supports both the BSD 4.2 Unix and the AT&T system V Unix programing environments. The BSD 4.2 environment was used. The program is written in the C language.

Attempts were made to write portable C code, and to adhere to the portable aspects of BSD 4.2 Unix. The only aspect of the code that is known to be BSD 4.2 specific is the use of pseudo ttys and the use of the "select" system function.

It was also assumed that the user would have sufficient privileges to read and write pseudo ttys. On some machines, especially student machines, users may not have sufficient privileges to use pseudo ttys.

Due to an apparent bug in older versions of the Pyramid operating system, version OSx64 or higher of OSx should be used.

2.3 The User Interface

In this section we describe the interface that MWTerm presents to the user.

The MWTerm user interface is the standard Macintosh user interface with a few exceptions, described later. The Macintosh's toolbox routines are designed to encourage the design of applications which follow a consistent set of user interface guidelines. These guidelines guarantee that once a user

learns a few simple rules, those rules will apply to most applications which run on the Macintosh.

MWTerm can maintain several windows which are displayed simultaneously. These windows are all the standard document window, with a close box, title bar, size box and a vertical scroll bar:

- The close box is a box in the upper left of a window. Clicking the mouse in the close box, causes the window to be deleted or closed.

- The title bar is a region which displays a string, the title, associated with a window. The title bar is also where a user grabs a window to move it about on the screen.

- The size box is a box in the lower right of a window. By clicking in the size box, the user can "grab" that corner of the window, and stretch or compress it, thereby changing the window's dimensions.

- Scroll bars are regions on the side of windows. By clicking in the top or bottom of scroll bars, the user can cause the contents of the window to scroll up or down respectively.

There are also guidelines concerning which window is "active", and how windows are allowed to overlap each other. Only one window can be active at a time, this is the window to which all keystrokes apply. Also, menu selections which apply to a window, generally apply to the active window. The active window is indicated by the presence of horizontal lines in its title bar. Also only the active window has a close box. Finally the active window is not overlapped by other windows. A inactive window can

be made the active window by clicking the mouse anywhere inside it.

There are many other aspects of the user interface to MWTerm, but the above are the main features that the user would normally encounter.

The semantics of the user interface are generally the same of those of the standard Macintosh user interface. In MWTerm, each window corresponds to a separate subprocess on the Unix host. The characters displayed in a window are the characters emitted by the corresponding Unix subprocess. Only the Unix subprocess corresponding to the active window can read the keyboard. Any other process will block on attempting to read until made active. However all the subprocesses can write simultaneously, and the characters from all the processes will appear in the appropriate window. It is not defined which window will get its characters before the other windows, being dependent on the order in which "select" picks subprocess to read.

The vertical scroll bar (there is no horizontal scroll bar) allows the user to scroll upwards, letting the user view previous output from the subprocess associated with the window being scrolled. Only the active window can be scrolled, although text being displayed on the other windows can be made available by selecting the desired window and then scrolling upwards in that window.

2.4 The Communication Protocol

This section describes the communication protocol that is used by the two programs in the MWTerm system. Also discussed are the performance implications of the protocol used and the merits of several other potential communication protocols.

Several styles of communication protocol were considered during the

design of MWTerm.

One protocol scheme is to use one bidirectional communications line and to embed control information in the stream of characters flowing over the control line. Commands and status information are asynchronously mixed in with text flowing from the host to the windows, or from the Macintosh to the active subprocess. One can further simplify this protocol by restricting the command and status information to the printable ASCII character set. This substantially eases debugging the system, since the character stream can be manually read, interpreted and simulated. This is the protocol used for this thesis.

The commands and status information are always preceded by an "escape character", a character which means that the character which follows is to be interpreted as special. For this implementation the escape character is the percent sign ("%"). A literal percent sign is passed through the system by converting it to two percent signs. Following the percent sign is a one letter code identifying the command or status information. Following the command or status information there could be a character of optional information, typically an ASCII digit associated used to identify a subprocess on the host.

- "%%": interpret this sequence as a single percent sign ("%".)

- "%A#": Message from the Macintosh to the host, please make subprocess number #, the new current active subprocess. Here "#" represents an ASCII digit between 0 - 9 which uniquely identifies one of the 10 subprocesses.

- "%I#": Message from the host to the Macintosh, here

is the identification number for the new shell requested previously.

- "%K#": Request from the Macintosh to the host, please kill process number "#", where "#" is the identification number associated with process to be killed.

- "%N": Message from the Macintosh to the host requesting the creation of a new subprocess.

- "%S#": Status information from the host. The characters following this sequence are from subprocess number "#". This is the sequence used by the Macintosh end to determine in which window to post characters.

This scheme proved to be straight forward to implement and was well suited for debugging the system. The major impact of the protocol on performance is due to the frequency of the "%S#" command sequence. The other command sequences occur so infrequently that they have little impact. But when more than one subprocess is generating output at the same time, each character tends to come from a different subprocess than the preceding character. Under these circumstances, each character to be displayed is preceded by a three character sequence effectively decreasing throughput by a factor of four. Also, additional processing is required on the Macintosh end to parse the command sequence and direct the subprocess output to the correct window.

This tendency for each character to come from a different subprocess is a result of the manner in which OSx doles out characters from the subprocesses. It is unknown if this behavior is common to all Unix implementations, or if it is specific to the Pyramid implementation of Unix.

In operating systems where each character I/O request generates an interrupt, it is reasonable to expect that multiple subprocess writing simultaneously will tend to each contribute one character to the output stream. This result speaks strongly for the desirability of a more sophisticated, packet oriented transmission scheme to avoid the overhead of constantly changing the output window.

There were several other styles of protocols that were considered. One style was to implement a packet oriented protocol between the Macintosh and the Unix host. This protocol has several advantages:

- Checksums could be easily implemented. These checksums could be used to provided transparent error recovery in both directions.

- Since the only order or display requirement is that characters destined for the same window be posted in the correct order, a modified sliding window protocol could do unusually well.

- A packet protocol could be easily extended to support file transfer between the Macintosh and the host computer.

- By labeling each packet with the name of the source, or equivalently the destination, delivery might be simplified.

- By collecting characters with the same destination into packets, rather than just blindly multiplexing characters, substantial increases in effective throughput should be possible.

The main disadvantage in this style of protocol lies in the time and effort to efficiently implement, and debug, communicating systems which

use it. It was decided to not use this style of protocol because the main goal of this thesis was to implement a prototype system, not to develop a commercial grade system.

Another style of protocol is to implement communication between the systems using multiple communication lines. For example, one could connect both of the Macintosh's serial ports to the host computer. One serial line would be used to send control information, such as the destination of characters being transmitted on the other serial line, or checksums.

If one were content to have only two windows, each serial line could be connected directly to the host, without any special program on the host. The Macintosh, would simply provide two windows, and serve as a terminal emulator, posting characters from one serial line to one window, and characters from the other serial line to the other window.

2.5 The Unix implementation and algorithms

This section discusses the implementation of the Unix end of MWTerm. The Unix end is viewed as having a number of distinct sections. Each section is described, both conceptualy and in terms of the main subroutines involved.

The Unix end of MWTerm consists of three main steps. They are:

- Open a pseudo-tty pair and establish a subprocess which reads and writes to the slave side of the pseudo-tty as STDIN, STDOUT and STDERR.

- Use the "select" function to read characters from the subprocesses and the Macintosh. Pass characters

from the Macintosh to the subprocess associated with the active window. Pass characters from the subprocesses to the Macintosh, preceded if needed by a header sequence identifying the source.

- Recognize and parse commands from the Macintosh end. Based on the command either create, kill, or modify the status of subprocesses.

The first part concerns itself with establishing a suitable pseudo-tty pair and creating a subprocess which thinks the pseudo-tty is really a user terminal. It was expected that this part would be relatively straight forward to implement. In reality, this part constituted almost half of the work in the project.

The implementation of pseudo-tty code and the subprocess creation follows standard Unix conventions. The subroutine *open_child* first gets a pseudo-tty from the function *get_pty*. (See Figure 3.)

The pseudo-tty is obtained by building strings with the names of potential pseudo-ttys, and then testing to see if the name is the name of an accessible pseudo-tty. If the pseudo-tty is accessible, both ends are opened as another test. If the pseudo-tty passes these tests, then its name is passed back as a valid, accessible pseudo-tty.

Open_child then initializes an entry in the data structure used to track subprocesses. After some IOCTL calls to set various characteristics of the pseudo-ttys, a call to FORK is made. The child end of the fork makes some more calls to IOCTL to set the characteristics of its end of the pseudo-tty, and then it uses the DUP2 function to set its standard input, output and error to the slave end of the pseudo-tty. Once all the pseudo-tty characteristics are established, and the standard I/O is set to the

MWTerm Structure: Unix end

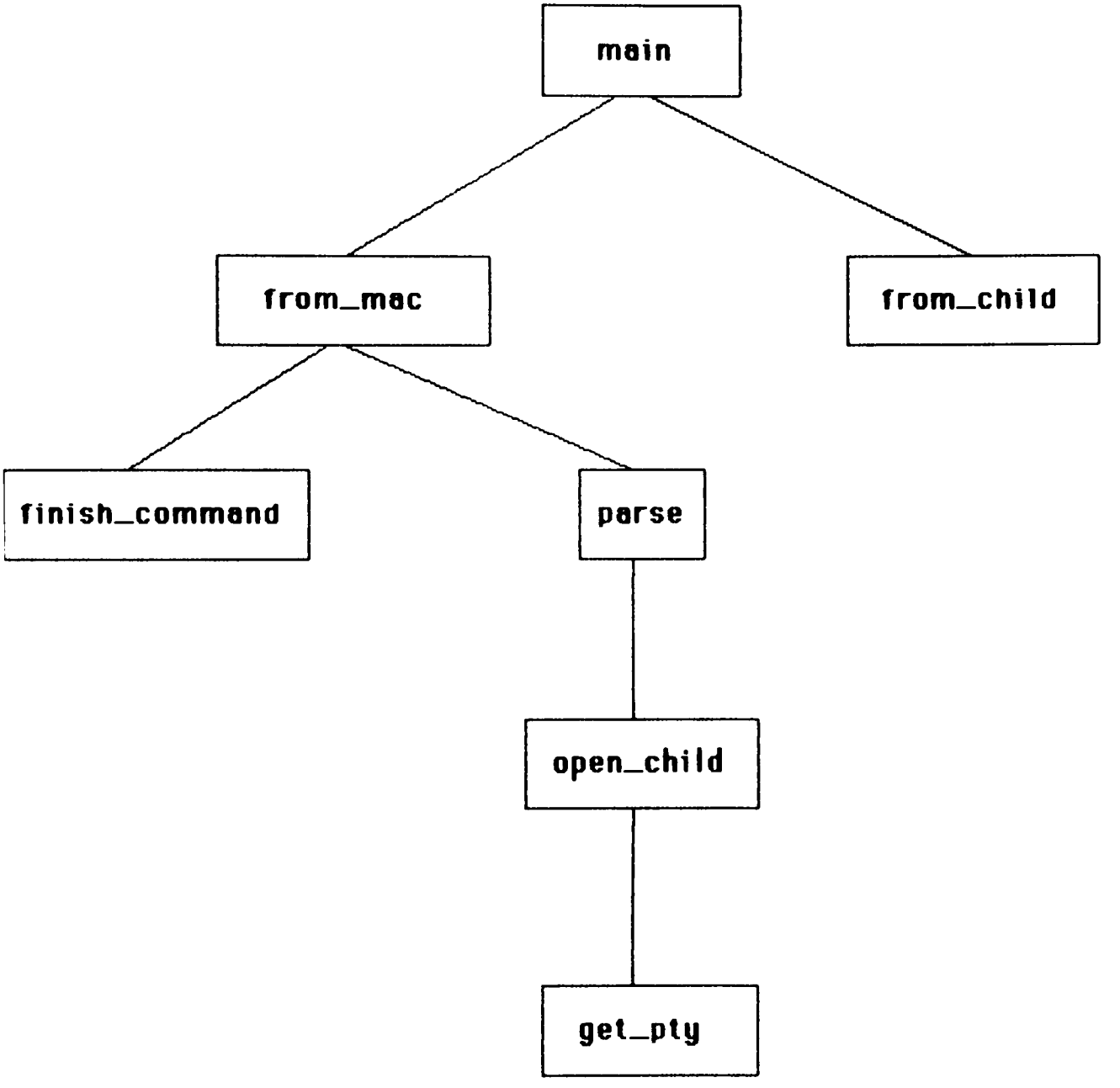

Figure 3

pseudo-tty, the subprocess calls EXECL to invoke the CSH. This causes the subprocess to begin executing the CSH, with all the I/O still directed to the pseudo-tty.

From this point on, the subprocess will execute the CSH, but all the I/O to and from the CSH will be via the pseudo-tty established before the call to EXECL.

This paradigm of forking a subprocess which establishes some sort of I/O redirection, and then using EXECL to invoke some program to use the modified I/O units, is a standard idiom in the Unix world.

The parent end of the FORK uses some IOCTL calls to establish the characteristics of the master end of the pseudo-tty. Finally it does a SLEEP(3), giving the subprocess three seconds to fork and execute, and then it does a write to the master end of the pseudo-tty. This write will block until the subprocess does the corresponding read which it does just before the EXECL. The effect of this write/read is to insure that both processes are synchronized. Whichever process is "ahead" of the other will block until the other one catches up. By keeping both processes synchronized, we can insure that both ends of the pseudo-tty are in a known state before the CSH begins execution.

The select function is invoked in the main procedure. The select function accepts as input several bit masks, each bit in the bit mask corresponding to a I/O descriptor. These bit vectors are maintained globally and are modified by the routines which create and kill children. The basic control flow of the main procedure is to call select to determine if there are any characters to read, either from the Macintosh, or from one of the subprocesses. If there is a character available from the Macintosh, then

the character is read and passed on to different routines, depending on the state of the parser. One routine just passes the character to the current active subprocess by writing to the appropriate pseudo tty. Other routines either change the state of the parser to reflect more information about the incoming command sequence, or if the command sequence is complete, execute the requested command.

The third part of the Unix implementation consists of the routines which recognize the incoming command sequences. The recognizer is a simple finite state machine. When a character arrives from the Macintosh, the recognizer changes state based on its current state and the value of incoming character. In practice, there are three types of states. In the NOP state, the recognizer is not in the middle of a command sequence. All characters are passed through to the subprocess, except for the escape character "%", which puts the state machine in the "GOT_ESCAPE" state. Once in this state, the recognizer looks at the next character from the Macintosh and compares it to the command characters "%,A,S,I,K,S,N" described above. If one of these characters is detected, then the recognizer either executes the requested command, or if another character (i.e. a process identification number) is still needed, it goes into a command character dependent state such as " got kill_subprocess_command but still need to know which one to kill". If more input is needed, the next character from the Macintosh will be accepted as the needed input. Notice that due to the lack of error correction, a garbled character here could have far ranging consequences. Although it is less likely, there is no protection against a complete command sequence being synthesized by line noise.

This simple style of parsing proved to be fast enough to keep up with

characters from the Macintosh and was sufficiently reliable to support testing the MWTerm system.

The main routine of this recognizer is FROM_MAC which invokes the routines PARSE and FINISH_COMMAND.

2.6 Macintosh implementation and algorithms

In this section I discuss the implementation of the Macintosh end of the MWTerm system. This includes the relevent aspects of the both the Macintosh toolbox and the MacForth routines used. Finally some of the routines in the Macintosh end of MWTerm are discussed.

The Macintosh end of MWTerm is written in the standard Forth style, as a series of vocabularies. Each vocabulary provides an abstraction for some utility or data type.

The vocabulary used most extensively contains the words used to manipulate the windows on the screen. MWTerm maintains a list of ten windows which can be mapped to subprocesses on the host computer. The window vocabulary contains words for allocating these windows, searching for the window associated with a given host subprocess, and deallocating windows, among others utilities.

The user interface is dominated by code devoted to handling windows. Windows must be created, kept track of, moved, resized, scrolled within, made active and inactive, refreshed if areas become uncovered, and destroyed. When the characters coming from the host start coming from a different subprocess, the stream of characters must be directed to the appropriate window. When a user clicks the mouse within a window, the currently active window must be deactivated, the window clicked within

must be made active, the window frames must be redrawn to reflect the window's new status, and the proper command sequence must be constructed and sent to the host end of MWTerm.

These are just the sort of actions that the Macintosh toolbox routines are written to support. It turned out however, that the window support in MacForth was not so general.

For example, in the Macintosh toolbox, when the user clicks in a inactive window, two different events are returned by the event manager. The first event is a "deactivate" event, informing the program that the currently active window should be deactivated. The program must then call the routines which will redraw the deactivated window with the proper visual cues to indicate that is has been deactivated. The event manager will then return an "activate" event, informing the program that the user has tried to activate the window he/she clicked in. It is up to the application program to handle these events as it sees fit. In general there is a published standard and a set of utility routines, supporting a standard way to handle user actions.

MacForth however deals with this type of event differently. When the user clicks in a window to activate it, MacForth terminates execution of the currently executing word, or program, and begins execution of a word set aside for handling activate events for that specific window. Thus when a window is created in MacForth, the programer usually declares a word to be executed when that window is activated. When the user tries to activate a window, the local context of any program running before the user activated that window is lost. It is clear that in a environment where a stream of characters is constantly being parsed from the host, there is no

reasonable way to deal with asynchronously losing the current program's context , even though we can pick the program which will be executed to handle the activate event . It's worth noting that a similar thing happens for the deactivate event generated when the user clicks in a window.

The Macintosh end of MWTerm deals with this specific problem by "stealing" the activate events before the MacForth kernel can get them.

This specific problem is indicative of a class of problems which dominated the Macintosh implementation:

- The TEdit record is the standard Macintosh data structure for managing text. The TEdit manager supports text display, scrolling, cutting and pasting, and a "history" of characters which have scrolled off the screen. The TEdit manager also supports refreshing text which is uncovered by a moved or deleted window. The MacForth support for the TEdit manager does not use the standard TEdit record, rather MacForth uses a different data structure called a TEfield. Since MacForth's handling of activate events was disabled, the associated TEfield support was also disabled. Thus the TEdit manager was used to support the MacForth style TEfield rather than the standard TErecord. Although this was possible, it proved to be neither elegant nor efficient.

  Similar problems occurred with MacForth's handling of controls, such as the vertical scroll bar used to scroll through text.

- The Macintosh provides support for certain predefined types of windows. The most common type is known as a text window. For some reason, MacForth redefined the text window with its own routines, some of which deviate from the behavior defined in the Macintosh user interface standard.

The code to deal with these issues is isolated in the sets of words which deal with the windows, controls and TEdit.

The algorithm used by the Macintosh end of MWTerm is similar to the algorithm used on the host end. The main program loops continously, looking for input from the host end, or events from the user interface. (See Figure 4.) If a character from the host is detected, it is passed on to a suite of routines known as the protocol words which check for command sequences from the host. If no command sequence is detected, the characters are passed to the TEdit manager, to post to the correct window, and save in the TErecord.

If a command sequence is detected, then it is parsed in the same manner as on the host. A simple finite state machine is driven by the current state and the input character, in the same manner as the host end of MWTerm.

The most common command sequence recognized from the host is the one which changes the output window for the stream of characters following it. This command sequence results in a global variable, "OUT.WINDOW", being changed to point to the new output window. The current value of OUT.WINDOW is given to the TEdit manager each time a character is to be posted to a window.

The other concern of the main loop is dealing with events caused by user interactions with the user interface. Much of the current code is devoted to dealing with maintaining the windows.

If the users uncovers a portion of a window, then an "update event" is generated by the event manager. The Macintosh end of MWTerm must then redraw the newly exposed parts of the affected window(s). A

MWTerm Structure: The Macintosh end

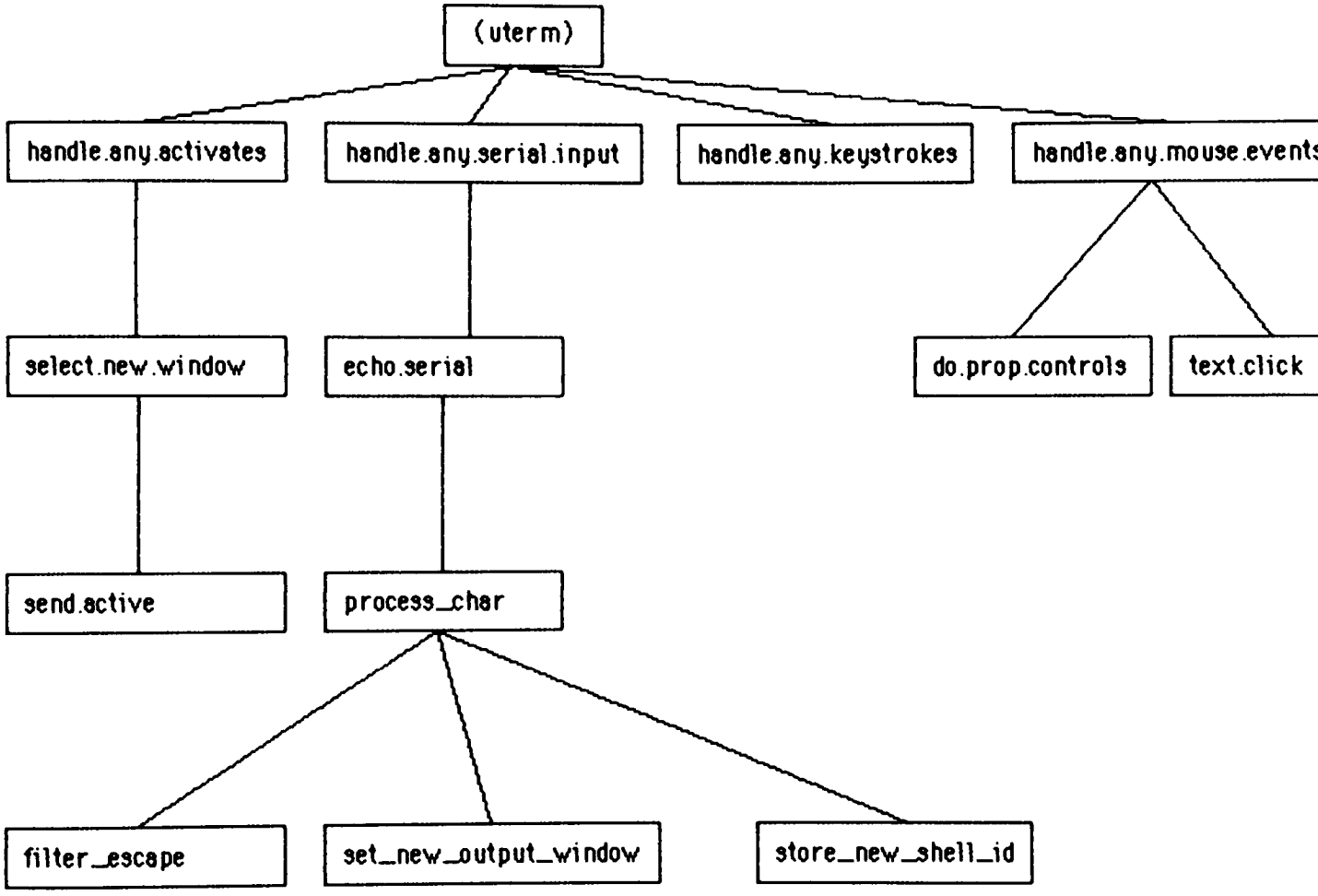

Figure 4

different set of actions are needed to redraw the text, the window boundaries, the scroll bars, and the size box. All of these separate entities need to be checked and perhaps redrawn. The main routines concerned with this are UPDATE.DISPLAY, UPDATE.TERECTS and UPDATE.TEXT.

If the user clicks on a window which is not the active window, a deactivate event for the currently active window, followed by an activate event for the window being made active, is generated. Both of these events are detected by the routine HANDLE.ANY.ACTIVATES. This is the routine which "steals" activate event from MacForth, and handles them itself. If an activate/deactivate event is detected, it is passed to SERVICE.ACTIVATE, which determines which type of event it is, and passes it on to either SELECT.NEW.WINDOW or DESELECT.OLD.WINDOW. These two routines interact with the window, control and TEdit managers to enable or disable the visual cues associated with the active window. Also SELECT.NEW.WINDOW sends a command sequence to the host computer, informing it that all future keystrokes are to be directed to the subprocess associated with the newly activated window.

If the user clicks the mouse button, that generates another type of event, known as a MOUSE.DOWN.EVENT When it detects a MOUSE.DOWN.EVENT MWTerm goes through a process of elimination to determine if the user was really trying to do something. First MWTerm checks to see if the cursor was in the active window. If the cursor was outside the active window, but the user was clicking in a menu, or activating another window, some other event than a MOUSE.DOWN.EVENT would have been generated. Thus if the user was outside the active window and a MOUSE.DOWN.EVENT was generated, the user was clicking

empty space, and MWTerm ignores the event. If the cursor was inside the active window, MWTerm checks to see if the mouse was clicked in the scroll bar. If it was, then DO.PROP.CONTROLS is invoked to handle the requested scrolling. If the user was not scrolling, then he/she must be trying to manipulate the text via the cut and paste action. In this case TEXT.CLICK is invoked to handle this request.

There are other routines which serve minor roles in providing the user interface. DO.HOUSEKEEPING handles repetitive tasks which need to be done regularly, such as examining the mouse position and adjusting the cursor image accordingly. The cursor changes depending on whether it is within the active window.

DECLARE.MENU.ACTION is invoked whenever the user pulls down the MWTerm menu. Currently this menu can only add a new window to the screen, which is done via the routine NEW_WINDOW_REQUEST. This routine adds a new window to the window list, and sends to the host the command sequence which request a new shell be created. Notice that when this happens, a flurry of deactivate/activate and update events are generated by the event manager.

2.7 Building & Installing the Software

The MWTerm system consists of three source modules. On the Unix end there are two files, mwterm.c and mwterm.h. Since the system is built with a single compile, there is no make file. To build the Unix end, the user should simply insure the mwterm.h is accessible, and issue the command:

cc mwterm.c -o mwterm

This will compile the program, and create an executable file named mwterm. To execute the Unix end of MWTerm, simply insure that the executable file MWTerm is in the user's path, and issue the command "mwterm".

The Macintosh end of MWTerm, in typical Forth fashion, consists of a single "blocks" file. This is a file in which the Forth code is formatted into logical blocks of 16 lines of 24 characters. Generally each Forth procedure, known as a "word", fits in one block. Forth words are located in the file by stating which block number they are located in.

Although the Forth code written for this thesis all resides in one file, use is made of some other utilities in other block files. The file "toolbox" contains a vocabulary of words which provide access to the defining word TOOLBOX. This word allows MWTerm to access the Macintosh toolbox routines directly, rather than going through the interfaces provided by MacForth. This utility was used extensively to circumvent some of the interfaces provided by MacForth. Use was also made of the utilities in a block file named "text edit". These utilities provided support for some of the TEdit routines, as well as the access to the MacForth TEfield data type.

To compile and load the Macintosh end of MWTerm, insure that the files "MWTerm", "toolbox" and "text edit" are all on the same disk, and then double click on the file "MWTerm". This will begin execution of MacForth, with "MWTerm" as its input file. After MacForth has started, it will give the "Ok" prompt, type "12 load" . This will load and compile the entire MWTerm system, a package at a time. Block 12, known as the "table of contents", serves as a directory of how the blocks are organized.

Once MWTerm is loaded and compiled, the MWTerm menu will appear in the menu bar, MWTerm is ready to run. First establish the baud rate by issuing the "300 baud" command at the "Ok" prompt. Use some number other than 300 for a different baud rate. Then start MWTerm by issuing the command "MWTerm". A MWTerm window will appear, providing a single terminal interface to your modem or computer connection. Use this connection to login to the Unix host. Once logged in to the Unix host, issue the "MWTerm" command as described in the section of this document on installing the Unix end of MWTerm.

3.0 Discussion

In this chapter I will discuss various conclusions that can be reached concerning MWTerm. First the usability of the system is described, both in terms of performance (throughput), and the quality of the user interface. Then I will discuss the lessons learned and what enhancements seem desirable.

3.1 System Performance

The current throughput of the system is quite low. Effectively, the system runs at about 300 baud. Other aspects of the system performance range from acceptable to quite good.

On the Unix end, there doesn't seem to be any major performance problem. At speeds up to 1200 baud, the Unix end did not seem to have any problems accepting characters from the Macintosh, or passing characters through from the subprocesses. The main performance limitation on the Unix end probably lies in the character-at-a-time style I/O it is doing with the Macintosh. There is no indication that an unacceptable amount of time is being lost during the process of parsing the input stream.

The Macintosh end is the real bottleneck in the system. There seem to be several sources of the performance loss.

- Slowness of the MacForth interpreter. Despite the fact that the MacForth interpreter is fast for an interpreter, it is still an interpreter. Part of the

philosophy of Forth is that the Forth interpreter is fast enough for most sections of code, but that time critical sections will often need to be recoded in assembler. There are sections of MWTerm that would probably benefit from being recoded in assembler.

- In an attempt to build a well structured system on the Macintosh end, several layers of code were built on top of each other. This tends to slow down the MacForth interpreter even more. In effect,the "thread" that the interpreter must follow to find code to execute became longer.

- Each character is posted a character-at-a-time to the current output window. This is done through a Macintosh toolbox routine (TEKEY) which is intended to accept keystrokes, rather than a stream of characters. This routine is probably one of the big bottlenecks on the Macintosh end.

- Dealing with unwanted features of MacForth. Code which circumvents unwanted MacForth features has to be executed often, resulting in more overhead.

- Serial I/O is done via polling, rather than with I/O interrupts.

3.2 Effectiveness of User Interface

Because of the performance problems, it is difficult to evaluate the user interface. Probably the best that can be said is that it shows promise. As implemented now, the MWTerm is effectively useless for any real work; it is too slow in terms of throughput and too limited. It needs to provide at least 1200 baud throughput, probably it should provide more like 2400

baud total throughput (~600 baud spread between four windows.)

The style of the user interface seems quite useful. The same style interface has been tested using other systems, such as the similar, but with better performance, UW system written by John Brunner. Using UW shows that the performance problems can be overcome with some compromises in the interface, and that the resulting system can be quite useful. Other systems such as the Vaxstation software environment on the DEC Vaxstation also reconfirm that utility of the multiple window/multiple process methodology.

MWTerm does not support cutting and pasting between windows, a feature which is also lacking in most workstation environments. Experience with the limited cut and paste feature of UW, and the powerful cut and paste available with the other terminal emulators for the Macintosh VersaTerm and MacTerminal, indicates that cutting and pasting can significantly enhance the power of a workstation environment.

It also became clear that a simple glass-tty interface is unacceptable. To be useful, MWTerm will need at least an editing terminal style of window. It seems likely that a graphics capable window would greatly increase the functionality of the system.

Finally, it seems likely that the Macintosh screen is too small to support more than two or three simultaneous windows. (See Figure 5.)

3.3 Implementation Lessons

The current implementation of MWTerm has confirmed several observations.

MWTerm: Exemplary Session

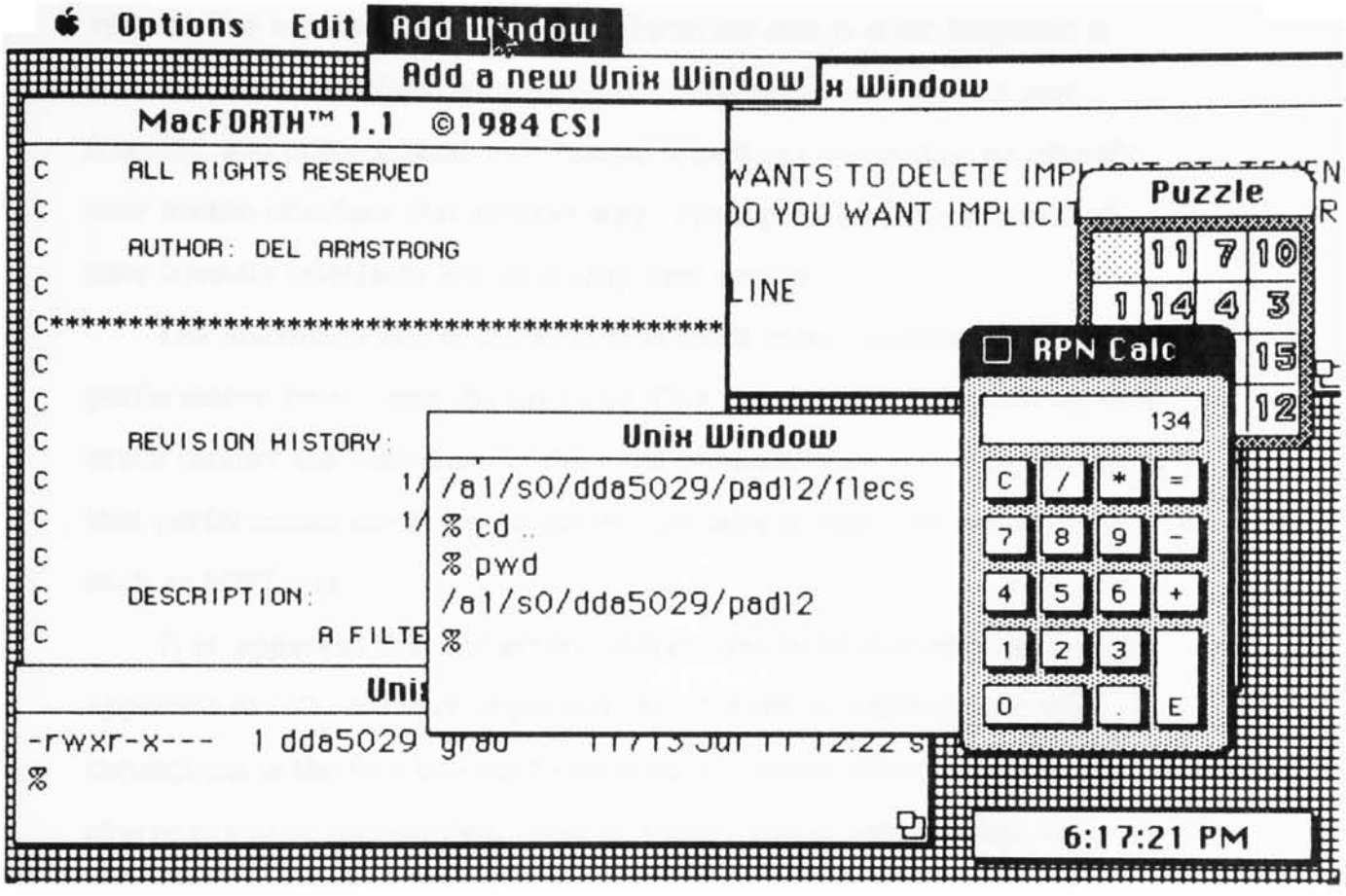

Figure 5

The Macintosh end of the system must be implemented with care. Flaws in the user interface detract seriously from the utility of the entire system. One aspect of a user friendly interface that is often forgotten is that the entire interface must be friendly. To have some aspects user friendly, and other aspects user hostile is perhaps worse than an entirely user hostile interface. Put another way, incomplete implementations of user friendly interfaces are unusually user hostile.

The Macintosh end of the system is much more sensitive to performance issues than the Unix end. This seems natural considering how much smaller the Macintosh is. Personal computers are not yet so powerful that performance can be squandered especially in real time applications such as MWTerm.

It is apparent that the system suffers due to its character-at-a-time approach to I/O. Another important impediment to achieving a useful throughput is the fact that each character displayed, often requires four characters to be transmitted, three of which have to pass through the parsing algorithms.

Clearly though, the biggest lesson has to do with the choice of implementation language and systems. At the time this thesis was first proposed and initial work was begun the only languages available for the Macintosh were Basic, Assembler (a beta release), and MacForth. In retrospect it is clear that Assembler would have been a better choice, since the problems with MacForth were its slowness due to being interpreted and its overly abstract interface to the Macintosh toolbox.

It also seems that this thesis would have benefited in the long run if the Macintosh had been allowed to mature before the project was begun.

Initial design decisions, such as the implementation language, were made shortly after the introduction of the Macintosh. Once these decisions were made, it was extremely difficult to reverse them as better choices presented themselves.

3.4 Future Enhancements

This implementation served well as an initial prototype to prove the feasibility of this type of interface. The next stage in the development of MWTerm should be to take the lessons learned, and use them to write a new system.

The new system should be implemented in a different language on the Macintosh end. Several possibilities are available now.

Mach1 is a new implementation of Forth for the Macintosh. It differs from MacForth in that instead of using threaded code, it has the ability to "unthread" the code, causing the code for invoked words to be compiled inline. Although this can substantially increase the size of the resulting program, execution speed is the same as assembly language or compiled programs. Mach1 also differs from MacForth in how it deals with the toolbox. Mach1 makes very few assumptions about how the the Macintosh toolbox should be used, giving the programmer much more freedom than MacForth does. Finally Mach1 provides support for writing applications in a multi-tasking environment. Multiple Mach1 programs can execute simultaneously, communicating with each other via atomic operators supplied by the Mach1 kernel. A multi-tasking environment could prove

very powerful for implementing the Macintosh end of a new MWTerm. Each window could be implemented as a separate task, with a central task accepting characters from the host and placing them in buffers unique to each task and accepting keyboard input to go to the host. An implementation in Mach1 might be able to make use of some of the already developed MacForth code.

In addition, there are now numerous compilers for the standard compiled languages, including C, Pascal, Modula-2, and Fortran. These compilers all generate efficient code, and allow the programmer to use the complete facilities of the Macintosh toolbox.

Although Mach 1 is an appealing option, there are other advantages to using a standard compiled language such as C. If both ends of MWTerm are written in the same language, there may be sections of the code dealing with the protocol which can be shared by the two ends.

There is an effort in the micro computer user community to develop a machine independent library of routines for accessing features of the user interface. If such a library comes about it, would be desirable to write the Macintosh end of MWTerm to use it, allowing a relatively easy port of the software to other machines such as the Atari and Amiga. This would also favor the use of a standard compiled language such as C.

One of the major limitations of the current implementation, is the cost in throughput of changing the destination window of the character stream from the host. This is caused by the host's strategy of blindly sending characters as soon as they became available, along with a destination address if needed. A much better strategy would be to buffer characters with the same destination, and send them all together as a packet. This

would substantially increase throughput, simplify the job of the recognizer, and allow error correction. It would also allow a natural implementation of file transfer.

The danger in "packetizing" the characters is that the output may become jerky, with characters in a window appearing in periodic chunks. It is not clear if this would be an acceptable behavior. There are some strategies that might overcome this problem.

The first strategy would be to time-out packets. Do not allow a packet to collect for longer than some period of time. After the packet times out, send it, even if it only has a few characters.

Another related strategy is to assign priorities to windows. For example, the currently active window should get characters very shortly after they are available, so its packets should have an extraordinarily short time-out period. Background windows would have lower priorities and longer time-out periods, possibly with the priority being adjusted dynamically, based on the how many characters are currently being passed to it. One possible algorithm says that the more characters going to a background window, the lower its priority.

Another possible enhancement is to move the I/O channel from the RS-232 line to the Apple Talk local area network. There is now hardware which allows connection of an Apple Talk network to a TCP/IP network running on a conventional Ethernet. It seems feasible to send packets over a local area network, such as the TCP/IP - Apple Talk connection, rather than a serial connection. This idea is worth considering when MWTerm is fast enough that its weak link is reading the serial line quickly enough. Certainly MWTerm is currently too slow to justify a LAN implementation,

and it is not obvious if MWTerm will ever be fast enough to exceed the effective baud rates available from current host machines.

Another direction for MWTerm to go in would be to simply use two serial lines, one line connected to each of the Macintosh's serial ports. Schemes which try to use one line for commands and the other for data would be a waste of hardware. There would be problems allocating both lines on the Unix end, and the line carrying commands would be under-utilized. One attractive implementation would be to simply restrict MWTerm to two windows, one window per line. Then MWTerm simply become a terminal emulator that supports both serial ports at the same time, and sends the output from the serial ports to different windows. There would be no need from MWTerm to have a cooperating program on the host end, since each line to the host would be a separate user from the host's perspective. Also MWTerm would be operating system independent; in fact each window could be logged in to a different computer, running a different operating system, with different line characteristics. The limitations to this implementation are the restriction to two windows, and the need for two serial lines serving two modems from the Macintosh. Experiments with future versions of MWTerm or with current alternatives should indicate if the two window limitation is an acceptable alternative.

Once the performance problems in MWTerm are settled, the next priority should be to add support for the editing and graphics windows. The editing window is really a prerequisite for doing useful work on most computers.

Also MWTerm should include the feature of allowing cutting and pasting between windows.

A useful implementation needs to have its interface rounded out. It needs to deal more gracefully with user errors such as not setting the baud rate, and handle the error conditions better. There is currently no mechanism for the Unix end to inform the Macintosh end when an error occurs.

For example, if the user requests a new window, the Macintosh end creates the window and then sends a message to host requesting a new subprocess to map to the window. If the host cannot create the subprocess, it does not signal the Macintosh end, effectively preventing the Macintosh end from recovering. Ideally MWTerm should inform the user of the error and then remove the newly created window.

3.5 Summary

The current implementation of MWTerm has served well as an initial feasibility study. It brought into focus several problems with the design, as well as suggesting solutions to these problems.

Desireable future enhancements center on improving performance and adding support for editing and graphics.

References

1) "The Blit: A multiplexed Graphics Terminal", R Pike, AT&T Bell Laboratories Technical Journal, Vol. 63, No 8, October 1984, pp 1607-1631

2) "Debugging C Programs with the Blit", T. A. Cargill, AT&T Bell Laboratories Technical Journal, Vol. 63, No 8, October 1984, pp 1633-1647

3) "Graphics in Overlapping Bitmap Layers", R Pike, ACM Transactions on Graphics, Vol 2, No 2, April 1983, pp 135-160

4) "The Smalltalk Graphics Kernal", David H. H. Ingalls, Byte, Vol 6, No 8, August 1981, pp 168

5) "The Smalltalk Environment", Larry Tesler, Byte, Vol 6, No 8, August 1981, pp 90

6) "The User Interface for Saphire", Brad A. Myers, IEEE CG&A, December 1984, pp 13-23

7) "The Apple Macintosh Computer", Gregg Williams, Byte, Vol 9, No 2, February 1984, pp 30

8) "The Macintosh", Bruce Webster, Byte, Vol 9, No 8, August 1984, pp 238

9) "Inside Macintosh", 1985, Apple Computer Co. , promotional edition.

10) "Macintosh Revealed Vol I & II", Stephen Chernicoff, 1985, Hayden Book Company

11) "The Lisa Computer System",Gregg Williams, Byte, Vol 8, No 2, February 1983, pp 33

12) "MacForth Level 1", Alan Clute, Dr. Dobb's Journal, Vol 10, No 10, October 1985, pp 100

13) "An Architectural Trail to Threaded-Code Systems", Peter M. Kogge, Computer, March 1982, pp 22-32

14) "What is Forth? A tutorial Introduction", John S. James, Byte, Vol 5, No 8, August 1980, pp 100

15) "The Evolution of Forth, an Unusual Language", Charles Moore, Byte, Vol 5, No 8, August 1980, pp 76

16) "Designing the Star User Interface", David Canfield Smith et al. , Byte, Vol 7, No 4, April 1982, pp 242-282

17) UW: A Multiple-Window Terminal Emulator for Use with 4.2BSD UNIX, John D. Bruner, 1985

Appendix A - Source Code for the Unix end of MWTerm

```
#include        <stdio.h>
#include        <sys/ioctl.h>
#include        <sys/wait.h>
#include        <signal.h>
#include        <fcntl.h>
#include        <sys/types.h>
#include        "uterm.h"

#define BIT_SET(n,p)    (p |=  (1<<(n)))
#define BIT_CLR(n,p)    (p &= ~(1<<(n)))
#define BIT_TEST(n,p)   ( p & (1<<(n)))
#define BIT_ZERO(n,p)   ( p = 0)

#define MIN(a,b)                ((a < b) ? a : b)

/*      The folowing two macros provide a mechanism for calling
        functions and trapping on error return statuses.
        Notice that CALL will not work if one of the arguments to
        the function is a quoted string
*/

#define CRASH(string)   (perror(string);exit(1);)
#define CALL(func)              (int c_rslt;c_rslt = func;if(c_rslt<0)CRASH("func");)

/*  Declare any forward references */

int             clean_up();
int             catch_child();


struct  child_rec{
        int             pty;
        int             pid;
        int             state;
};

struct  child_rec       child[CHILD_LIM];

struct  sgttyb  old_tty_sg;             /* save /dev/tty characteristics till end */
struct  sgttyb  new_tty_sg;             /* new /dev/tty, also use to set pty slave */
struct  sgttyb  new_pty_sg;             /* use to set up pty for children */
struct  tchars  new_pty_tc;
struct  ltchars new_pty_ltc;
int             new_pty_lmode;

int     active_child;
int     command_state;
int     nfds;
int     readmsk;
int     writemsk;
int     exceptmsk;
int     read_bits;
int     done = FALSE;

FILE    *fp,*fopen();
```

```
/* ***************************************************************************
        Main
        ____

                The main loop.

*************************************************************************** */

main()
{
        int             i;                              /* utility integer */


        fp = fopen("/dev/null","w");

        CALL (initialize());

        active_child = open_child();
        printf("first child open\n");

        while(~done){
                readmsK = read_bits;
                dump_state("above select");
                CALL (select(nfds,&readmsK,&writemsK,&exceptmsK,0));
                dump_state("below select");
                if (BIT_TEST(0,readmsk)){               /* it's from the mac */
                        CALL (from_mac());
                )
                for (i=0; i < CHILD_LIM; i++){
                        if (child[i].state != UNUSED_STATE){
                                if (BIT_TEST(child[i].pty,readmsk)){
                                        CALL (from_child(i));
                                }
                        }
                }
        }
}


/* ***************************************************************************

        Open_child
        ----------

        Spawn a new subprocess. Set up ptys to control the subprocess, and
        set up a child_rec in the array of child_rec for the new child.
        Finally we need to set active_child to point at this guy, since a
        newly opened window is always made the active window.

        Inputs:

        -- None --
```

```
        Outputs:

        Return the index into child of the child_rec for the new child.
        if we can't open a new child, either crash, or return the current
        active child index.

************************************************************************ */

open_child()
{
        char    pty_name1[30];          /* used to construct file name of pty */
        char    pty_name2[30];          /* used to construct file name of pty */
        char    buff[3];                        /* buffer for read */
        int             pid;                            /* gets pid of new process */
        int             pty_slave;                      /* file desc of pty slave for new process */
        int             status;                         /* get return status from function call */
        int             i;                                      /* utility integer */
        int             child_index;            /* gets index of child to use */
        struct  sgttyb  pty_master_sg;  /* used to set pty master */
        struct  sgttyb  old_pty_sg;
        int             pgrp;                           /* used to set process group of pty */

        status = get_pty(pty_name1,pty_name2);
        if (status < 0) {
                mac_err("no more ptys!");
                return(active_child);
        }

        for (child_index=0; child_index <= CHILD_LIM; child_index++){
                if (child[child_index].state == UNUSED_STATE) break;
        }

        if (child_index > CHILD_LIM){
                mac_err("No more windows!");
                return(active_child);
        }

        child[child_index].state = ACTIVE_STATE;/* mark child as active */
        printf("about to open(%s,...)\n",pty_name1);
        child[child_index].pty = open(pty_name1,O_RDWR,0);
        if (child[child_index].pty < 0) CRASH("open ptyname1...");

        /* I don't know why I need to make this ioctl, but fionread returns
                        0 characters available unless I do this */

        CALL (ioctl(child[child_index].pty,TIOCGETP,&old_pty_sg));

        BIT_SET(child[child_index].pty,read_bits);

        /* Set the process group for pty */

        CALL (ioctl(0,TIOCGPGRP, &pgrp));
        CALL (ioctl(child[child_index].pty,TIOCSPGRP,&pgrp));
        CALL (getpid());

        printf("About to fork\n");
```

```
        Outputs:

        Return the index into child of the child_rec for the new child.
        if we can't open a new child, either crash, or return the current
        active child index.

************************************************************************ */

open_child()
{
        char    pty_name1[30];          /* used to construct file name of pty */
        char    pty_name2[30];          /* used to construct file name of pty */
        char    buff[3];                        /* buffer for read */
        int             pid;                            /* gets pid of new process */
        int             pty_slave;                      /* file desc of pty slave for new process */
        int             status;                         /* get return status from function call */
        int             i;                                      /* utility integer */
        int             child_index;            /* gets index of child to use */
        struct  sgttyb  pty_master_sg;  /* used to set pty master */
        struct  sgttyb  old_pty_sg;
        int             pgrp;                           /* used to set process group of pty */

        status = get_pty(pty_name1,pty_name2);
        if (status < 0) {
                mac_err("no more ptys!");
                return(active_child);
        }

        for (child_index=0; child_index <= CHILD_LIM; child_index++){
                if (child[child_index].state == UNUSED_STATE) break;
        }

        if (child_index > CHILD_LIM){
                mac_err("No more windows!");
                return(active_child);
        }

        child[child_index].state = ACTIVE_STATE;/* mark child as active */
        printf("about to open(%s,...)\n",pty_name1);
        child[child_index].pty = open(pty_name1,O_RDWR,0);
        if (child[child_index].pty < 0) CRASH("open ptyname1...");

        /* I don't know why I need to make this ioctl, but fionread returns
                        0 characters available unless I do this */

        CALL (ioctl(child[child_index].pty,TIOCGETP,&old_pty_sg));

        BIT_SET(child[child_index].pty,read_bits);

        /* Set the process group for pty */

        CALL (ioctl(0,TIOCGPGRP, &pgrp));
        CALL (ioctl(child[child_index].pty,TIOCSPGRP,&pgrp));
        CALL (getpid());

        printf("About to fork\n");
```

```
        if ((pid=fork()) == 0) {                /* I'm the Chid */
                status = ioctl(open("/dev/tty",O_RDWR),TIOCNOTTY,(char *)0);
                if (status < 0) CRASH("ioctl(open(),TIOCNOTTY,...)");

                close(0);
                close(1);
                close(2);

                pty_slave = open(pty_name2,O_RDWR,0);
                if (pty_slave < 0) CRASH("Pty_slave open");


                CALL (ioctl(pty_slave,TIOCSETN, &new_pty_sg));
                CALL (ioctl(pty_slave,TIOCSETC,&new_pty_tc));
                CALL (ioctl(pty_slave,TIOCSLTC,&new_pty_ltc));
                CALL (ioctl(pty_slave,TIOCLSET,&new_pty_lmode));

                CALL (dup2(0,1));
                CALL (dup2(0,2));

                CALL (read(pty_slave,&buff[0],2));


                execl("/bin/csh","csh",NULL);
                CRASH("Past execl");
        }
        else{                                           /* I'm the parent */

                /* CALL (ioctl(0,TIOCGETC,&pty_master_sg)); */
                pty_master_sg.sg_ispeed = old_tty_sg.sg_ispeed;
                pty_master_sg.sg_ospeed = old_tty_sg.sg_ospeed;
                pty_master_sg.sg_erase = old_tty_sg.sg_erase;
                pty_master_sg.sg_kill = old_tty_sg.sg_kill;
                pty_master_sg.sg_flags = old_tty_sg.sg_flags;
                pty_master_sg.sg_flags |= CBREAK;
                pty_master_sg.sg_flags &= (~ECHO);
                CALL (ioctl(child[child_index].pty,TIOCSETN,&pty_master_sg));

                i = 1;
                CALL (ioctl(child[child_index].pty,FIONBIO, (char *) &i));

                sleep(3);
                status = write(child[child_index].pty,"go",2);
                if (status < 0) CRASH("write(child[i].pty,...)");

                child[child_index].pid = pid;
                return(child_index);
        }
}


/* ***************************************************************************

        Get_pty
        -------
```

```
        This routine will try to find a ttypx/ptypx pair that are available
        for use. The tests that is conducts to determine the the pty is acutally
        usable are calls to access and trying to open both ends of the pty.


        Inputs:

        --None--

        Outputs:

        master_str:     pointer to a string, gets the name of the master half of pty
        slave_str:      pointer to a string, gets the name of the slave half of the pty
        return status : error status

*************************************************************************** */

get_pty(master_str, slave_str)
char    *master_str;
char    *slave_str;
{
        int             status;         /* get status from func calls */
        long int                fd_master;      /* get master fie descriptor */
        long int                fd_slave;       /* get slave file descriptor */
        char    master_name[30];        /* gets candidate master pty names */
        char    slave_name[30];         /* gets candidate slave pty names */
        char    *endings = "0123456789abcdef"; /* candidate name endings */
        char    *test;          /* points a current test ending */

        for (test = endings; *test; test++){    /* scan endings list */
                sprintf(&slave_name[0],"%s%c","/dev/ttyp",*test);
                printf("slave name = %s\n",slave_name);

                sprintf(&master_name[0],"%s%c","/dev/ptyp",*test);
                if ((access(master_name,6)==-1) || (access(slave_name,6)==-1))
                        continue;

                printf("past access calls\n");
                fd_master = open(master_name,O_RDWR,0);
                fd_slave = open(slave_name,O_RDWR,0);
                printf("fd_master = %d\tfd_slave =%d\n",fd_master,fd_slave);
                if ((fd_master )-1) && (fd_slave )-1)){
                        close(fd_master);
                        close(fd_slave);
                        break;
                }
                if (fd_master )-1) close(fd_master);
                if (fd_slave )-1) close(fd_slave);
        }
        if (*test == '\0') return(-1);
        strcpy(master_str,master_name);
        strcpy(slave_str,slave_name);
        return(1);
}

/* ***************************************************************************
```

```
        From_child
        ----------

        This is the routine which handles reading the characters from a child
        and passing them on the the mac. If the characters are from a child
        other then the child we last read from, then first send a header to
        the mac telling it which child the characters are comming from.


        Inputs:

        index:  The index into the child array of the child to read from.

        Outputs:

        return status: Return an error status.

******************************************************************************* */

from_child(index)
int             index;
{

        char    buff[BUFF_SIZE];        /* buffer for reading & writing */
        long    int             count;  /* how many chars are there to read */
        static  int     last_in = 0;    /* index of child last read from */


        CALL (ioctl(child[index].pty, FIONREAD, &count));
        count += 1;

        if (index != last_in){                  /* different child then last read */
                buff[0] = ESCAPE_CHAR;
                buff[1] = SH_IO_ID;                     /* shell I/O id command */
                buff[2] = index+48;                     /* ascii digit */
                CALL (write(1,&buff[0],3));     /* send command string */
                last_in = index;                        /* set new last_in */
        }

read_loop:

        CALL (read(child[index].pty,&buff[0],MIN(BUFF_SIZE,count)));
        CALL (write(1,&buff[0],MIN(BUFF_SIZE,count)));

        if (count > BUFF_SIZE){
                count -= BUFF_SIZE;
                goto read_loop;
        }

        return(1);
}

/* ******************************************************************************

        From mac
```

```
        --------

        This routine reads the input from the mac, and scans through the
        input stream for commands from the mac end. Non command characters
        are passed to the active child, command characters are dispatched to
        lower level routine to handle.

        Inputs:

        --None--

        Duptuts:

        Return status: return an error status

*************************************************************************** */

from_mac()
{
        char    buff[BUFF_SIZE];        /* I/O buffer */
        int             count;                          /* gets # characters to read */
        int             i;                                      /* counting int */

        CALL (ioctl(0,FIONREAD, &count));


read_loop:

        CALL (read(0,&buff[0],MIN(BUFF_SIZE,count)));

        for (i=0; i < MIN(BUFF_SIZE,count); i++){
                if (command_state == GOT_ESCAPE){       /* already got an escape */
                        CALL (parse(buff[i]));
                        continue;
                }
                if (command_state != NOP_STATE){        /* expecting command data */
                        CALL (finish_command(buff[i]));
                        continue;
                }
                if (buff[i] == ESCAPE_CHAR){            /* got start of command */
                        command_state = GOT_ESCAPE;
                        continue;
                }

                CALL (write(child[active_child].pty,&buff[i],1)); /* just pass it on */
        }

        if (count > BUFF_SIZE){
                count -= BUFF_SIZE;
                goto read_loop;
        }

        return(1);
}
```

```
/* ************************************************************************

        Parse
        -----

        This routine is called with the character following an escape character.
        It is supposed to decide what command the the mac is giving it. If the
        command isn't followed by a child index, execute it, otherwise set the
        command state so that we're looking for the child index.

        Inputs:

        c:      the command character to act on


        Outputs:

        return status: return error status


************************************************************************** */

parse(c)
char    c;
{
        int             child_index;            /* gets index of new child */
        char    return_stat[5];         /* buffer to return child index to mac */
        int             status;                         /* gets return status */
        switch(c){
        case    NEWSH_CMD:
                child_index = open_child();
                sprintf(&return_stat[0],"%c%c%1d",ESCAPE_CHAR,NEWSH_ID,child_index);
                CALL (write(1,&return_stat[0],3));
                command_state = NOP_STATE;
                break;
        case    KILLSH_CMD:
                command_state = GOT_KILL;
                break;
        case    ACTIVESH_CMD:
                command_state = GOT_ACTIVE;
                break;
        case    ESCAPE_CHAR:
                CALL (write(child[active_child].pty,&c,1));
                command_state = NOP_STATE;
                break;
DEFAULT:
                command_state = NOP_STATE;      /* error !*/
        }
        return(1);
}



/* ************************************************************************

        Finish command
```

```
        -------------


        This routine reads a child index from the mac, and based on the
        pending command implied by command_state, executes the command
        for the appropriate child.


        Inputs:

        c:      The input character from the mac, ascii digit of child  index


        Outputs:

        return status: Returns an error status.


************************************************************************** */

finish_command(c)
char    c;
{
        int             child_index;            /* gets the actual child index */
        int             i;                                      /* utility integer */

        child_index = c - 48;           /* convert from ascii to binary integer */

        switch(command_state){
        case    GOT_KILL:
                BIT_CLR(child[child_index].pty,read_bits);      /* don't read anymore*/
                /* CALL (close(child[child_index].pty)); */
                CALL (kill(child[child_index].pid,SIGKILL));
                CALL (wait(&i));
                command_state = NOP_STATE;
                child[child_index].state = UNUSED_STATE;
                /* Check to see if any children left */
                done = TRUE;
                for (i=0; i<CHILD_LIM;i++){
                        if (child[i].state != UNUSED_STATE) done=FALSE;
                }
                if (done) exit(1);
                break;
        case    GOT_ACTIVE:
                active_child = child_index;
                command_state = NOP_STATE;
                break;
        }
        return(1);
}


/* ************************************************************************

        Initialize
        ----------
```

```
        Initialize global variables.


        Inputs:

        --None--

        Outputs:

        Return value: Return an error status.


******************************************************************************* */

initialize()
{
        int             i;              /* utility integer */

        /* Set up to catch any signals */

        CALL (signal(SIGHUP,clean_up));
        CALL (signal(SIGTERM,clean_up));
        CALL (signal(SIGCHLD,catch_child));
        /* CALL (signal(SIGCHLD,clean_up)); */
        CALL (signal(SIGINT,clean_up));

        /* initialize children */

        for (i=0; i< CHILD_LIM; i++){
                child[i].state = UNUSED_STATE;
        }

        CALL (ioctl(0,TIOCGETP,&new_pty_sg));
        new_pty_sg.sg_flags += 2;               /* turn on CBREAK for pty */

        CALL (ioctl(0,TIOCGETC,&new_pty_tc));
        CALL (ioctl(0,TIOCGLTC,&new_pty_ltc));
        CALL (ioctl(0,TIOCLGET,&new_pty_lmode));

        CALL (ioctl(0,TIOCGETP,&old_tty_sg));   /* get a copy to save */
        CALL (ioctl(0,TIOCGETP,&new_tty_sg));   /* get a copy to munge */
        new_tty_sg.sg_flags += 2;                       /* turn on cbreak mode */
        new_tty_sg.sg_flags &= (~ECHO);                 /* Turn off echo mode */

        CALL (ioctl(0,TIOCSETN,&new_tty_sg));   /* set /dev/tty */

        read_bits = 1;  /* read from stdin at first */
        nfds = CHILD_LIM + 1; /* just set for max number of inputs */
        /* nfds = getdtablesize(); */
        command_state = NOP_STATE;

        return(1);
}
```

```
/* ************************************************************************

        Mac_err
        -------

        Do something about letting user know of a problem

        Inputs:

        string: A string describing problem to user

        outputs:

        --None--

************************************************************************** */

mac_err(string)
char    *string;
{
        printf("%s\n",string);
}


/* ************************************************************************

        Clean_up
        --------

        Do any cleaning up of children and I/O units before exiting.

------------------------------------------------------------------------

        Inputs:

        signal: This routine catches certain signals. The signal parameter
        gets the value of the signal that this routine is catching.

------------------------------------------------------------------------

        Outputs:

        --None--

************************************************************************** */

clean_up(sig)
int             sig;
{
        int             status;
        int             i;
```

```
        dump_state("top of clean_up");
        CALL (signal(SIGHUP,SIG_DFL));
        CALL (signal(SIGTERM,SIG_DFL));
        CALL (signal(SIGCHLD,SIG_DFL));
        CALL (signal(SIGINT,SIG_DFL));

        printf("signal(%d)\n",sig);

        status = ioctl(0,TIOCSETP,&old_tty_sg);
        if (status < 0) perror("clean_up ioctl(0,...)");

        for (i=3; i < CHILD_LIM + 3; i++) close(i);

        status = kill(0,SIGKILL);
        if (status < 0) perror("clean_up kill");

        printf("Exit (%d)\n",sig);

        exit(1);
}
/* ***************************************************************************

        catch_child
        -----------

        Deal with deceased children

---------------------------------------------------------------------

        Inputs:

        signal: the signal handler calls catch_child with the signal
        ID.

---------------------------------------------------------------------

        Outputs:

        --None--

*************************************************************************** */

catch_child(signal)
int     signal;
{

        int     i;
        int     done;

        done = TRUE;

        /* Are there any children still in use, if not we're donw */

        for (i=0; i< CHILD_LIM; i++){
                if (child[i].state != UNUSED_STATE) done = FALSE;
        }
```

```
        if (done) exit(1);
}

dump_state(string)
char    *string;
{
        int     i;
        fprintf(fp,"***************** %s ******************\n",string);
        for (i=0;i<CHILD_LIM; i++){
                fprintf(fp,"child[%d].pty=%d\tchild[%d].pid=%d\tchild[%d].state=%d\n",
                i,child[i].pty,i,child[i].pid,i,child[i].state);
        }
        fprintf(fp,"active_child=%d\n",active_child);
        fprintf(fp,"command_state=%d\n",command_state);
        fprintf(fp,"read_bits=%d\n",read_bits);
        fprintf(fp,"************************************\n");

        fflush(fp);
}
```

```
/* ****************************************************************************

        UTERM.H

        The include file for UTERM.

**************************************************************************** */

#define CHILD_LIM               10
#define READ                    0
#define WRITE                   1
#define FALSE                   0
#define TRUE                    1

#define ESCAPE_CHAR             '%'
#define NEWSH_CMD               'N'
#define KILLSH_CMD              'K'
#define ACTIVESH_CMD    'A'
#define NEWSH_ID                'I'
#define SH_ID_ID                'S'

#define UNUSED_STATE    0
#define ACTIVE_STATE    2
#define BACK_STATE              4

#define GOT_ESCAPE              0
#define GOT_NEWSH               1
#define GOT_KILL                2
#define GOT_ACTIVE              3
#define NOP_STATE               4

#define BUFF_SIZE               256
```

Appendix B - Source Code for the Macintosh end of MWTerm

```
Block#0                  "toolkit:uterm"      08/07/86      10:09:16 PM
 0  Uterm, Rev .5                                          ( 030985 Del)
 1                     Author: Del Armstrong
 2
 3                     Uterm Development blocks.
 4
 5     Uterm is a multiple window/multiple process interface to
 6     a UNIX host. This set of blocks impliments the Mac end
 7     of the system.
 8
 9     See the table of contents on block 12
10
11
12
13
14
15

Block#1                  "toolkit:uterm"      08/07/86      10:09:18 PM
 0 ( Thesis work load block )                               ( 040586 Del)
 1  FORTH DEFINITIONS  SET.FENCE
 2  CR ." Extending Thesis System..."   DECIMAL
 3  apple.menu OPTIONS.MENU  ( display the apple and options menu )
 4
 5  100000 MINIMUM.OBJECT      ( establish mimimum object & Vocab )
 6  5000  MINIMUM.VOCAB
 7
 8  3 load                     ( define some useful words )
 9  7 load                     ( include any useful tools )
10  8 load                     ( more debugging words )
11  init.cursor    cr
12  abort                      ( start interpreting from console )
13
14
15

Block#2                  "toolkit:uterm"      08/07/86      10:09:21 PM
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

MacFORTH Kernel 2.4
```

```
Block#3                  "toolkit:uterm"     08/07/86     10:09:23 PM
 0 ( debugging tools )                                  ( 040586 Del)
 1 : .p count print crlf 2 print ;
 2 variable debug.flg  false debug.flg !
 3
 4 : ok.mem  ( check heap size )
 5     ?heap.size 1000 < if
 6         ." heap size =" ?heap.size .
 7         true error" out of heap space"
 8     then
 9 ;
10  -->
11
12
13
14
15

Block#4                  "toolkit:uterm"     08/07/86     10:09:25 PM
 0 ( debugging tools: check stack usage )               ( 040586 Del)
 1 create stack_stack  20 4 * allot  ( 20 4-byte entries )
 2 create string_stack 20 4 * allot  ( 20 string pointers )
 3 variable stack_stack_ptr   stack_stack   stack_stack_ptr !
 4 variable string_stack_ptr  string_stack  string_stack_ptr !
 5
 6 : stack_stack_push      ( n --- )
 7     stack_stack_ptr @ !
 8     stack_stack_ptr @ 4+ dup   stack_stack 80 +  > if
 9           error" stack_stack overflow"
10     then stack_stack_ptr ! ;
11 : string_stack_push     ( StrngAddr --- )
12     string_stack_ptr @ !
13     string_stack_ptr @ 4+ dup   string_stack 80 +  > if
14           error" string_stack overflow"
15     then string_stack_ptr ! ;  -->

Block#5                  "toolkit:uterm"     08/07/86     10:09:29 PM
 0 ( debugging tools: check stack usage )               ( 040586 Del)
 1
 2 : stack_pop   ( --- StackDepth ! also pop string stack )
 3   string_stack_ptr @ 4-  string_stack_ptr !
 4   stack_stack_ptr @ 4- dup stack_stack_ptr ! @ ;
 5 : begin_stack? ( StringAddr --- ! push depth & StringAddr )
 6     string_stack_push   depth stack_stack_push
 7 ;
 8 : stack?  ( n --- ! check that only n items added on stack )
 9     depth swap - 1- stack_pop = not if
10         sys.window window
11         cr ." Wrong number of items on stack in word: "
12         string_stack_ptr @  @ count type     cr .s  abort
13     then
14 ; -->
15

MacFORTH Kernel 2.4
```

```
Block#12                    "toolkit:uterm"     08/07/86     10:11:57 PM
 0 ( Uterm Table of Contents ... load whole system ) ( 030985 Del)
 1
 2 15   load    ( initializations)
 3 30   load    ( window manipulations )
 4 45   load    ( event record stuff )
 5 60   load    ( control handeling stuff)
 6 75   load    ( TEdit stuff )
 7 \ include" Text Edit"
 8
 9 90   load    ( protocal support )
10 105  load    ( VT-100 emulator )
11 120  load    ( Tek 4010 emulator )
12 135  load    ( serial I/O support )
13 150  load    ( main UTERM words )
14
15

Block#13                    "toolkit:uterm"     08/07/86     10:11:59 PM
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

Block#14                    "toolkit:uterm"     08/07/86     10:12:01 PM
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

MacFORTH Kernel 2.4
```

```
Block#15                  "toolkit:uterm"     08/07/86     10:13:03 PM
 0 ( initializations: allocate & access windows )     ( 030985 Del)
 1 : me.not  ;
 2                    \ 10 x [handle,shell-id,next node]
 3 create window.array  10 12 * allot
 4 : 'th.window   ( n --- address of nth window handle )
 5     12 *    window.array + ;
 6 : 'th.link     ( n --- address of nth link node )
 7     12 *    window.array +     8+ ;
 8 : 'th.shell.id ( n --- address of nth shell id )
 9     12 *    window.array +     4+ ;
10 : th.window    ( n -- w.handle   return nth hndl from array)
11     12 *      window.array +    @ ;
12 : th.link      ( n --- contents of nth link node )
13     12 *    window.array +     8+     @ ;
14 : th.shell.id  ( n --- contents of nth shell id )
15     12 *    window.array +     4+     @ ;      -->

Block#16                  "toolkit:uterm"     08/07/86     10:13:05 PM
 0 ( initializations: allocate windows )             ( 030985 Del)
 1
 2
 3 ( brute force ... create 10 windows, and store the handles )
 4 new.window n.w0    n.w0 0 'th.window !
 5 new.window n.w1    n.w1 1 'th.window !
 6 new.window n.w2    n.w2 2 'th.window !
 7 new.window n.w3    n.w3 3 'th.window !
 8 new.window n.w4    n.w4 4 'th.window !
 9 new.window n.w5    n.w5 5 'th.window !
10 new.window n.w6    n.w6 6 'th.window !
11 new.window n.w7    n.w7 7 'th.window !
12 new.window n.w8    n.w8 8 'th.window !
13 new.window n.w9    n.w9 9 'th.window !
14 -->
15

Block#17                  "toolkit:uterm"     08/07/86     10:13:08 PM
 0 ( initializations: initialize linked list )       ( 030985 Del)
 1
 2 variable free.window    \ points at the next free window node
 3 : init.links    ( initilize the "next free node" link pointers )
 4     9 0 do
 5         i 1+     'th.window   \ address of next node
 6         i        'th.link     \ address of "next node" pointer
 7         !                     \ store it
 8     loop
 9 ;
10
11
12 init.links                     \ initialize the linked list
13 0 'th.window    free.window !  \ point at head of linked list
14 -->
15

MacFORTH Kernel 2.4
```

```
Block#18                  "toolkit:uterm"     08/07/86     10:13:12 PM
 0 ( initializations: Set window's attributes )
 1
 2 : set.attributes.of.windows
 3     ( set attributes for windows in window.array)
 4     9 0 do
 5         " Uterm Window"                  i th.window w.title
 6         close.box size.box +
 7    \    not.visible +
 8         text.record +
 9         scroll.up/down +                 i th.window w.attributes
10         50 50 100 400                    i th.window w.bounds
11    \    i th.window  add.window
12     loop
13 ;
14 set.attributes.of.windows       \ actually set the attribs
15  -->

Block#19                  "toolkit:uterm"     08/07/86     10:13:14 PM
 0 ( initializations: Define menus )                  ( 030985 Del)
 1
 2 10 constant new.window.menu
 3
 4 ( define menu to show a new window )
 5 ( first we create a new menu in the menu bar )
 6
 7     0 " Windows"  new.window.menu                 new.menu
 8
 9 ( then we add an item to the menu)
10
11     " Add a New Unix Window" new.window.menu     append.items
12
13 ( Now redraw menu bar with out new menu )
14     draw.menu.bar
15 -->

Block#20                  "toolkit:uterm"     08/07/86     10:13:23 PM
 0 ( initializations : globals & constants )          ( 033086 Del)
 1
 2 variable filter.state
 3 ascii % constant escape.char
 4 ascii I constant newsh.id.char
 5 ascii S constant sh.io.id.char
 6
 7 0 constant nop.state
 8 1 constant got.newsh.state
 9 2 constant got.sh.io.state
10 3 constant got.escape.state
11
12 nop.state filter.state !
13
14 -->
15

MacFORTH Kernel 2.4
```

```
Block#21                    "toolkit:uterm"     08/07/86     10:13:31 PM
 0 ( initialize window titles )                            ( 040286 Del)
 1
 2  " Unix Shell 0"  0 th.window  w.title
 3  " Unix Shell 1"  1 th.window  w.title
 4  " Unix Shell 2"  2 th.window  w.title
 5  " Unix Shell 3"  3 th.window  w.title
 6  " Unix Shell 4"  4 th.window  w.title
 7  " Unix Shell 5"  5 th.window  w.title
 8  " Unix Shell 6"  6 th.window  w.title
 9  " Unix Shell 7"  7 th.window  w.title
10  " Unix Shell 8"  8 th.window  w.title
11  " Unix Shell 9"  9 th.window  w.title
12
13    -->
14
15

Block#22                    "toolkit:uterm"     08/07/86     10:13:42 PM
 0 ( define some toolbox calls )                           ( 040386 Del)
 1
 2 hex
 3
 4 A904    mt     drawgrowicon   ( windowpointer --- )
 5 A957    mt     showcontrol    ( controlhandle --- )
 6 A969    mt     drawcontrols   ( controlhandle --- )
 7 A91C  w>mt     highlightwindow ( windowptr\boolean --- )
 8
 9 decimal
10
11
12
13
14
15

Block#23                    "toolkit:uterm"     08/07/86     10:13:52 PM
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

MacFORTH Kernel 2.4
```

```
Block#30                    "toolkit:uterm"     08/07/86     10:14:57 PM
 0 ( Window management: allocate)                          ( 040586 Del)
 1
 2 : allocate.new.window   ( --- window handle )
 3     " allocate.new.window" begin_stack?
 4     free.window @        ( get the next free node )
 5     dup 0 =
 6         if error" No more windows in window.array"
 7         else
 8             free.window @ 8 + @ ( get contents of link field )
 9             free.window !       ( store in free.window)
10             @                   ( get the window handle )
11         then
12     1 stack?
13 ;
14 --)
15
```

```
Block#31                    "toolkit:uterm"     08/07/86     10:14:59 PM
 0 ( Window management: release.window )                   ( 040586 Del)
 1     \ We put a node in window.array back into the free list. We
 2     \ just do a brute force search through the list to find the
 3     \ right node. This will be a fairly infrequent operation.
 4
 5 : release.window      ( window.handle --- freed in window.array )
 6     " release.window" begin_stack?
 7     9 0 do                          ( loop through windows )
 8         dup    i th.window =       ( does window.handle = i'th? )
 9         if
10             free.window @           ( push ptr to head of free lst)
11             i 'th.window free.window !  ( i'th window is head )
12             i 'th.link !            ( set up link of head node )
13         then
14     loop  -1 stack?
15 ;    -->
```

```
Block#32                    "toolkit:uterm"     08/07/86     10:15:02 PM
 0 ( Window management: Make window active )               ( 031985 Del)
 1
 2 variable out.window     ( contains the current output window)
 3 variable active.window
 4 : the.window     ( -- hndle ! return the current output window )
 5     out.window @
 6 ;
 7
 8 : make.active     ( window.handle --- send output to window )
 9     out.window ! ;
10
11  --)
12
13
14
15
```

```
MacFORTH Kernel 2.4
```

```
Block#33                    "toolkit:uterm"     08/07/86     10:15:05 PM
 0 ( Window management: window.index )                    ( 032585 Del)
 1
 2 : window.index    ( wptr -- n | index of wptr in window.array )
 3    9 0 do
 4        dup
 5        i th.window
 6        = if
 7            drop
 8            i
 9            leave
10        then
11    loop
12 ; -->
13
14
15
```

```
Block#34                    "toolkit:uterm"     08/07/86     10:15:07 PM
 0 ( Window management)
 1
 2 : dump_window ( index --- | list values for node )
 3    dup   ." window(" . ." ) = "
 4    dup   th.window .   4 spaces
 5    dup   ." shell.id(" . ." ) = "
 6           th.shell.id .   4 spaces cr
 7 ;
 8
 9 : dump_windows   ( ---  | dump the window array )
10    cr
11    10 0 do
12        i dump_window
13    loop
14 ;  -->
15
```

```
Block#35                    "toolkit:uterm"     08/07/86     10:15:09 PM
 0 ( Window management )                                  ( 042686 Del)
 1 \ Nasty things happen if you try to use the TErecord associated
 2 \ with windows that don't have TErecords (such as desktop
 3 \ accessories), use this word to check the incoming window.
 4
 5 : ?my_window  ( wptr --- wptr\[T or F] | wptr from uterm? )
 6    " ?my_window" begin_stack?
 7    false swap                ( false\wptr)
 8    9 0 do                       ( wptr = any of uterm's? )
 9       dup i th.window = if      ( false\wptr\wptr=wind[i])
10           swap drop true swap   ( true\wptr )
11       then
12    loop drop ( not  temporary)   ( [T or F] )
13    0 stack?
14 ;       -->
15
```

MacFORTH Kernel 2.4

```
Block#45                  "toolkit:uterm"     08/07/86     10:16:23 PM
 0 ( Event record stuff: toolox words )                ( 031185 Del)
 1
 2    \ The word toolbox will take arguments off of the stack, and
 3    \ set up a toolbox trap. It will also return values.
 4
 5 hex
 6
 7 A970 W) L) )W toolbox get.next.event    ( See IM event manager)
 8 A922 L) toolbox begin.update            ( See IM window manager)
 9 A923 L) toolbox end.update              ( See IM window manager)
10
11 decimal
12
13 create er    20 allot                  ( our event record )
14
15 --)

Block#46                  "toolkit:uterm"     08/07/86     10:16:25 PM
 0  ( Event record stuff: )                          ( 040586 Del)
 1
 2 256 constant activate.event.mask        ( bit mask )
 3
 4 : disable.activates    ( --- Keep 4th from getting activates )
 5    " disable.activates" begin_stack?
 6    events @            ( event mask: " accept everything" )
 7    activate.event.mask
 8    xor            ( mask out the activate.event bit )
 9    events !       ( set the mask 4th uses )
10  0 stack? ;
11
12 : er.type    ( --- contents of event type field in er )
13    er w@ ;    ( first word in record )
14
15   --)

Block#47                  "toolkit:uterm"     08/07/86     10:16:28 PM
 0 ( Event record stuff: )                         ( 031185 Del)
 1
 2 : er.message         ( --- return message field from er )
 3    er 2+ @ ;         ( second long word in record )
 4
 5 : er.modifiers       ( --- return modifiers field from er )
 6    er 14 + w@ ;      ( modifiers start at  byte 14 in record )
 7
 8 : ?activate/deactivate  ( --- true if wndw is being activated )
 9    er.modifiers         ( get modifier bits of event record )
10    1 and ;              ( first bit on if window activation )
11
12
13  --)
14
15

MacFORTH Kernel 2.4
```

```
Block#48                  "toolkit:uterm"     08/07/86     10:16:31 PM
 0 ( Event record stuff: )                                ( 040586 Del)
 1
 2 : ?er.activate     (  --- true/false; is er an activate.event )
 3    er.type activate.event = ;
 4
 5 : ?any.activates    ( -- true/false | if we caught an activate)
 6    " ?any.activates" begin_stack?
 7    " get.any.activates" thats_me
 8    disable.activates   ( prevent MacForth from stealing any )
 9     activate.event.mask ( mask to request activate events )
10     er                  ( where to put event record )
11     get.next.event      ( get event from queue )
12    " done get.any.activates" thats_me
13    1 stack?
14 ;  -->
15

Block#49                  "toolkit:uterm"     08/07/86     10:16:33 PM
 0 ( Event Record Stuff)                                  ( 040586 Del)
 1
 2 64 constant update.event.mask
 3 : disable.updates
 4    events @  update.event.mask xor   events ! ;
 5 : get.any.updates   ( --- true if we got any updates )
 6    " get.any.activates" begin_stack?
 7    disable.updates
 8    update.event.mask  er
 9    get.next.event \ say ." here's the updates" .s cr say_end
10    1 stack?
11 ;
12
13
14
15

Block#50                  "toolkit:uterm"     08/07/86     10:16:41 PM
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

MacFORTH Kernel 2.4
```

```
Block#60                "toolkit:uterm"     08/07/86     10:17:45 PM
 0 ( Controls: scrolling )                              ( 033086 Del)
 1 hex
 2     8  constant teviewrect     ( offset to viewrect in TErecord )
 3     1c constant teselpoint     ( offset to selection point )
 4 decimal
 5 variable #pixels
 6 : @pen.point       ( -- x/y : position of bottom of current char)
 7     @pen
 8     get.window   +wline.height @   +  ;
 9 : #pixels.to.scroll     ( --- n : dist. of pen from content rect)
10     get.window +wcbounds 4+ @    ( bottom of content rectangle )
11     point>xy swap drop           ( just Y of bottom of content)
12     @pen.point swap drop         ( get Y part of pen position)
13     -  1-                        ( calc # of pixels to scroll )
14  dup #pixels ! ;
15 -->

Block#61                "toolkit:uterm"     08/07/86     10:17:48 PM
 0 ( Controls: scrolling )                              ( 033086 Del)
 1     \ ?scroll.lines says how many pixels we need to scroll to
 2     \ bring the pen position back into the content rectangle
 3 : ?scroll.lines     ( --- <#pixels> | false  )
 4     @pen.point xy>point       ( get pen position as a point )
 5     get.window                ( get content region for ... )
 6     +wcbounds ( @ )              ( ... the current window )
 7     ptinrect not              ( is pen out of content region ?)
 8     if
 9         #pixels.to.scroll         ( how many pixels to scroll)
10         dup 0> if                 ( is pen above view rect ?)
11             drop false            ( if so, then return false)
12         then
13     else false            ( return no scroll needed )
14     then
15  ;   -->

Block#62                "toolkit:uterm"     08/07/86     10:17:50 PM
 0 ( Controls: scrolling )                              ( 040586 Del)
 1
 2 : scroll.lines         ( --- | scroll lines if needed )
 3     " scroll.lines" begin_stack?
 4     get.window            ( save current window on stack )
 5     the.window window     ( set up window to scroll )
 6     ?scroll.lines         ( do we need to scroll the text? )
 7     dup if
 8         0 swap tescroll   ( if so then scroll the text in Y )
 9     else drop
10     then
11     window                ( reset current window )
12     0 stack?
13 ;
14
15   -->

MacFORTH Kernel 2.4
```

```
Block#63                  "toolkit:uterm"     08/07/86     10:17:54 PM
 0 ( Controls: adjust cursor )                              ( 040586 Del)
 1
 2    \ When the cursor is in the content region of the active
 3    \ window, it should be the I-beam.
 4
 5 : my.adjust.cursor     ( --- )
 6   " my.adjust.cursor" begin_stack?  active.window @ if
 7    active.window @ window @mouse     ( get position of cursor )
 8    active.window @ +wcbounds     ( get content region )
 9    ptinrect                  ( is cursor in content region )
10    if
11        ibeam  set.cursor    ( make the cursor an I-beam)
12    else
13        ( init.cursor) 0 set.cursor  ( make cursor the arrow )
14    then  the.window window then
15  0 stack? ;  -->

Block#64                  "toolkit:uterm"     08/07/86     10:17:59 PM
 0 ( Controls: scroll bars  )                               ( 031385 Del)
 1
 2 : @vpos    ( -- position : fetches current scroll bar position)
 3    the.window +vbar @    get.control ;
 4
 5 : !vpos    ( position -- : store control position & update )
 6    the.window +vbar @          ( get pointer to control )
 7    swap set.control            ( put pos. on top, set control )
 8    the.window show.controls    ( redraw the control bar )
 9 ;
10
11   -->
12
13
14
15

Block#65                  "toolkit:uterm"     08/07/86     10:18:08 PM
 0 ( Controls: scroll bars )                                ( 050686 Del)
 1 : hilitcontrol  ( true/false -- : draw cntrl dark or empty )
 2    if    this.control @  this.part  hilite.control
 3    else  this.control @  0          hilite.control
 4    then
 5 ;
 6 : inc.control    ( n -- : move the control by n pixels )
 7    " inc.control"  begin_stack?
 8    0 swap             ( set horiz scroll to 0 )
 9    tescroll           ( change the new value for the control )
10    true hilitcontrol ( make the toggled control stand out )
11    the.window show.controls  ( re-draw the contrl bar)
12    false hilitcontrol ( return the toggled contrl to normal)
13    the.window show.controls   ( once again, redraw it )
14 -1 stack?  ;
15   -->

MacFORTH Kernel 2.4
```

```
Block#66                    "toolkit:uterm"      08/07/86      10:18:24 PM
 0 ( Controls: scroll bars )                                  ( 031385 Del)
 1
 2 : do.cntrl     ( n -- : increment the control by n while the )
 3                (         mouse button is held down )
 4     begin
 5         mouse.button                  ( is the mouse button down)
 6         while
 7             dup     inc.control      ( increment control by n )
 8     repeat
 9     drop
10 ;
11
12    -->
13
14
15
```

```
Block#67                    "toolkit:uterm"      08/07/86      10:18:36 PM
 0 ( Controls: scroll bars )                                  ( 031385 Del)
 1
 2 : do.thumb     ( -- : handle thumb control )
 3     this.control @     @mouse     track.control  ( move move box)
 4     drop                          ( ignore reslt from track.contrl)
 5     the.window show.controls     ( redraw control bars )
 6 ;
 7
 8   -->
 9
10
11
12
13
14
15
```

```
Block#68                    "toolkit:uterm"      08/07/86      10:18:45 PM
 0 ( Controls: scroll bars )                                  ( 050686 Del)
 1 : do.prop.controls     ( -- : handle scroll bar controls )
 2     " do.prop.controls"  begin_stack?
 3     @mouse.dn           ( get position of mouse down )
 4     the.window          ( get active window )
 5     find.control        ( did mouse hit a control? which one?)
 6     ?dup
 7     if  swap drop       ( if so, drop the control handle )
 8         case     in.thumb          of     do.thumb        endof
 9                  up.button         of     10  do.cntrl    endof
10                  down.button       of     -10 do.cntrl    endof
11                  page.up           of     60  do.cntrl    endof
12                  page.down         of     -60 do.cntrl    endof
13         endcase
14     then
15 0 stack? ;     -->
```

```
MacFORTH Kernel 2.4
```

```
Block#69                    "toolkit:uterm"     08/07/86     10:19:01 PM
 0 ( Controls: scroll bars )                              ( 040586 Del)
 1
 2 : activate_controls  ( WindowPtr --- | activate scroll bars )
 3   " activate_controls" begin_stack?
 4   +vbar @ get.control    ( get handle to vertical scroll bar )
 5   0 hilite.control       ( insure that it get displayed )
 6   -1 stack?
 7 ;
 8  : deactivate_controls  ( WindowPtr --- | deactivate scrl bar)
 9   " activate_controls" begin_stack?
10   +vbar @ get.control    ( get handle to vertical scroll bar )
11   255 hilite.control     ( "dim" scroll bar )
12   -1 stack?
13 ;
14 --)
15

Block#70                    "toolkit:uterm"     08/07/86     10:19:14 PM
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

Block#71                    "toolkit:uterm"     08/07/86     10:19:17 PM
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

MacFORTH Kernel 2.4
```

```
Block#75                    "toolkit:uterm"     08/07/86     10:20:22 PM
 0 ( TEdit: Set up TE.array )                              ( 032585 Del)
 1
 2 create te.array    10 4 * allot    ( set up an array of 10 )
 3
 4 : @th.te    ( i -- addr of text field )
 5    4* te.array + @ ;
 6
 7 : !th.te   ( n i -- store n in ith element of te.array )
 8    4* te.array + ! ;
 9
10  -->
11
12
13
14
15

Block#76                    "toolkit:uterm"     08/07/86     10:20:24 PM
 0 ( TEdit: init TE records )                              ( 031285 Del)
 1    \ <dest rect> <view rect> <wptr> TEXT.FIELD <name>
 2    \ will create a text field named <name>. A text field has
 3    \ two longwords pointers: TErecord ptr, Window ptr.
 4
 5 0 th.window +wcbounds dup 0 th.window    text.field    text0
 6 1 th.window +wcbounds dup 1 th.window    text.field    text1
 7 2 th.window +wcbounds dup 2 th.window    text.field    text2
 8 3 th.window +wcbounds dup 3 th.window    text.field    text3
 9 4 th.window +wcbounds dup 4 th.window    text.field    text4
10 5 th.window +wcbounds dup 5 th.window    text.field    text5
11 6 th.window +wcbounds dup 6 th.window    text.field    text6
12 7 th.window +wcbounds dup 7 th.window    text.field    text7
13 8 th.window +wcbounds dup 8 th.window    text.field    text8
14 9 th.window +wcbounds dup 9 th.window    text.field    text9
15 -->

Block#77                    "toolkit:uterm"     08/07/86     10:20:27 PM
 0 ( TEdit: init TE records )                              ( 032585 Del)
 1
 2    \ Stuff a pointer to the TErecord into the te.array fields
 3
 4 text0 @   0 !th.te
 5 text1 @   1 !th.te
 6 text2 @   2 !th.te
 7 text3 @   3 !th.te
 8 text4 @   4 !th.te
 9 text5 @   5 !th.te
10 text6 @   6 !th.te
11 text7 @   7 !th.te
12 text8 @   8 !th.te
13 text9 @   9 !th.te
14  -->
15

MacFORTH Kernel 2.4
```

```
Block#78                    "toolkit:uterm"     08/07/86     10:20:30 PM
 0 ( TEdit: Initialize TErecords)                          ( 032885 Del)
 1 hex
 2 : +tefont 4A +   ;  ( word only )
 3 : +cronly 48 +   ;  ( byte only )
 4 decimal
 5 : init.terecords     ( -- : set some of the characteristics)
 6     9 0 do
 7         i @th.te @
 8         dup +tefont 9 swap w!     ( set font to monaco )
 9         dup +cronly -1 swap c!    ( new line on <cr> only )
10         drop
11      loop
12 ;
13    init.terecords
14  -->
15

Block#79                    "toolkit:uterm"     08/07/86     10:20:32 PM
 0 ( TEdit: Update things )                                ( 040586 Del)
 1 : event.window    ( --- | return event.record window field )
 2     er  2+ @
 3 ;
 4 : update.text     ( -- : update dsply of text for update evnt)
 5 " update.test" begin_stack? " update.text" thats_me
 6     the.window                ( get window from event )
 7     dup ?my_window  if        ( don't update sys.window )
 8       dup window                  ( make update wndw current )
 9       terecord                    ( get TE record for window )
10       dup 60 + dup teset.select  ( make sure insertion pt is ok
11       +tvisrect                   ( get visible rect for TErecord
12       teupdate                    ( update the display )
13       the.window window  drop     ( restore active window )
14     else drop then
15 0 stack? " done update.text" thats_me ; -->

Block#80                    "toolkit:uterm"     08/07/86     10:20:43 PM
 0 ( TEdit: Update things )                                ( 040586 Del)
 1
 2 : update.terects     ( -- : fix both rects in terecord )
 3 " update.terects" begin_stack? " update terects" thats_me
 4 the.window dup ?my_window  if
 5     \ get the top-left corner of content region & use it to
 6     \ set top-left corner of dest & view rect in the TE record
 7     the.window +wcbounds @ dup  ( get content rgn of wndw x 2)
 8     the.window terecord !       ( stuff one in dest rect )
 9     the.window terecord 8+  !   ( stuff other in view rect )
10     \ Now do the same thing for lower-right corner of the rgns
11     the.window +wcbounds 4+ @ dup ( contnt rgn of wndw x 2)
12     the.window terecord 4+ !    ( stuff one in dest rect )
13     the.window terecord 12 + !    ( stuff other in view rect )
14 then drop
15 0 stack? " done update.terects" thats_me ;  -->

MacFORTH Kernel 2.4
```

```
Block#81                    "toolkit:uterm"     08/07/86     10:21:00 PM
 0 ( TEdit: write a char to current window )          ( 040586 Del)
 1  hex
 2  A9DC W> L> toolbox my_tekey  \ TEKey (ch:char; TErec: handle)
 3 \ A9DC w>mt my_tekey
 4 decimal
 5
 6 : temit (  c --- | write c to TE record in current.window )
 7   " temit" begin_stack?
 8     the.window +wrefcon @ my_tekey
 9  -1 stack? ;
10 : load_wrefcons
11     9 0 do
12         i @th.te  i th.window +wrefcon  !
13     loop
14 ;     load_wrefcons
15 -->

Block#82                    "toolkit:uterm"     08/07/86     10:21:11 PM
 0 ( TEdit: deal with activate & deactivate )        ( 040586 Del)
 1
 2 : activate_te  ( WindowPtr --- | make sure TE record is ready )
 3    " activate_te" begin_stack?
 4    teactivate          ( set up select rgn for current window)
 5    drop                ( we never used WindowPtr)
 6    -1 stack?
 7 ;
 8
 9 : deactivate_te  ( WindowPtr --- | put TE record to sleep )
10    " deactivate_te" begin_stack?
11    tedeactivate        ( turn off select rgn for current window)
12    drop                ( we never used WindowPtr)
13    -1 stack?
14 ;
15

Block#83                    "toolkit:uterm"     08/07/86     10:21:23 PM
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

MacFORTH Kernel 2.4
```

```
Block#90                    "toolkit:uterm"      08/07/86     10:22:25 PM
 0 ( Uterm:  Handle protocol)                                ( 041986 Del)
 1
 2 : set_nop_state     ( --- | set filter state to nop.state )
 3     nop.state filter.state ! ;
 4
 5 : store_new_shell_id     ( c --- | store shell-id in window )
 6     " store_new_shell_id" begin_stack?
 7     9 0 do
 8         i th.window  active.window @ = if ( find active window)
 9             i 'th.shell.id !             ( stuff in i'th window)
10             set_nop_state                ( reset filter.state )
11         then
12     loop
13     -1 stack? ;
14
15 --)

Block#91                    "toolkit:uterm"      08/07/86     10:22:28 PM
 0 ( Uterm: Handle Protocal )                                ( 041986 Del)
 1
 2 \ set the output window to match one with shell-id char
 3 \ also set out.window global variable to point to new outwindow
 4 : set_new_output_window  ( char --- )
 5     " set_new_output_window" begin_stack?
 6     9 0 do
 7         dup    i th.shell.id  = if
 8             i th.window  out.window !  ( set global flag )
 9           \ i th.window  window        ( send output to there )
10             set_nop_state              ( set filter.state )
11           then
12     loop drop
13     -1 stack?  ;
14 --)
15

Block#92                    "toolkit:uterm"      08/07/86     10:22:30 PM
 0 ( Uterm: Handle Protocol )                                ( 042686 Del)
 1  : set_w the.window window ;
 2 \    handle case where we've already gotten escape.char
 3 : filter_escape    ( char --- )
 4     " fileter_escape" begin_stack?
 5     set_nop_state    ( default case )
 6     case
 7         escape.char     of     ( escape,escape = 1 real escape )
 8             escape.char  ( set_w tekey ) temit
 9           endof
10         newsh.id.char  of  got.newsh.state filter.state ! endof
11         sh.io.id.char  of  got.sh.io.state filter.state ! endof
12     endcase
13     -1 stack? ;
14 --)
15

MacFORTH Kernel 2.4
```

```
Block#93                  "toolkit:uterm"     08/07/86     10:22:33 PM
 0 ( Uterm: Handle Protocol )                                ( 040586 Del)
 1 : set_escape    got.escape.state    filter.state ! ;
 2 : process_char     ( char --- I do the right thing with char )
 3     " process_char" begin_stack?
 4     filter.state @   case
 5                  nop.state    of
 6                          dup escape.char = if drop set_escape
 7       else dup ( set_w tekey) temit 13 = if scroll.lines then
 8                  then endof
 9                  got.escape.state    of  filter_escape endof
10                  got.sh.io.state     of
11                  set_new_output_window  endof
12                  got.newsh.state     of
13                  store_new_shell_id      endof
14      endcase
15     -1 stack? ;  -->

Block#94                  "toolkit:uterm"     08/07/86     10:22:36 PM
 0 ( protocol debugging stuff )                              ( 041986 Del)
 1 variable sim.state   0 sim.state !  variable s.cnt  0 s.cnt !
 2 : pick_window   37 process_char  83 process_char process_char ;
 3 : ?cr s.cnt @ 67 = if 48 pick_window 13 process_char
 4                       49 pick_window 13 process_char
 5                       0 s.cnt ! else 1 s.cnt +! then ;
 6 : a.char random 10 mod 49 + ;
 7 : simulate_serial  " simulate_serial"  begin_stack?
 8     sim.state @
 9     case
10     0    of  ascii 0  pick_window a.char  process_char
11              1 sim.state ! endof
12     1    of  ascii 1 pick_window  a.char process_char
13              0 sim.state ! endof
14     endcase
15     ?cr " done simulate_serial" thats_me 0 stack? ;  -->

Block#95                  "toolkit:uterm"     08/07/86     10:22:50 PM
 0                                                           ( 041986 Del)
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

MacFORTH Kernel 2.4
```

```
Block#135                  "toolkit:uterm"     08/07/86     10:23:56 PM
 0 ( Serial I/O : declarations )                          ( 031385 Del)
 1 \    Much of the serial I/O is based on code supplied by
 2 \    Creative Solutions with MacFORTH. Use of that code requires
 3 \    the inclusion of the following notice.
 4 \  Copyright, 1984 Creative Solutions, Inc.
 5 \  Permission is hereby granted for personal, non-commercial
 6 \  reproduction and use of this program by Creative Solutions
 7 \  MacFORTH Licensees, provided that this notice is included
 8 \  in any copy.
 9
10 \  Creative Solutions, Inc.          CIS id:  75226,1577
11 \  4701 Randolph Rd Suite#12
12 \  Rockville, Md 20852
13
14 \  Some of this code contains enhancements by William Bond
15  -->

Block#136                  "toolkit:uterm"     08/07/86     10:23:59 PM
 0 ( Serial I/O : declarations )                          ( 031585 Del)
 1
 2 ( use an illegal file# to detect uninitialized state )
 3 create serial.file#     100 ,
 4
 5 4096 constant input.size                    ( size of input buffer)
 6 create input.buffer     input.size allot  ( allocate buffer)
 7 hex
 8 create serial.options    00001113 , 000000FF ,
 9 decimal
10 \ disable XON/XOFF and CTS output handshaking
11 \ XON char is 11 (^Q), XOFF char is 13 (^S)
12 \ don't abort on parity, overrun or framing error
13 \ don't post event on break or CTS  status change
14 \ disable XON/XOFF input handshake ( due to bug in ROM driver)
15 -->

Block#137                  "toolkit:uterm"     08/07/86     10:24:01 PM
 0 ( Serial I/O : open)                                   ( 031585 Del)
 1 : open.serial   ( addr\cnt\file# --- | Opens a pair of files )
 2                 ( for serial port A. addr\cnt is the buffer )
 3     dup serial.file# !      ( store file# in serial.file#)
 4     " .AIN" over )fcb       ( --- ...\addr\cnt\file#\"ain"\fcb )
 5     dup>r                   ( stash fcb for a bit )
 6     open.device             ( open the "A" input port )
 7     1+                      ( get file# 1 bigger )
 8     )fcb " .AOUT" swap      ( --- ...\".AOUT"\fcb )
 9     open.device             ( open the A output port )
10     16 scale                ( left shift cnt into 2 high bytes )
11     swap                    ( --- cnt'\addr )
12     9 r>                    ( set buffer cmd, get stashed fcb )
13     device.control          ( issue set buff cmd to serial drvr)
14 ;
15 -->

MacFORTH Kernel 2.4
```

```
Block#138                  "toolkit:uterm"     08/07/86     10:24:05 PM
 0 ( Serial I/O : type, expect)                          ( 031585 Del)
 1
 2 : serial.in    ( -- file# )    serial.file# @ ;
 3 : serial.out   ( -- file# )    serial.in 1+ ;
 4
 5 : s.type    ( addr\cnt --- | type buffer to serial )
 6    serial.out  write.text
 7 ;
 8
 9 : s.expect  ( addr\cnt --- | read cnt chars to addr from serial)
10    serial.in  read.text
11 ;
12
13 --)
14
15

Block#139                  "toolkit:uterm"     08/07/86     10:24:08 PM
 0 ( Serial I/O : more serial words )                    ( 031585 Del)
 1
 2 : s.?terminal    ( -- n : return # of chars in serial buffer )
 3    2                ( status command, "how many bytes gotten")
 4    serial.in >fcb   ( get a fcb for serial in )
 5    device.status    ( issue the trap )
 6    swap drop        ( get rid of refnum result)
 7 ;
 8
 9 : s.key    ( -- char | get a char from serial port )
10    0 sp@ 3+         ( a glorious hack, allocate a word from )
11                     ( stack, and put it's address above it )
12    1                ( expect 1 character )
13    s.expect         ( read a char into allocated word, i.e TOS)
14 ;
15 --)

Block#140                  "toolkit:uterm"     08/07/86     10:24:23 PM
 0 ( Serial I/O : emit, status )                         ( 031585 Del)
 1
 2 : s.emit    ( char -- | send char out serial port )
 3    sp@ 3+    ( get address of TOS )
 4    1         ( # of chars to type )
 5    s.type    ( type 1 char out seral port )
 6    drop      ( eat our input )
 7 ;
 8
 9 : s.status  ( -- stat2\stat1 | error & handshake status )
10    8               ( command, return error & handshake status )
11    serial.in >fcb  ( get fcb )
12    device.status   ( issue status trap )
13 ;
14 --)
15

MacFORTH Kernel 2.4
```

```
Block#141                  "toolkit:uterm"      08/07/86     10:24:33 PM
 0 ( Serial I/O : ready, break )                          ( 031585 Del)
 1 : s.?ready    ( -- flag | true when serial is ready for output)
 2    s.status    ( get 2 words of status info )
 3    swap drop   ( get rid of second word of status info )
 4    not         ( 0000 means ready, convert to TRUE )
 5 ;
 6
 7 : s.break    ( -- | toggle break for 6 ticks [1/10])
 8    0 0 12 serial.out )fcb device.control  ( 12 = set break)
 9    tickcount 6+                           ( get ticks + 6)
10    begin                                  ( start looping )
11        dup tickcount <                    ( still < tickcount?)
12    until drop                             ( loop until done )
13    0 0 11 serial.out )fcb device.control  ( 11 = clear break)
14 ;
15 --)

Block#142                  "toolkit:uterm"      08/07/86     10:24:50 PM
 0 ( Serial I/O : backspace)                              ( 031585 Del)
 1
 2 : backspace    ( char  -- | backspace, erasing prev character)
 3    drop            ( the input char [a bs] is ignored )
 4
 5    @pen swap       ( get current posit y\x on stack )
 6    bl charwidth    ( get width of a blank )
 7    -               ( subtract from current x position )
 8
 9    swap 2dup       ( restore y\x to normal order, and copy)
10    (move.to)       ( move to new positon 1 char to left )
11    bl draw.char    ( write a blank over last character )
12    (move.to)       ( and move backto new position )
13 ;
14
15 --)

Block#143                  "toolkit:uterm"      08/07/86     10:25:03 PM
 0 ( Serial I/O : echo.serial )                           ( 031585 Del)
 1 : echo.serial    ( -- | echo serial input to current window )
 2    pad    s.?terminal    ( -- addr\cnt | cnt = #input chrs)
 3    2dup                  ( -- addr\cnt\addr\cnt )
 4    s.expect              ( -- addr\cnt | read input to addr)
 5    over                  ( -- addr\cnt\addr )
 6    +                     ( -- addr_low\addr_high | aha!)
 7    swap do               ( do i=addr_low,addr_high)
 8        ic@               ( get character)
 9        127 and           ( mask those nasty bits)
10        process_char      ( just handle the character )
11        scroll.lines      ( do any scrolling if necessary)
12    loop ;                ( loop for whole buffer, then exit )
13 --)
14
15

MacFORTH Kernel 2.4
```

```
Block#144                "toolkit:uterm"     08/07/86     10:25:20 PM
 0 ( Serial I/O : set baud )                              ( 031585 Del)
 1 : baud    ( baud rate -- ) set baud rate for serial port )
 2           (  also initialize port first time through )
 3    serial.file# @ 100 =       ( do we need to init serial?)
 4    if
 5        input.buffer  input.size   ( get buffer & size)
 6        #files 2-                  ( pick a file #)
 7        open.serial                ( open the port )
 8    then
 9    serial.in  >fcb  >r            ( get a fcb & stash it)
10    dup 300 -                      ( is baud rate 2 300 ?)
11    if serial.options 2@ else 0 0 then ( if so, fake options)
12    10 r@  device.control          ( set handshake options )
13    >r                             ( stash baud rate )
14    1 ( stop bits)  0 ( no parity)  8 ( data bits) r> ( baud)
15    r> ( fcb)  setup.serial ;  -->

Block#145                "toolkit:uterm"     08/07/86     10:25:35 PM
 0 ( Serial I/O : Keymap)                                 ( 031685 Del)
 1 ( Use keymap to map control keys )
 2 hex
 3 create option.keymap
 4   01130406 , 08071A18 , 03160002 , 11170512 ,
 5   19148182 , 83848685 , 8889878A , 88801D0F ,
 6   151B0910 , 0D0C0A00 , 0B001C00 , 0F0E0D0E ,
 7   0000007F , 00000000 , 00000000 , 00000000 ,
 8
 9 (   A  S  D  F     H  G  Z  X     C  V     8     Q  W  E  R )
10 (   Y  T  1  2     3  4  6  5     =  9  7  -     8  0  ]  O )
11 (   U  [  I  P     cr L  J  '     K  ;  \  ,     /  N  M  . )
12 (  tab sp `  bs    enter                                    )
13 decimal
14  -->
15

Block#146                "toolkit:uterm"     08/07/86     10:25:48 PM
 0 ( Serial I/O : map.option )                            ( 031685 Del)
 1 hex
 2 : map>option    ( char -- char | option-char ==> cntl-char )
 3    key.stroke 10 +  c@    ( get contents of modifiers field )
 4    8 and                  ( check option-key-down field )
 5    if
 6        drop               ( throw away input char )
 7        key.stroke 6+ c@   ( get key code...NOT ASCII VALUE)
 8        3F and             ( mask out bits for keypad )
 9        option.keymap +    ( calculate index into keymap)
10        c@                 ( get new character )
11    then
12 ;
13 decimal
14 -->
15

MacFORTH Kernel 2.4
```

```
Block#147                  "toolkit:uterm"      08/07/86      10:26:00 PM
 0 ( Serial I/O : cursor )                                  ( 031685 Del)
 1
 2 : d_    ( -- | draw an underline cursor)
 3     6 0 (line)      ( draw a line, using rel coordinates )
 4    -6 0 (move)      ( return to start )
 5 ;
 6
 7 -->
 8
 9
10
11
12
13
14
15

Block#148                  "toolkit:uterm"      08/07/86      10:26:09 PM
 0 ( Protocol stuff: Send commands to host)                  ( 042686 Del)
 1 : get_shid     ( wptr --- shell id )
 2     window.index
 3     th.shell.id
 4 ;
 5 : send_kill    ( wptr --- | sends kill command  for
 6                  wptr shell id )
 7     ascii % s.emit  ascii K s.emit get_shid  s.emit ;
 8 : send_newsh    ( --- | send the request for a new shell )
 9     ascii % s.emit   ascii N s.emit  ;
10 : send_active  ( wptr --- | send request to make wptr
11                   the active shell )
12     ascii %  s.emit   ascii A s.emit get_shid s.emit ;  -->
13
14
15

Block#149                  "toolkit:uterm"      08/07/86      10:26:20 PM
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

MacFORTH Kernel 2.4
```

```
Block#150                    "toolkit:uterm"     08/07/86     10:27:25 PM
 0 ( Uterm : Select active window )                          ( 050486 Del)
 1 \ The user has picked a new window to be the active window
 2 : select.new.window                ( -- | handle activate event)
 3  " selct.new.wndw" begin_stack? ( check the stack usage )
 4    er.message window               ( make selected window current)
 5    er.message  activate_te     ( go set up TE stuff )
 6    er.message  activate_controls  ( go turn on controls )
 7    er.message active.window !    ( MWTerm's active window ptr )
 8    -1 er.message hilite.window  ( turn hilite on for window )
 9    active.window  @ ( window.index)  ( get the active window )
10    ( th.shell.id )  send_active      ( make it the active shell)
11  0 stack?     ; -->
12
13
14
15

Block#151                    "toolkit:uterm"     08/07/86     10:27:28 PM
 0 ( Uterm : deselect a window )                             ( 040686 Del)
 1 : deselect.old.window  ( -- | handle deactivate event )
 2  " desel.old.wndw" begin_stack?     ( check stack usage)
 3     er.message window               ( make it current window )
 4     er.message deactivate_te        ( deactivate TE stuff )
 5     er.message deactivate_controls ( put controls to sleep )
 6     0 er.message hilite.window      ( turn hilite off )
 7  0 stack? ;  -->
 8
 9
10
11
12
13
14
15

Block#152                    "toolkit:uterm"     08/07/86     10:27:30 PM
 0 ( Uterm : Handle activates )                              ( 040586 Del)
 1
 2 : service_activate  ( --- | service an activate event )
 3    " service_activate" begin_stack?
 4    er.message  ?my_window if  ( only mess with my windows )
 5       ?activate/deactivate    ( get activate/deactivate flag)
 6       if                      ( if it was an activate )
 7         select.new.window     ( declare new window active )
 8       else
 9         deselect.old.window   ( declare old window deactive)
10       then
11       er.message DrawGrowIcon ( fixup the resize box )
12    then
13  0 stack?  ; -->
14
15

MacFORTH Kernel 2.4
```

```
Block#153                  "toolkit:uterm"     08/07/86     10:27:33 PM
 0 ( Uterm : Handle activates )                          ( 040586 Del)
 1 \    We need to catch and handle window activate events ourself
 2 \    since the MacFORTH way to handle them is all wrong.
 3 : handle.any.activates    ( -- | find and handle activates )
 4   " handle.any.activates" begin_stack?
 5    ?any.activates
 6    if                     ( if we got an activate )
 7      service_activate     ( go take care of it )
 8    then
 9   0 stack? ;  -->
10
11
12
13
14
15

Block#154                  "toolkit:uterm"     08/07/86     10:27:35 PM
 0 ( Uterm : Handle Keystroke)                           ( 031685 Del)
 1
 2 : handle.any.keystrokes    ( -- | xlate & xmit any keystrokes )
 3     ?terminal              ( get any keys hit recently )
 4     if
 5         key.stroke 7+ c@   ( get key code )
 6         map>option         ( map option key to control )
 7         key.stroke off     ( zero key buffer )
 8         dup 126 =          ( is it a ~ [exit command])
 9         if  abort then     ( then exit )
10         s.emit             ( send xlated char to serial port )
11     then
12 ;
13 -->
14
15

Block#155                  "toolkit:uterm"     08/07/86     10:27:44 PM
 0 ( Uterm : handle serial input )                       ( 031685 Del)
 1
 2 : handle.any.serial.input    ( -- | serial input ==> window )
 3     s.?terminal         ( any serial input? )
 4     if
 5         echo.serial
 6     then
 7 ;
 8
 9 -->
10
11
12
13
14
15

MacFORTH Kernel 2.4
```

```
Block#156                  "toolkit:uterm"      08/07/86      10:27:51 PM
 0 ( Uterm : mouse events )                                  ( 040586 Del)
 1
 2 : handle.any.mouse.events     ( -- I deal with mouse clicks )
 3  " handle.any.mouse.events"  begin_stack?
 4     mouse.down.record @     ( get first field of event record )
 5     if                      ( if field is - 0)
 6     mouse.down.record 4+ @ active.window @ = if \ active window?
 7         ?in.control         ( was event in a control )
 8         if
 9             do.prop.controls  ( if so, handle control stuff )
10         else
11             text.click       ( else user clicked in text )
12         then
13         mouse.down.record off ( clear field )
14     then then
15 0 stack? ;    -->

Block#157                  "toolkit:uterm"      08/07/86      10:28:06 PM
 0 ( Uterm : housekeeping )                                  ( 040586 Del)
 1 variable wcheck.var  get.window wcheck.var !
 2 : wcheck  get.window wcheck.var @ = not if  60 120 5000 tone
 3     get.window wcheck.var ! then ;
 4 : do.housekeeping     ( -- I do any useful repeat tasks )
 5   " do.housekeeping"  begin_stack?
 6     my.adjust.cursor     ( make sure cursor is apropriate )
 7   ( wcheck )  teidle            ( make insertion bar blink )
 8  0 stack? ;
 9 : update.display      ( -- I update both rect and the display )
10  " update.display" begin_stack?
11     update.terects    ( update display rectangles )
12     update.text       ( redraw any text that needs to be )
13     the.window terecord 60 + dup teset.select  update.field
14    " update.display is done" thats_me
15 0 stack? ; -->

Block#158                  "toolkit:uterm"      08/07/86      10:28:22 PM
 0                                                           ( 041986 Del)
 1
 2  -->
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

MacFORTH Kernel 2.4
```

```
Block#159                  "toolkit:uterm"     08/07/86     10:28:28 PM
 0 ( Uterm: mask events)                                  ( 040586 Del)
 1 \ Since MacFORTH seems to sometimes reset the event mask used
 2 \ by do.events (despite my wishes to the contrary) my_do.events
 3 \ will insure that desired events are masked out when I think
 4 \ they are (such as before calling do.events). It also insures
 5 \ the MacFORTH is straight about the currently active window
 6 : mask_events     ( mask --- | apply mask to EVENTS )
 7     -1 xor  events @  and   events !
 8 ;
 9 : my_do.events    ( --- event# | mask EVENTS then do.events )
10  " my_do.events"  begin_stack?
11   active.window @  window     ( make active window TheWindow )
12   activate.event.mask mask_events  ( mask off activates )
13   do.events
14  1 stack?
15 ; -->

Block#160                  "toolkit:uterm"     08/07/86     10:28:44 PM
 0 ( handle menu stuff )                                  ( 050686 Del)
 1 : new_window_request    ( --- | drop a new window for the user )
 2           " New_window_request"  begin_stack?
 3           allocate.new.window
 4           dup ?wptr if
 5              dup out.window ! dup add.window
 6              dup  dup window.index @th.te swap  +wrefcon !
 7              window
 8              send_newsh
 9           else 100 error" bad window in declare menu"
10           then
11           0 stack?
12 ;
13 : kill_window_request
14          active.window @ send_kill ;  -->
15

Block#161                  "toolkit:uterm"     08/07/86     10:28:57 PM
 0 ( Uterm : Main loop )   : nop ;                        ( 050686 Del)
 1 : (uterm)    ( -- | loop, handling user & serial events )
 2    new_window_request
 3    begin
 4    do.housekeeping
 5         handle.any.activates
 6         my_do.events
 7            case in.size.box   of update.display  endof
 8                 in.close.box  of kill_window_request endof
 9                 update.event  of update.display  endof
10                 abort.event   of " abort" thats_me     endof
11            endcase
12        ( simulate_serial )           handle.any.serial.input
13        handle.any.keystrokes
14        handle.any.mouse.events
15    again ;  -->

MacFORTH Kernel 2.4
```

```
Block#162                  "toolkit:uterm"     08/07/86     10:29:12 PM
 0   -->                                                  ( 042286 Del)
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

Block#163                  "toolkit:uterm"     08/07/86     10:29:18 PM
 0 ( Uterm : Initializations )                            ( 041986 Del)
 1  : declare.menu.action      ( -- I handle new window menu )
 2     new.window.menu   menu.selection:
 3     case
 4       1 of  new_window_request    endof
 5       endcase
 6     0 hilite.menu
 7 ;
 8    declare.menu.action      -->
 9
10
11
12
13
14
15

Block#164                  "toolkit:uterm"     08/07/86     10:29:25 PM
 0 ( Uterm: Check for the window wierdness)               ( 032385 Del)
 1  : w.test
 2     4 0 do
 3         cr i .
 4         i th.window .
 5         i th.window +wrefcon @ .
 6     loop
 7 ;
 8  : test_window ascii % process_char ascii S process_char
 9         process_char ;
10  : simulate
11     begin  30 0 do  ascii 0 dup  test_window process_char
12                     ascii 1 dup  test_window process_char
13               loop    ascii 0 test_window 13 process_char
14     ascii 1 test_window 13 process_char do.events drop
15     again ; -->

MacFORTH Kernel 2.4
```

```
Block#165                  "toolkit:uterm"     08/07/86     10:29:37 PM
 0                                                    ( 050486 Del)
 1 : load_shell_ids
 2     9 0 do
 3         i 48 +    i 'th.shell.id !
 4     loop
 5 ;
 6 : initialize
 7    load_shell_ids
 8    set_nop_state
 9    0 th.window dup     out.window !    active.window !
10    reset_stack?
11 ;
12 initialize      300 baud ( set the def baud rate )
13
14   -->
15

Block#166                  "toolkit:uterm"     08/07/86     10:29:47 PM
 0 ( terminal emulator )                              ( 042686 Del)
 1
 2 : get_char
 3     s.?terminal if s.key 127 and emit then ;
 4 : send_char
 5     ?keystroke if s.emit then ;
 6 : terminal
 7     begin  do.events drop get_char send_char again ;
 8
 9
10
11
12
13
14
15

Block#167                  "toolkit:uterm"     08/07/86     10:29:57 PM
 0 ( code debugging example)                          ( 032986 Del)
 1
 2 : window sys.window window cr
 3     ." window called ... x to stop" key 120 = if
 4     error" you asked for it" then window ;
 5
 6 : do.activate  ( flag --- | handle "on.activate)
 7   if
 8     event.record 2+ @   select.new.window
 9   else
10     event.record 2+ @   deselect.old.window
11   then
12 ;
13 0 th.window on.activate  do.activate
14 1 th.window on.activate  do.activate
15 2 th.window on.activate  do.activate

MacFORTH Kernel 2.4
```