A Proposal for Kernel Implementation of a Window Facility for ASCII Terminals in the UNIX System V Operating System.

Margareta Mihordea

A Major Report

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the Degree of Master in Computer Science at

Concordia University

Montréal, Québec, Canada

March 1988

ISBN 0-315-44243-3

ABSTRACT

A Proposal for Kernel Implementation of a Window Facility for ASCII Terminals in the UNIX System V Operating System

Margareta Mihordea

UNIX System V on the Cadmus processor has a standard terminal driver for a multiplexer board. This report proposes the implementation of a windowing facility for the Cadmus, by extending the multiplexing capability of this standard driver to a second level: window-process pairs for each terminal, allocated from a common pool on user request. This approach differs from the user level implementation encountered in most of the existing window facilities, in that the user level window manager implements only initialization and termination functions, while the kernel additions handle window multiplexing as a simple extension to the standard driver functions. The advantage of our proposal consists of the fact that with only slight modifications in the kernel, the overhead due to the multiple system calls of the user level approach is significantly diminished. In this work, the design proposal has been restricted to full screen windows.

Acknowledgements

Je remercie Dr. T. Fancott qui par son aide, sa confiance en moi, son exigeance professionnelle de haut niveau, a su rendre possible l'exécution de ce travail.

Table of Contents **page**

List of Figures

Chapter 1

Introduction

The concept of multiple windows on a user workstation has been with us since Alan Kay introduced the Smalltalk (Tes81) environment in the early 70's. The concept was essentially developed to represent each mode of execution in a user session by a separate window, thus theoretically retaining all information appropriate to that session accessible to the user in a visual format. In the words of Alan Kay (Tes81) this constitutes the basis of what he calls an "integrated environment", where all the capabilities of a system are available to apply to any appropriate information.

In the last few years the idea of an integrated environment has been adopted and developed by many non-Smallfalk systems. A great number of window facilities supporting both text and raster graphics displays have been developed for a large range of personal computers (Apple Macintosh system, GEMDOS and Microsoft Windows on MSDOS, and the Xerox Star are a few examples) and larger configurations, including timeshared and network computers (VTMS (Lan79), WINDOWS (Uni86) and others).

In a multiprogramming environment such as UNIX, a user can run different programs in the background while working on other jobs in his command interpreter from the same terminal. The problem in running multiple programs without windows resides in the fact that messages from the background processes will appear asynchronously with the input or output of the current process on the same screen, assuming appropriate system settings. As stated by Jacob (Jac84) this happens because "Traditional user interfaces for computers that handle parallel processes place all inputs and outputs in one chronological stream, identifying the process associated with each, but interleaving the data". Although several tasks can be executed in parallel by maintaining a conventional terminal display environment, only one task can be visibly active at any given time. The user is thus forced to switch display contexts in order to monitor multiple jobs.

These two problems can be solved by replacing the conventional display with a windowed one. Each of the related processes run from a given terminal is associated with a window, i.e. a reserved portion of the terminal screen. When programs in different windows are run, their respective outputs containing compiler messages, results, etc., will be directed to the appropriate windows thus avoiding the intermingling of messages on the screen. Additional functions such as creating a new window; switching between existing windows, and changing window size and position, will provide the user the facility of switching from one task to the other without having to save and restart any program he is using.

We may say that windowed systems represent a natural step in the evolution of the interface between humans and interactive computer software; "...a window-based user interface enables a user to manage a collection of dialogues by associating a spatial location with each dialogue, in much the same way one organizes a desk" (Jac84). The analogy with the desktop model is also mentioned in (Mey81), (Gam84), as with the windowing technique each window corresponds to an item on paper or a dossier.

1.1. Definitions and General Terms.

A **window** is defined ((Jac84), (Gam84), (Wei85) et al.) as a portion of the terminal screen which may range in size from the area occupied by one character up to the entire screen. We might call it a small screen. We should distinguish between **system windows** which simply display messages destined for user guidance, and user **created windows**, each associated with a process in order to allow programs to run in the same way as from a dedicated terminal.

Depending on the desired complexity and overhead, the windows may be overlapped, tiled, full screen or combinations of all (BRUWIN (Mey81), BSD 4.3 WINDOWS (Uni86)). The user may switch back and forth between the windows without loss of window context.

A window which is visible partially or totally on the screen is said to be "active", meaning that output may be displayed in these windows. Only one of these active windows may accept keyboard data, the one which has been selected by the user to work in. This is called the **current window.** It has to be completely visible and it is distinguished from the others by the presence of the cursor. Some authors use the term "active" referring to "current", but from the context it is always possible to detect the right meaning. A newly created window is always made current. Outputs appear in the visible windows as they are generated.

The main advantages of windowing are :

1.The user has access to several system facilities at a time by direct interaction. For example,when debugging a program, the user may run the program in one window and edit it in another. When noticing error messages in the former he will intervene immediately by correcting the error in the source code in the other window. The user thus avoids switching between an edit mode and an execute mode. Alternatively, the user may edit a program in one window and upon need ask for information from an on-line manual in another window. He also may hold one window for mailing in order to communicate his latest changes to his team mates, etc.

2.The user can have several tasks in progress each represented in a different window. He can switch with minimal effort between tasks by switching between windows. This is similar to an increase of the number of terminals in the system. Although it translates to a need for more computing power, this is not a problem as hardware is inexpensive, and getting more so every year, while the cost of manpower is constantly rising.

The window facilities present two main disadvantages:

1.Windows compete with each other for screen space. A solution to this is represented by overlapping windows as figured out by Alan Kay.

2.There is additional overhead due to new functions needed in addition to those for regular screen management like mapping of the outputs to the windows, multiplexing and switching between windows.

1.2. Windows for Workstations or Terminals.

Some window systems are conceived for ordinary terminals (Rosetta Smalltalk (War79), USCD p-System Windows (Tat82), VT (Gam84), WSH (Bre84)). Some others may be adapted to run on bitmapped displays (CWSH (Wei85)) or raster-graphics display (BRUWIN (Mey81)). There are also window systems designed for workstations (such as Xerox Interlisp Programmer's Assistant (Tei81) for ALTO minicomputers, BLIT (Pik84), WINDOWS (Uni86), or Wm (Jac84)) where each window contains a dialogue between an intelligent terminal and one of the computers in the network.

1.3. The Goal of This Report.

This report presents a proposal for the design of a kernel level implementation of the multiplexing functions of a windowing display for ASCII terminals. The proposed design is based on a UNIX System V driver for a tty multiplexer board. This driver already handles multiplexed tty input and output from the interface board. In this work, we extend this multiplexing function to a second level, the level of multiple windows on each terminal. The existing structures, with their synchronizing and character handling functions, are used to the greatest extent possible to provide each window with the same level of control as a dedicated terminal in a standard UNIX environment.

Some user level software is needed to control the initialisation and termination of processes in windows opened by the user. This software is minimal in the proposed system, and is not active during normal window I/O.

1.4. The Description of the Organization of This Report.

Ch.2 surveys existing window facility concepts conceived for ASCII terminals in the last four years.

Ch.3 gives a detail study of two existing window implementations namely Blit (Pik84), for the conceptual part of multiplexing a physical port between a number of virtual terminals, and WINDOWS (Uni86) which adapts the Blit concept to ASCII terminals. Ch.3 also describes a local Concordia experiment, WOW (Lac87) done for our Unix V system on the Cadmus.

Ch.4 describes the tty driver architecture as it is encountered in the present system V configuration for the CADMUS computer along with the main structures and the control terminal concept in Unix. The information in Ch.4 gives the background of our approach for a window facility implemented using kernel modifications as opposed to a user level implementation.

Ch.5 describes the modifications required to the kernel tty driver routines in order to implement a window facility. A discussion of the overhead for both the user level and kernel level implementations is included as well as the extensions required to this work to support overlapping windows.

Chapter 2

Background

The program implementing a window facility is usually referred to as a ***window manager*** and has a number of modules for the different functions. The basic modules encountered are: the ***display manager***, the ***virtual terminal emulator***, the ***command interpreter*** (if non-standard) and a ***multiplexer***. The following material emphasizes a few aspects of each as they appear in recent articles.

2.1. The Command Interpreter.

Each user created window is associated with a process running a command interpreter which for UNIX is called a shell (or cshell).

There are basically two approaches related to this:

1. The first approach leaves the standard shell supplied with the respective UNIX version and adds additional code usually as part of the control program module in order to interpret control characters assigned to window control functions (commands). An example of such a software architecture is given in Fig.2.1 (Mey81).

This approach implies two modes of operation:

1.An input or process mode in which the user typed data is directed to the current window where it is interpreted appropriately by the window shell and
2.A command mode where the user typed data is interpreted as a window operation (move, create, etc.).

Normally a special character such as escape ensures easy switching between the two modes. This method presents the advantage of leaving the UNIX facilities unchanged.

2. The second approach is based on an integrated window shell which is achieved by modifying the conventional shell to understand the commands for window operations. In this integrated approach the design consists of three

distinct modules: a ***virtual terminal emulator*** (vte), ***an integrated window interpreter*** and a ***router*** which appropriately switches between the two. An example of this architecture is given in Fig.2.2 (Bre84).

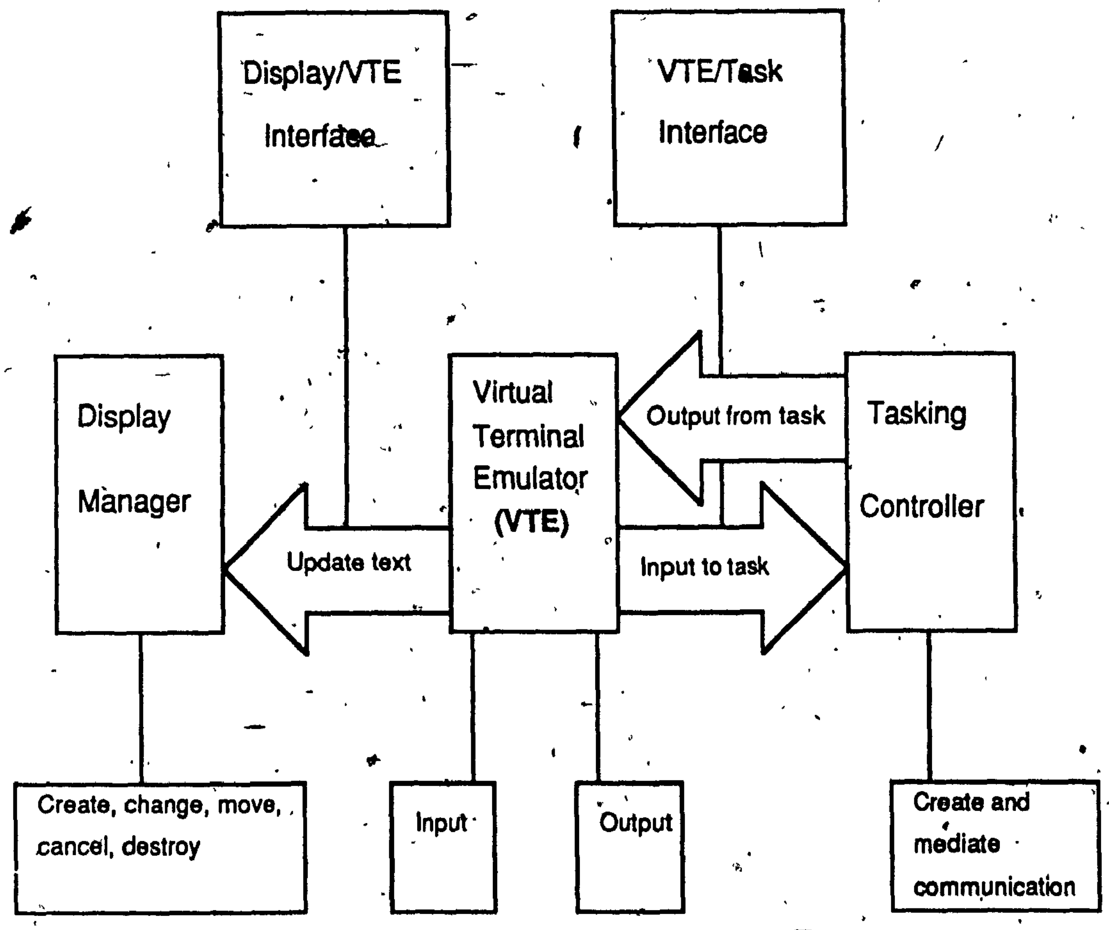

Figure 2.1
Structure of BRUWIN (Mey81)

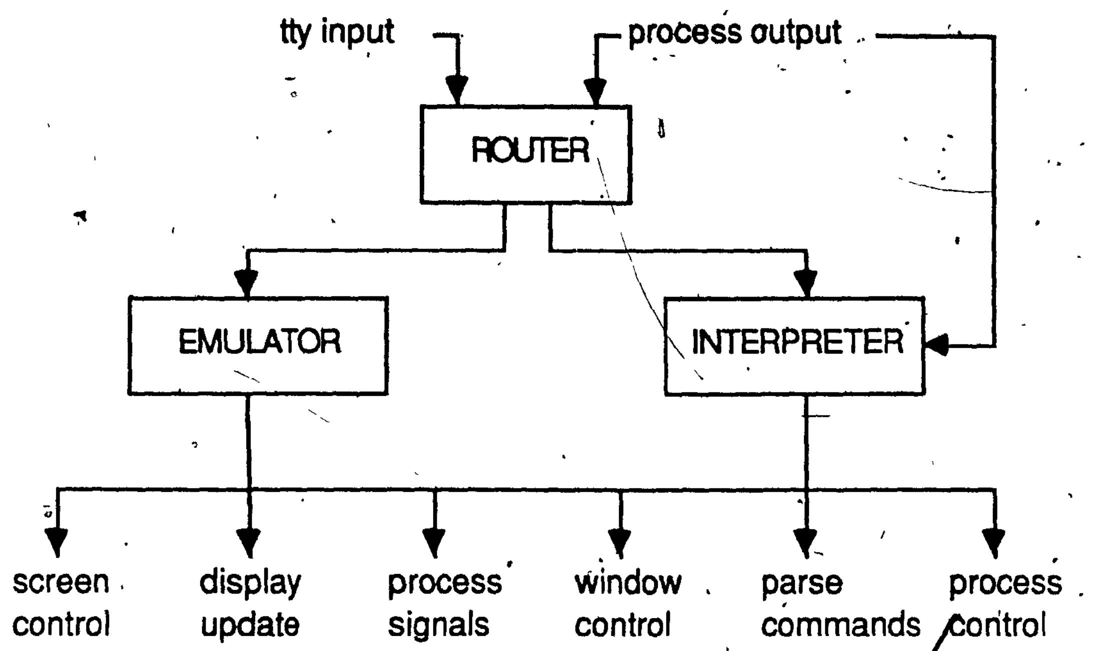

Fig. 2.2
The Division of Labour in WSH (Bre84)

The advantage of this approach over the former is that it eliminates an additional command layer between the operating system and the user's shell.

2.2 The Display Manager.

The display manager groups functions related to the screen control in order to maintain the appropriate display appearance. Its complexity may vary according to the requested window operations. If complex, it is implemented as a separate module as in the BRUWIN (Mey81), WSH (Bre84), and CWSH (Wei85) systems. This gives a window facility the possibility of working with new versions of display managers (Wei85). A further refinement of the general design for a display manager is given in (Mey81). It should contain three main submodules:

1. A ***command interface submodule*** accepting user commands in a system window(s) which may be either statically allocated in a predefined area as in BRUWIN (Mey81), WOW (Lac87), VT (Gam84), WINDOWS (Uni86) or dynamically allocated (usually near the cursor) and erased when the message is no longer needed as encountered in typical PC windowing.

The display manager can accept commands in several lexical forms according to the terminal in use. For ASCII terminals they are usually given as cryptic reduced function keys (RIG VTMS (LAN79), WSH (Bre84)) or written as command strings which are either short, one character, ((Tat82), WOW (Lac87), WINDOWS (Uni86)) or long, a string of characters, (Wm (Jac84), WINDOWS (Uni86), BRUWIN (Mey81)) or a combination of the two (CWSH (Wei85)).

2.A ***display maintenance submodule*** manipulating information which tells when and where to draw the windows.

If overlapping of windows is not required, the display manager is simple, controlling tiled windows in predefined viewing areas. With this scheme the structure defining the window contains the x,y coordinates of the upper left hand corner and its size.

If the facility supplies sizing and/or overlapping window functions, then the window structure contains additional information about the visible part of a window and window boundaries (Unib). This information is used in order to build routines which solve space conflicts and redraw the windows. For example a list of existing windows is needed (UCSD (Tat82), WINDOWS (Uni86)) to show the order in which they were displayed to ensure correct overlapping when several operations apply to them. A set of flags showing different characteristics of the window such as scrolling, is also required. For the redraw strategy there are systems with the capability of drawing only the visible portions of windows. The calculations involved in this case are complex, thus possibly resulting in a slower time than when the window is completely redrawn as done in BRUWIN (Mey81).

3.A ***submodule for graphical operations*** which deals with primitives to draw lines, boxes and characters. These primitives are needed to provide icons and borders around the windows by outlining them. For simple ASCII terminals, the borders may be drawn with horizontal and vertical bar characters ('-','I') (WOW (Lac87)) or by using a different character for each new window ((Tat82), (Jac84)). For high resolution graphic terminals they may be formed of continuous lines either coloured or with a different intensity relative to the screen. The window borders represent one frequent technique to help the user distinguish his concurrent tasks when he is working with many windows ((Hol86)).

If a window facility puts no limitation on the number of windows to be created for a single user, even if the windows are properly outlined, the user may get lost, particularly if totally overlapped windows are allowed (BRUWIN (Mey81), WINDOWS (Uni86), Wm (Jac84), CWSH (Wei85)). Some authors exclude from their window definition the "hidden windows" due to the " violations of user interface principles" ((Hol86)) .

Another technique helping user orientation in the above situation is to supply icons. An icon is a reduced box containing a graphic symbol which is representative for the activity in the respective window. Icons are usually placed close to the screen margins. Any icon can be made current by pointing to it via the cursor keys or a specific mechanism like a mouse.

A similar technique is proposed ((Hol86)) for ASCII terminals. It consists of replacement of the icons with shrunken windows, each one containing a character string defining the activity of that window. These small windows are placed in a similar manner to icons around the edges of the terminal screen. At any given time a large screen area is left for the current window. An additional window operation should be provided in order to swap any one of the shrunken windows with the current window when the user so requires.

The display manager functions are usually implemented by accommodating specific changes to an already existing Screen Management Library Package such as curses ((Jac84), WOW (Lac87)). Another example using a different screen package is WINDOWS (Uni86) described in Ch.3.

According to (Unib), the curses package contains routines which "update a screen with reasonable optimization, get input from the terminal in a screen oriented fashion and move the cursor optimally from one point to another independent of the two previous functions". All curses routines can access the TERMCAP database describing the terminal capabilities in a terminal independent fashion such that "common terminal functions such as scroll, insert character, delete line, are looked up in the database with a generic name. Programs need only know these generic names and not the specific codes" (Mey81). A display manager using the TERMCAP feature (CWSH (Wei85), WSH (Bre84), Wm (Jac84) BRUWIN (Mey81)) has the following advantages:

1.it may run on many different terminals
2.it is portable to other systems having this feature and
3.it allows all software using TERMCAP to run with no modifications.

The original version of wm (Jac84), which does not use curses, has grouped all terminal dependent code in a few simple routines which can be easily replaced with other ones if a graphics terminal will be in use.

In the VTMS system (Lan79), the screen space is managed through an hierarchical decomposition as done for graphics systems. The screen primitives are of two kinds: logical primitives (window and superwindow), operated by processes and which are mapped via configurations into physical primitives (viewport, region, image), manipulated by the user. A configuration specifies the relative positions and sizes of a window. The images contain the invisible windows. The user swaps between them by using a special key.

In VT (Gam84) (see Fig.2.3.), the screen management is done by a screen driver process which along with a port mechanism forms the interface between processes and the screen. The port is a nonstandard call which has as effect the creation of a named pipe. Its purpose is to allow communication between arbitrary processes through messages or streams (see Ch.3).

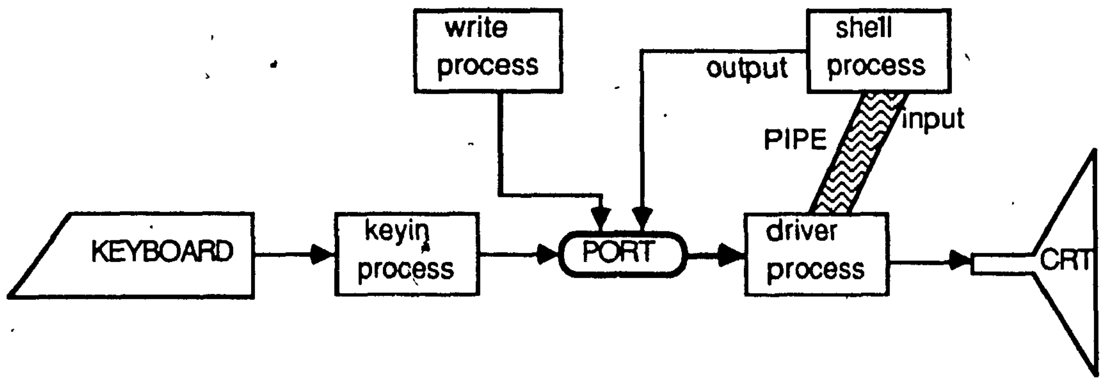

Fig 2.3

VT processes, Devices and Data Flow (Gam84)

When invoked for the first time, the window facility creates a full screen window which becomes current. Any new window can be created by splitting the current window either horizontally or vertically (subject to some limitations as described in (Gam84)) at the cursor position indicated by the user. The user thus controls the display space through the window commands as in any windowing facility.

2.3. The Virtual Terminal Emulator (VTE).

In a windowing environment each user created window should behave like an interactive terminal. This is achieved with a ***virtual terminal emulator (vte) module*** which accomplishes for windows the functions done by the line discipline routines normally associated with an interactive terminal.

The virtual terminal is "an abstraction of a real terminal" ((Bre84), (Mey81)). It has the role of supplying each window with UNIX terminal capabilities. Each window process will read and write its associated virtual terminal instead of a physical line corresponding to a real device (BRUWIN (Mey81)). This ensures device independence as processes do their I/O on a "universal terminal". The translation from device independent to device dependent code is still needed, but it is performed in software by operating on the virtual terminal data structure (Mey81).

The existing window facilities use different techniques in designing virtual terminals influenced by the interprocess facilities of the UNIX version in question as illustrated by a few examples given below.

In RIG VTMS (Lan79) the virtual terminal has three logical components: the ***line***, the ***pad*** and the ***window***, each being managed by independent processes called the LINE, the PAD and the SCREEN HANDLER respectively. They are created with the first user access to the VTMS window and they form together the Virtual Terminal Controller (VTC) as seen in Fig.2.4.below.

A ***line*** is a structure, actually a queue of characters serving as input end to a virtual terminal. Any number of lines may be created, but only one may receive user typed characters; the one designated by the user as being "active" (Lan79). When, the characters are read by a user process they are echoed to their window in a manner similar to a terminal driver function in a traditional interactive environment. Moreover, the input may be processed in three modes: one character, one line or one page at a time.

The ***pad*** is a disk data structure which is a two dimensional array of lines accessed by line number and character position within that line. The main functions of a pad are to store and edit virtual terminal output. The pad is also used for recovering from crashes through the use of two temporary disk files (Mey81). The text editing feature of the pad refers to the cursor movement, character, word and page deletion, character overwrite and insertion, string location and substitution and text selection and transfer. The editing pad feature permits the use of the output of

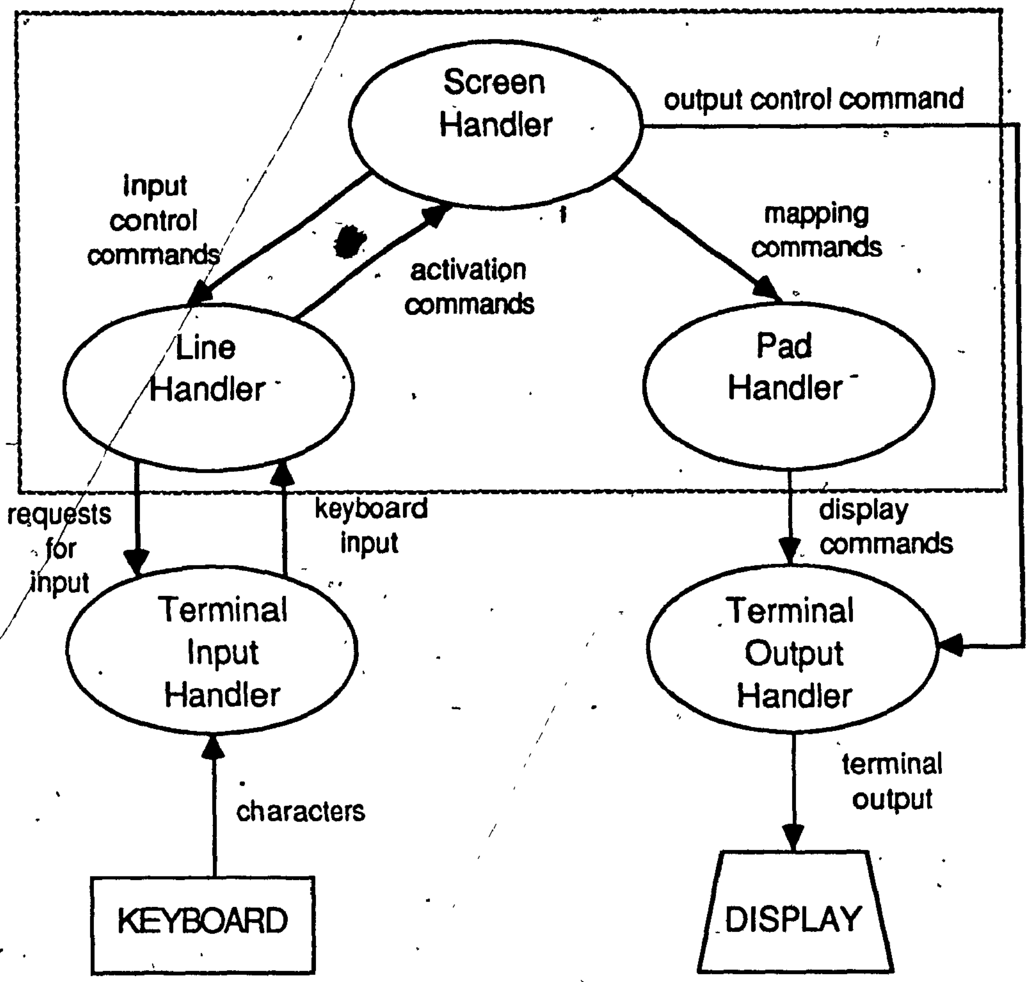

Fig 2.4

The Virtual Terminal Controller (Lan79)

one Virtual Terminal as input to another in a similar way as for UNIX pipes (Lant79).

The third component of the RIG VTMS virtual terminal, the ***window*** structure, serves the purpose of the so called "virtual terminal mapper" (Mey81). In RIG VTMS it is called the Screen Handler and it maps the pad contents to the screen.

The input and output character interpretation is done from the LINE and PAD HANDLERS respectively which in turn communicate to their corresponding device specific terminal Input and Output Handlers. The Line Handler also distinguishes between different types of control characters. Some of these may come from screen management keys in which case the Screen Handler is activated from the Line Handler.

To communicate with VTC, the application processes use a Virtual Terminal Protocol which is not influenced by the physical terminal type. Terminal dependency is handled by the protocol between the terminal Input and Output Handlers and the terminal. With this strategy a wide range of devices may be handled reliably.

In the BRUWIN window system (Mey81), the virtual terminal emulator manipulates only one logical component, the ***map***, which is a two-dimensional array of characters addressed by the cursor. The map corresponds to the pad in RIG VTMS (Lan79) but has no editing capabilities. There is one map associated with each window and it simply plays the hardware terminal character buffer role for its virtual counterpart.

In BRUWIN each virtual terminal emulator also keeps information about the cursor position relative to its window and the number of characters already in it. There are three routines belonging to the interface between vte and the display manager: ***x_map***, ***y_map*** and ***puttext***. The first two routines correlate the virtual terminal cursor position to the physical device coordinates relative to the screen. Based on the correlated value, the routine puttext calculates the cursor address and calls a system *write* routine to insert a piece of text at that address.

The virtual terminal in BRUWIN is implemented with two (unidirectional) pipes which constitute the read and the write communication paths between the control program called the Task Mediator and the user window process. The child's (the window process) standard channels are redirected to the pipes.

Consequently in the pipe BRUWIN version (Mey81), the vte is split into two parts: a ***vte_input*** and a ***vte_output*** routine performing the input and output character interpretation respectively as their RIG VTMS counterparts.

The pipe approach for implementing virtual terminals is also used in WSH (Bre84), CWSH (Wei85), Wm (Jac84), VT (Gam84) and WOW (Lac87).

In VT (Gam84), the virtual terminal is implemented with the above-mentioned port mechanism. The disadvantage is that the port is not commonly provided in different Unix versions. Hence the proposal of replacing it in future VT versions with an mpx call. The effect of mpx is to create a multiplexed file, but even this mechanism seems obsolete presently as explained further on.

In WSH (Bre84), the virtual terminal emulator has to interpret characters associated with window and process control in addition to those needed in the conventional UNIX terminal environment as done in BRUWIN, due to the integrated

shell approach. Consequently it needs a more complex data structure which consists of:
-a ***display buffer*** and an ***input queue*** which are involved in updating functions and
-a ***display buffer pointer*** and a ***cursor*** which are used for mapping the buffer contents to the coordinate system of the controlling shell; the cursor structure is also affected when updating is done (insert line, scroll up, delete character, etc.).

While pipes are an elegant UNIX way of substituting the standard terminal I/O channels for the purpose of controlling concurrent processes, they do not have the features neccessary to run programs with access to real terminals (such as stty). Consequently window managers implementing the virtual terminals communication paths with pipes (WSH (Bre84), Wm (Jac84), CWSH (Wei85)) can not run programs which use "raw mode" such as most of the screen editors. This is a major limitation for a successful window system as users work with screen editors most of the time.

The CWSH system in its original version solves this problem by emulating the kernel functions in user mode via the UNIX "multiplexed files" mechanism which allows the interception of control characters at an intermediate user level through special UNIX "ioctl" (input-output control) calls. Based on this, CWSH attempts to intercept all UNIX kernel operations referring to I/O and interprets them relative to the window process making the call. Consequently for each window, CWSH duplicates the line disciplines at the user level. While this method also allows user programs running in windows to access window control functions via the special ioctl calls mentioned above, a feature which is not available in other window systems like BRUWIN or WSH, it still appears to be a complicated implementation. Moreover the "multiplexed files" mechanism is no longer provided in the recent UNIX releases, but a derivative of it called pty (pseudo terminal driver) (Pik84) has been made available. While intended to be used in future versions of CWSH, the pty interprocess communication method is already in use in a newer version of BRUWIN (Mey81), wm.v42 (Jac84) for Berkeley 4.2 UNIX and WINDOWS (Uni86) for UNIX BSD 4.3, both running on the VAX 780.

The pty driver (Unib) provides support for a device-pair of character devices, one called a ***master***, the other a ***slave*** and together a ***pty***.

The slave device supplies to the processes an interface identical to that of a real terminal but with no hardware support. The pty offers a bidirectional communication path similar to what a "bidirectional pipe" might be. Whatever is written on the master is given as input to the slave and vice-versa. In addition to pipes, the ptys can handle echoing and line editing (Jac84). Along with a number of ioctl calls applied to them, they provide a full Unix terminal environment in each window thus eliminating the need of duplicating these functions as done in CWSH.

There is a fixed number of pseudo terminals determined at the configuration time, the default being 32. As they are a system wide resource (Jac84), they put a limitation on the total number of windows in use.

Finally virtual terminals also use sockets for interprocess communication as provided in BSD 4.3 WINDOWS. This approach is useful for processes located on different machines. The obvious application is for windowing in a computer network.

Chapter 3

Implementations of Windows

3.1. General Concept of User-Space Implementation for Windowing.

Each window is normally associated to a process spawned by a control program which needs a "virtual terminal" (also called pseudo terminal or even phantom terminal) to communicate with the process.

Each window process may be a control process as mentioned in Appendix 3. A control process keeps track of the states of its descendant processes which usually correspond to different user commands.

A virtual terminal may be defined as a non-hardware terminal which behaves the same as a real character device with echoing, CR/NL mapping and interrupt catching being handled by the driver. A virtual terminal is obtained by establishing a software connection associated with some data structures initialized appropriately. The design of a virtual terminal therefore depends on the interprocess facilities of the UNIX version in question and to a certain extent on the designer's decision (see distinction between WOW and OUR APPROACH).

In a window facility there is one physical port corresponding to the terminal and a multiple number of virtual terminals, one for each created window. The basic functions of the control program are:

1.Multiplexing when transmitting output from the virtual terminals to the physical terminal
2.Demultiplexing when transmitting input from the physical terminal to the correct virtual terminal.

In exercising these functions, the control program needs extra information to identify the window. This could be an ID number in a header which is either added (if multiplexing) or stripped off (if demultiplexing) according to the direction of communication. The control program also needs the following functions:

1.to distinguish between control information related to the window operations and data coming for/from the windows

2.to poll all the software connections.

We will see in the following the distinctions in achieving these functions in three different designs of a window facility: in Blit (Pik84), WINDOWS (Uni86) and WOW (Lac87). The window facility may be provided in two ways:

1.The user logs into the cshell as in the original system and he may choose to work with the regular screen or to invoke the window facility as an application program (available in one of the directories for instance /bin).
2.The user is automatically logged in to the window facility.

Obviously the first possibility is to be preferred as the user has more flexibility in choosing what best fits his needs. In addition, no changes are involved in the regular log-in procedure described in Appendix 3.

In our proposal (see Chap.5), the window facility allows the user to open and work with a number of windows which is limited by the number of window terminal structures in the system. The system operates on a first come, first served basis. The user of the window facility may take the maximum number of windows if so he wishes and if no one else has accessed them. The approach is valid on the assumption that a user will not open windows without needing them. This assumption is reasonable as a window facility available to a number of different terminals on a system is appropriate in an environment where users are working in a team or at least with a reasonable degree of cooperation.

Two distinct windowing approaches are presented and compared in the following chapters:

1.One approach considers the implementation of window facility at the application level. This is illustrated with two applications from the outside world, Blit (Pik84), WINDOWS (Uni86) and with a Concordia University experiment, WOW (Lac87) which has been conceptually tested to a certain extent (more work would have to be done to make all the commands work).
2.A second approach referred to as Our Approach considers two alternatives for the implementation of a window facility in the driver itself (kernel) taking into account the multiplexing capability of Q/DH device.

3.2. Windowing Architecture in the Blit Application.

An alternative scheme for the I/O system called streams has been proposed and implemented ((Rit84b), (Uni87a,b)). Its use in multiplexing virtual terminals appears in the Blit application (Pik84) and it is also summarized in (Bac86). Although conceived for a bitmap graphics terminal called Blit, the software multi-window environment can be used for ASCII terminals as well (Bac86).

The stream concept represents a reorganization of the UNIX character device driver with the purpose of increasing the efficiency due to the elimination of the clists of the original driver. It was also intended to give a significant gain in modularity.

A ***stream*** consists of a set of linearly connected modules. It has two ends, the interface to the system on the process side, and the stream driver. Both ends are opened when the terminal device is opened. Processes can "push" and "pop" modules onto or from an opened stream through appropriate ioctl() calls, a feature which offers great flexibility in combining modules according to the need of the application.

Each module consists of a pair of queue structures, one for input and the other for output. The modules communicate by passing messages through a "put" procedure as part of the queue structure from one module to another in the stream. While in a module queue, a message is processed by a "service" procedure which is scheduled when the queue is enabled.

The idea behind the software multiplexer in Blit is the following: the control program called mpx spawns a process for each user created window. From this child process, a cshell is executed which will communicate with the parent mpx through a previously set ***pseudo-terminal*** (pty). Ptys in Blit are software character devices which are added to the stream concept in order to transmit data and control messages between processes in both directions. Ptys operate in odd-even pairs which can be found as pty file names in the /dev directory. When two processes initiate communication, each has to open a member of the pty file pair, after which the data written on the odd member (also called the master) is sent to the input of the even member (also called the slave) and vice-versa (Unib). At most one process can open the master, but more than one process may open the corresponding even-numbered file in order to communicate with the master. As ptys are just transmitters of messages with no terminal processsing mechanisms attached to them, they need to be assisted by line discipline modules, ttyld, (as illustrated in Fig.3.1.(Bac86)) which achieve the terminal settings for the virtual connection and which are pushed through an ioctl() call in the child before executing the cshell.

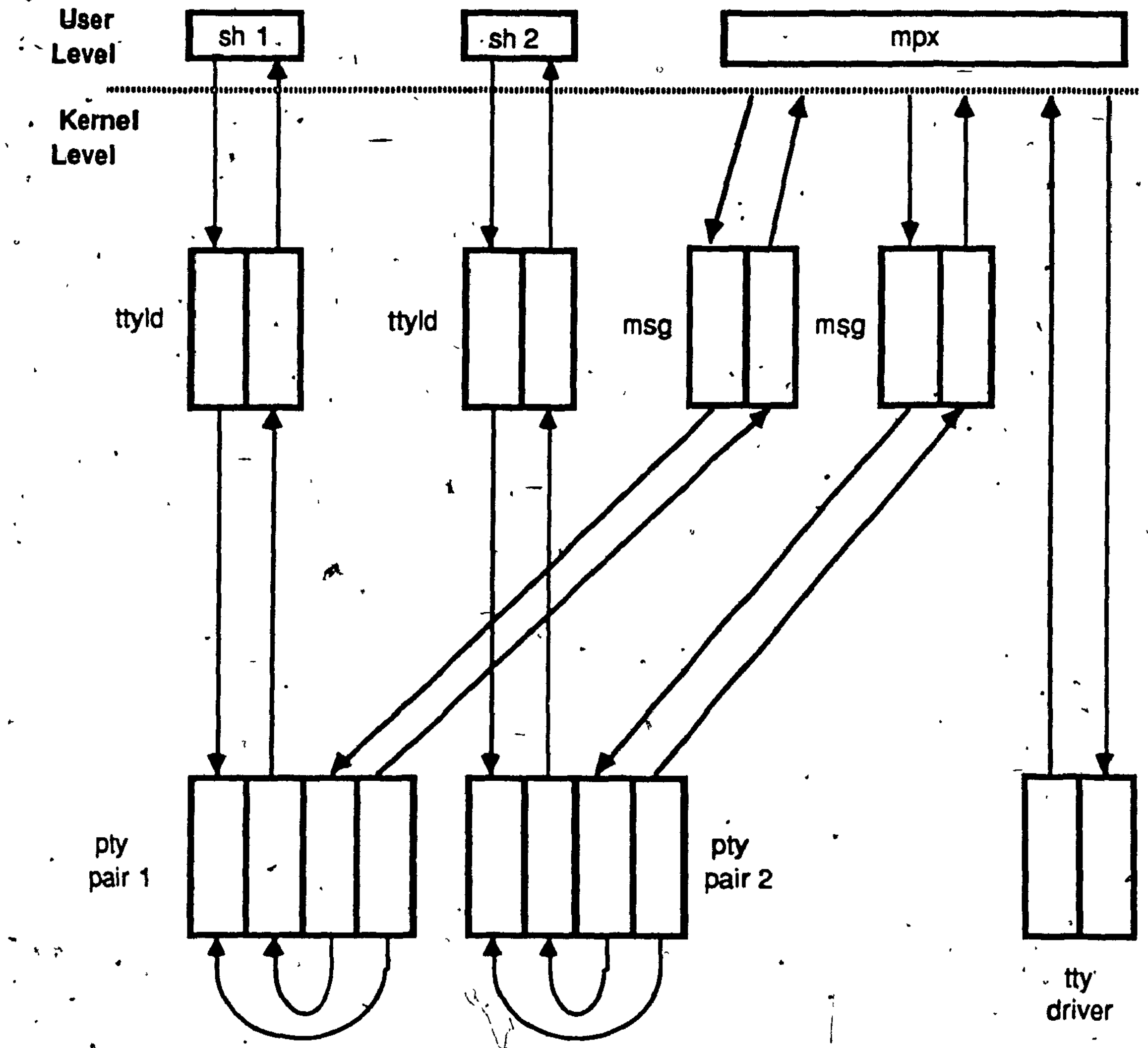

Fig.3.1
Windowing Virtual Terminals on a Physical Terminal (Bac86)

The parent mpx needs to push message discipline modules which have the role of converting the control messages generated by ioctl() calls into data messages prefixed by a header identifying the type of the message. The data messages obtained as above are to be read or written by mpx, reformatted with the information about the virtual terminal ID and transmitted further to the physical tty.

Mpx.c is invoked as a mpx command by the user after he has successfully logged in. Mpx is designed as an infinite loop which does an I/O polling based on a *select()* system call used in a nonblocking mode. The *select()* is followed by a *read*() which reads the physical line. Mpx distinguishes between control commands (such as creation of a window, switching control to another window, etc.) and data coming up for or from the virtual tty. In the first case it takes appropriate action according to the respective command, while in the second case it accomplishes its demultiplexing, respectively multiplexing function based on the id of the pseudo-terminal contained in a header.

The Blit architecture is of interest to us mainly as a predecessor of other software multiplexers, in particular WINDOWS (Uni86). Its full architecture differs in that it is designed to drive intelligent terminals which handle a part of the window software.

3.3. Windowing Architecture in UNIX BSD 4.3.

Unix BSD 4.3 implements a window facility called WINDOWS on ASCII terminals which is achieved as an application program invoked by the user as a "window" command possibly followed by some optional arguments. If no arguments are specified, by default two equally sized windows appear on the screen, each one with a cshell prompt waiting for a user command to execute (Uni86).

Windows can overlap and are framed as needed. There are two types of windows in this facility:

1. User windows which are created by user typed commands. Each one is associated with a cshell process which communicates with the parent control program through a virtual terminal either of type pty or socket pair according to the user request, the default being pty. User windows may be in the foreground or not. Windows have system-assigned integer IDs from 1 to 9 displayed along with an optional user defined label string on the top edge of its frame.
2. Information windows which are used for displaying error messages and other information which are always in the foreground.

3.3.1 Data Structures

Visible windows are maintained as a doubly linked list wwhead of window structures ww_w (because of overlapping information !). Each window has two elements:

1. A cursor which is positioned relative to the window size. The cursor is defined as a structure containing two integer coordinates, one for rows and one for

columns, with values updated from the structure ww_w. This structure in turn contains six integers, two for the window size representing the number of rows and columns, and the other four for the position namely top, bottom, left and right. The ww_w structure is initialized upon the opening of a window based on either default values or on the parameter values of the respective call in case of a long command.

2.A text buffer which may contain if so specified more lines than the window itself, its default value being 512 bytes.

The main structure in this program is a window structure, ww, which contains information necessary for the following purposes:

-to manipulate the window and cursor size and position

-to support a set of control functions such as line and character insertion and deletion, mapping of \n to \r\n, etc., which are performed by functions belonging to the tty driver

-to describe the type of virtual terminal in use (whether pty or sockets, its fd, etc.

-to achieve I/O appropriately. For this purpose an output buffer is provided which holds the data available from the virtual terminals and which will be written to the screen after being processed by the lowest level driver routines.

3.3.2 Modes of Operation

The window program operates in two modes: command and conversation.

In command mode user keyboard data is interpreted as window commands. They can be of two kinds: long (a sequence of meaningful characters) or short (one character). The commands are echoed on the first row of the screen representing the command window previously "opened" in main.c. When referring to system windows, "opening" translates into the allocation of a window structure, the initialization of those fields specific to the window type which may be command, frame or box window and displaying of a rectangular box corresponding to the opened window on the screen. When referring to a user window, "opening" in addition of the above actions involves the following steps:

-**making** the window current which implies the following operations:

-to select the window

-to bring it on the top of all windows

-to update it

-to highlight its identifier and label in reverse video

-**spawning** a child process with a cshell environment by calling *wwspawn*() routine which

-spawns a child with using *vfork*() for efficiency

-executes a cshell from the child which will prompt the user on the screen waiting for commands

-**storing** the pid of the child in the appropriate field of the window structure

-**marking** the window state field in the window structure as having a process (WWS_HASPROC). It is this field which when tested in mloop.c will automatically bring the program into command mode when the spawned process dies.

In conversation mode two events may happen:

1.**The outputs from the pseudo terminals are copied to the respective output buffers in the window structure and from there to the terminal windows**. This is done in the routine wwiomux() which is the terminal output handler and which is interrupted any time user input is detected.

2.**User input data is sent to the current window**. There are three commands referring to the current window:

-**typing** the respective **window id** which selects it as current window and returns to conversation mode

-**typing %** character followed by the respective window ID which selects it as the current window but remains in command mode.

-**typing** a cntrl ^ which selects the previous window and returns to conversation mode.

Typing an escape character while in conversation mode switches to command mode.

3.3.3 Program Operation

The heart of the program which ensures the I/O synchronization is achieved by two routines, *wwiomux*() and *wrint*(), in mloop.c. The synchronization mechanism is based on the *select*() system call in combination with *setjmp*() and *longjmp*() system calls as described further on. The keyboard typed characters are entered by *wwrint*() into a linear input buffer and later taken out either by *wwgetc*() routine called from *docmd*() in command mode and interpreted as commands or by a *write*() system call using the input pty file descriptor corresponding to the current window in conversation mode. *Wwiomux*() monitors the input keyboard buffer and whenever it detects that it is non-empty, it returns to the point of the call letting *wwrint*() to handle the tty input. If no input is present *wwiomux*() continues its job as output handler. Due to this behaviour, *wwiomux*() is invoked in both command and conversation mode, thus exercising its role of multiplexing/demultiplexing as suggested by its name.

Wwiomux() carries out its multiplexing function when it polls all pseudo-terminals in order to detect those having outputs for their windows in which case it writes them to the terminal (many to one !). Its demultiplexing function is accomplished when it returns to let *wwrint*() to handle the one input from the terminal which is to be directed by additional code in mloop.c to the current window (one of many!).

This functionality is ensured in *wwiomux*() by use of a *select*() system call which has as timeout value a function which according of a previously set flag called "noblock", returns either a pointer to a zero-valued "timeval" structure tv, thus ensuring a non-blocking poll behaviour of *select*() <u>or</u> a zero pointer to the same structure which determines *select*() to block.

The non-blocking poll occurs when at least one of the output buffers of the existing user window structures has data for its terminal window and the window can accept it. In this particular use, *select*() polls the process file descriptor set and returns its number of ready-to-read pseudo-terminal file descriptors. The file descriptor set has been formed prior to *select*() call by scanning the list of user created windows which have room in their respective output buffers. The purpose of the non-blocking *select*() is to allow the reading of any pseudo tty output that may have arrived while the already existing data in the window buffers are written on the terminal. This operation is done in a non-interruptable *wwwrite*() routine also called from *wwiomux*().

Wwwrite() tests one character at a time from the window buffer. According to the type of character, special, control, extended or regular, it will take appropriate action concerning the cursor position in the window (screen updating, scrolling, etc.).

In performing special functions such as character and line insertion and deletion to the terminal windows, *wwrite*() is assisted by low level routines having pointers in a tt structure (defined in tt.h) which represents the interface between the window package and the terminal driver. In addition, three other functions, *ttflush*(), *ttputs*() and *ttwrite*(), in ttoutput.c are needed for buffered output.

The blocking *select*() occurs when all window output buffers are empty in which case the "noblock" flag remains initialized to 0. Under this condition, a *setjmp*() call is done prior to *select*(). If *setjmp*() executes correctly, it returns a zero value. Subsequently the flag "wwsetjmp" is set to 1. If "wwsetjmp" is set and input data is present, *wwrint*() enters a portion of code which executes a *longjmp*(). Its purpose is to restore the environment saved in wwjmpbuf by the last call to *setjmp*() in such a way that execution continues from wwiomux after *setjmp*(). In this case it means that *wwiomux*() returns or in other words gets out of the blocking *select*(). The portion of code illustrating the above behaviour is given below.

```
if (!noblock) {
........
  if(setjmp(wwjmpbuf))
    return; /*i.e. return is executed after the return from longjmp() in wwrint */
  wwsetjmp = 1;  /* this occurs after the return of setjmp() which
                      normally returns a 0 */
.........
n = select(wwdtablesize, &imask, (fd_set*)0, (fd_set*)0,
     noblock ? &tv : (structure timeval *)0);
```

The structure imask in the above statement contains pty file descriptors corresponding to those window structures which have room in their output buffer. It is set with a function *FD_SET* (w - >ww_pty, &imask) which loads the pty file descriptors which are ready in &imask.

In addition to the above role *wwrint*(), the input interrupt handler, does the following:

-when called, it waits for input data from the terminal due to the setting of FNDELAY flag in a *fcntl*() call

-if data is available, it appends it at the current write position indicated by a pointer "wwiq" in the input buffer

-it then resets the window flags such that subsequent reads (in other functions) from the terminal are non-blocked.

3.4. WOW - A Local Experiment.

WOW is a Concordia project which implements the window facility outside a standard driver in a control program. On a user request, it creates a number of windows for him (within a predefined limit) and executes user commands from those windows.

To achieve this, the control program multiplexes in software the physical connection for the user's terminal and creates a virtual terminal for each such software connection.

In WOW, a virtual terminal appears on the real terminal screen as a small rectangular box accessible to the user for reading and writing. This is what is conventionally called a window and represents the mapping of a memory area to the monitor screen. The mapping in WOW is done through the use of the curses package supplied with UNIX V which provides most of the low level window functions of the cursor control. The Concordia project, WOW, added in new functions for window manipulation such as:

- change a window size within a minimum (5 * 8 ") and a maximum (full screen) limits
- move the top window across the screen
- create a new window on the top of an old one while saving the contents of the latter
- reactivate the lowest window to the top
- remove a window from the screen without logging off by putting it to sleep (in the background). The background window becomes an icon in reverse video in the left corner of the screen with its id number on it. Its text and area become invisible and they are saved in the so-called phantom window, which is invisible to the user and is located below the icon window
- reactivate a background window by placing the cursor over the icon and pressing the <ENTER> key
- send a message or a command from a source window which is to be activated if not already active to a receiver window, both identified by their IDs
- logout which releases a phantom terminal identified by its ID number and removes the window from the screen
- exit which finishes the control program after logging out all phantom terminals previously opened and
- help which gives indications about the usage of the window facility.

The above commands appear in a menu window in the upper left corner when the user invokes the window facility. The menu displays alternative control characters for each existing command providing the user with the choice of giving a command, namely either by using a distinct control character for each command or placing the cursor on the corresponding menu line and pressing ~~the~~ <ENTER> key.

Besides the window types mentioned previously there also exists a two-line message window at the bottom of the screen which guides the user through messages and instructions. The total screen area is divided as shown in Fig.3.2. below.

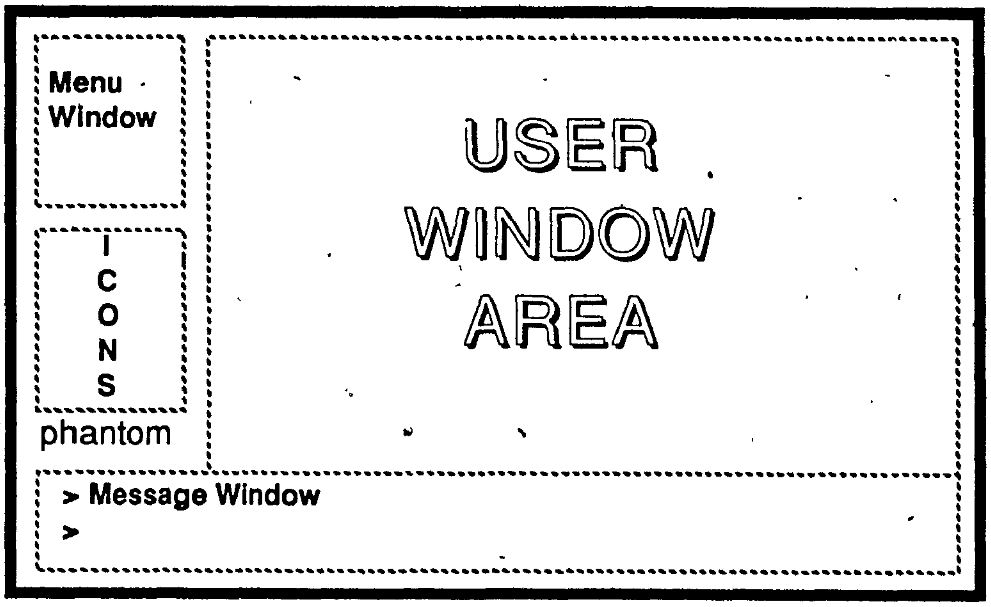

Fig 3.2

The Screen Layout in WOW (Lac87)

The following material is to be read in relation to Fig.3.3.

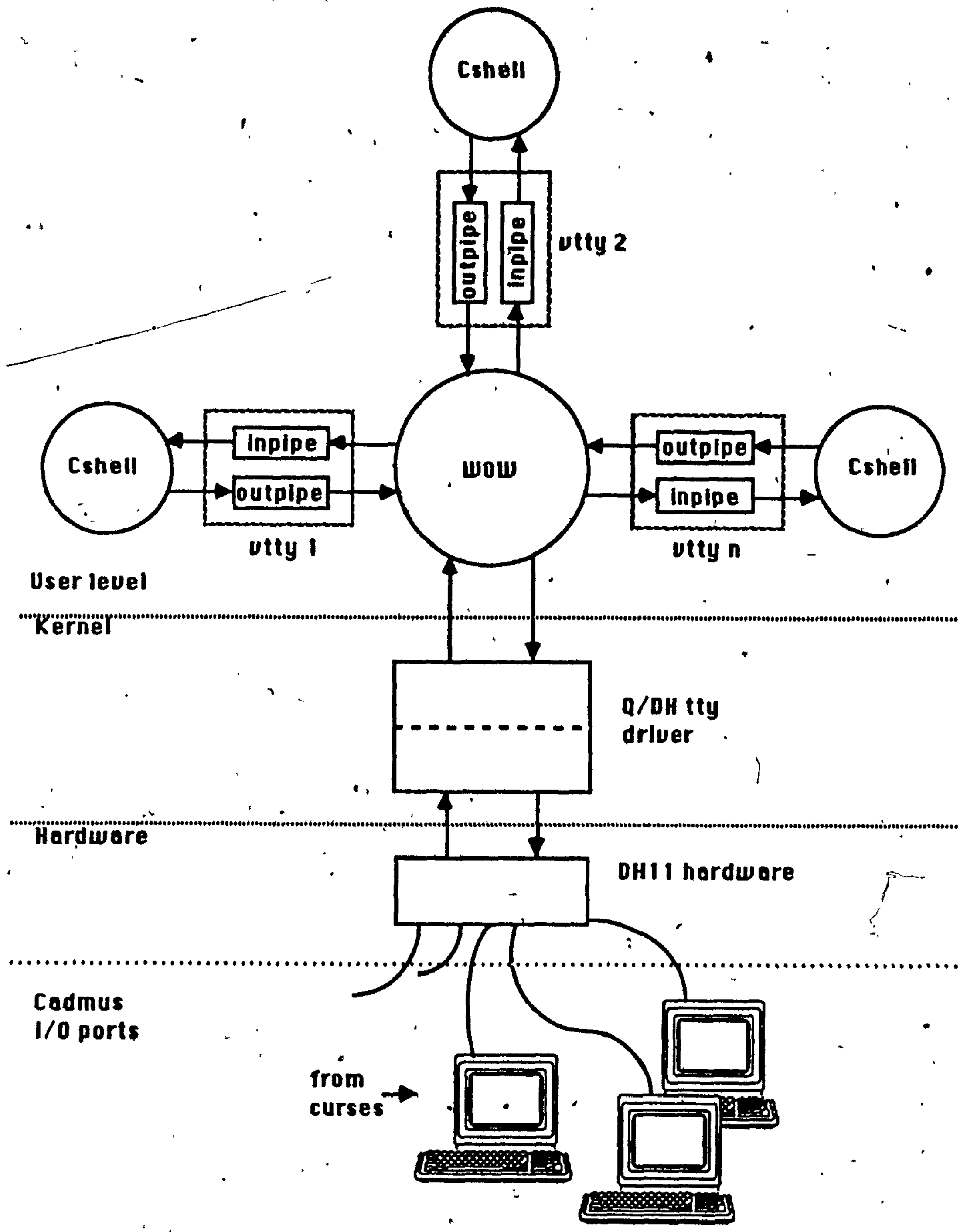

Figure 3.3
Implementation Detail in WOW

Internally a phantom terminal in WOW is obtained in the following way. For each window number requested (i.e., typed in by the user), the control program also referred to as the parent process takes the following actions:

1.Opens two pipes called inpipe and outpipe using the pipe() system call
2.Forks a child
3.Finds a free entry for the user window number in the table data structure described below and fills it in with the appropriate information.

The child does the following:

1. Connects the four file descriptors of the pipes to a vtty structure by using an *ioctl*() system call of the form *ioctl*(valid_file_desc, PIOCVTTY, &vttyb). Following this call and by appropriately redirecting the two pipes' file descriptors, vttyb.inpipe and vttyb.outpipe will correspond to the input, respectively to the output side of the phantom terminal.
2.Executes a cshell which if successful prompts the user waiting for his commands. Now, the parent and the child communicate through the two pipes as described below. The child will transmit the cshell commands to the parent by writing to the outpipe. The outpipe is continously monitored (in *check_output*() routine) by the parent through a non-blocking read achieved with a previous *fcntl*() system call of the form:
fcntl(p.outpipe(RD),F_SETFL, *fcntl*(p.outpipe(RD), F_GETFL,0) | O_NDELAY). The parent will transmit the results of the cshell commands (in *write_to_term*() routine) to the child by writing the inpipe. The inpipe is read by the child (the cshell) only when the information is available. This occurs due to a blocking read achieved by restoring the initial (before forking !) file_flags through another *fcntl*() on stdin this time. The *fcntl*() system call appropriately used, achieves the synchronization between the parent and the child processes.

The main data structures (see Fig.3.4. below) used by the control program are:
-a WINDOW structure defined in curses.h with fields holding information about the current cursor (x,y) coordinates, the maximum values (max x, max y) measured relative to the starting coordinates (beg x, beg y), the address of the first and last character, etc.
-a VTERM structure defined in wow.h with fields containing information related to
-the window id
-whether the window is put on background

-the packet addressed or issued to/from that window which is stored in a buffer of maximum 256 characters

-the pointer to the last character in the buffer

-the pointer to the WINDOW structure

-<u>an array</u>, vt_stk(MAXDEPTH) of pointers to VTERM structure defined in wow.h acting as a window stack with position 0 always assigned to the current window

-<u>an array</u>, table(MAXDEPTH)(TABLESIZE) of integers with as many entries as windows allowed in the system, with each entry holding information about:

-the window id

-the file descriptor of the outpipe

-the process id spawned by the control program for each newly created window.

We may notice in Fig.3.4 that the same window id appears in two places, in the VTERM structure and in the corresponding "table" entry. This id represents the software link between the window and the process created for it. This pair is treated as being the virtual terminal in WOW.

The <u>control program</u> consists of:

-an initialization part which is responsible for the following:

-terminal screen initialization

-creation and initialization of the menu, message and background icon windows

-global variables and main data structures initialization

-a continuous monitoring loop waiting for a user command from those displayed in the menu.

Software functions are provided (*getkbd()*, *wgetch*(win), etc) to determine whether the user-typed character is coming from the arrow keys, or corresponds to one of the control characters from the menu, or is a regular character to be appended to the window. A function *whichwindow()* extracts the current cursor position and, based on the appropriate pointer information, determines whether the cursor is in one of the menu, background, message or user windows. It returns the index of the corresponding WINDOW structure.

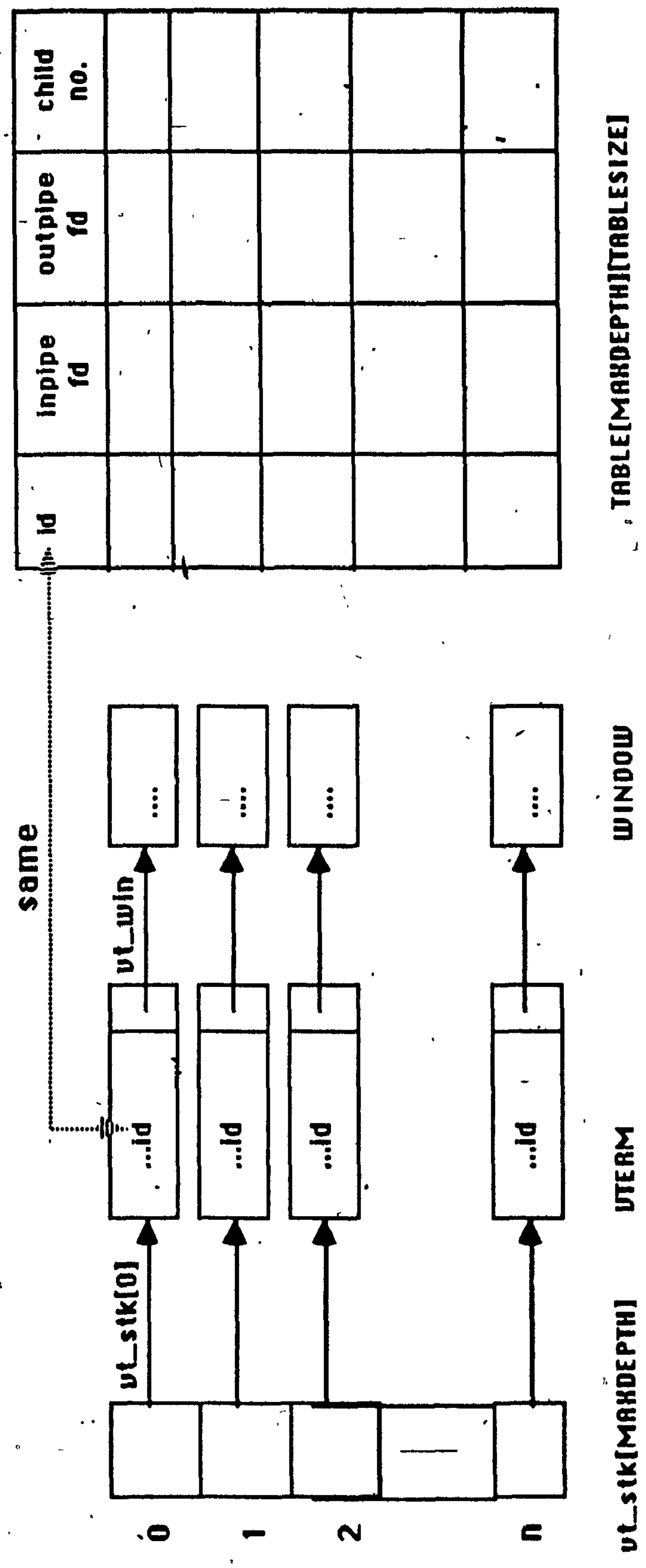

Figure 3.4 WOW Control Structures

Chapter 4
The Terminal Control Approach in UNIX System V.

4.1. The Present Architecture of the Terminal Driver.

This section presents a discussion of the architecture of the Unix driver for a terminal multiplexer board. This driver is discussed in detail to provide the background for the presentation of the window manager architecture proposed in this work. The internal multiplexing capability of this driver can provide the basis of many of the functions required by the windowing system presented in chapter 5. The material presented here is based on the Unix system V implementation of the tty dh driver (tty multiplexer board) for the Cadmus workstation. A documented version of this driver was previously produced as a project report submitted by D. Brown (Bro86).

The Q/DH driver is a kernel module serving as a software interface between user processes and the terminal on which they perform I/O operations, as illustrated by Fig.4.1. It controls the number of terminal devices defined by the value of DH11 in the conf.h file. In the present configuration 8 terminals are connected.

The driver routines are organized in two halves which communicate through the tty structures, mainly the raw and canonical input queues and the output queue, all maintained as clists.

The driver routines belonging to the upper half represent the user interface to the system and are called from processes by searching the character device switch table. They contain code to copy from the user buffer to the appropriate queue, and vice versa.

The driver routines belonging to the lower half represent the interface to the hardware control which handles interrupts and communicates with the machine. They are called by the operating system through the device interrupt routines or less frequently by the upper half routines. An example of the latter is the initiation of the transmission of the first block from the output queue as it will be seen in section 4.1.3.

The design of the driver in two halves is a classical organization used by Unix for character device drivers. It has the following advantages:

- It allows an inherent I/O synchronization. The upper half puts processes to sleep when the lower half is not ready to service the I/O requests. When the I/O

operation is completed through interrupt control, the upper half is woken up. If no input is available the process remains blocked (asleep) in the upper half.
-It permits a separation between normal processing done by upper-half routines and hardware interrupts manipulated by the lower half routines.

UNIX treats a peripheral device, whether block or character, as a special file. There is one such file name for each I/O device in the system appearing in the directory /dev. For example, the file for a terminal is referred to as /dev/tty#, for a line printer as /dev/lptr, etc. This treatment of an I/O device has the following consequences in the design of the driver:

-The peripheral device driver must be designed to respond to the file access operations: *open, close, read, write* and *ioctl.*
-Each peripheral device corresponds to an inode created through the *mknod* system call. In the case of the creation of a special file corresponding to a character device, this call is used to allocate an inode containing in its file type field the specification "character device" and in its block pointer field the major and the minor numbers defined by the superuser.

In the case of a character device, the major number is an integer used to index the character device switch table. For each entry, this table contains pointers to a set of driver routines, for example *dhopen, dhclose, dhread, dhwrite* and *dhioctl* for the dh driver. The major number specifies the driver type while the minor number is passed as a parameter to the above-mentioned routines and serves to select from the set of data structures for the physical tty ports found in the array dh_tty().

As with any file, in order to be able to do operations such as reading, writing or controlling the device, the system needs to access the inode for the special file from the system calls being made. This can be done only if the respective file has been opened with an *open()* system call. The effect of *open()* is twofold:
- it creates the linkage to the file inode (created previously by *mknod* system call) through the three kernel data structures as shown in Fig.4.1.
- -it returns (if successful) the file descriptor fd which indexes the array of the file descriptor entries in the u_area. Fd is used in subsequent system calls (*read, write*, etc.) to access the inode for the file through the chain of pointers

as in Fig. 4.1. In particular, for open, the upper half routine *dhopen()* is called which does the following:

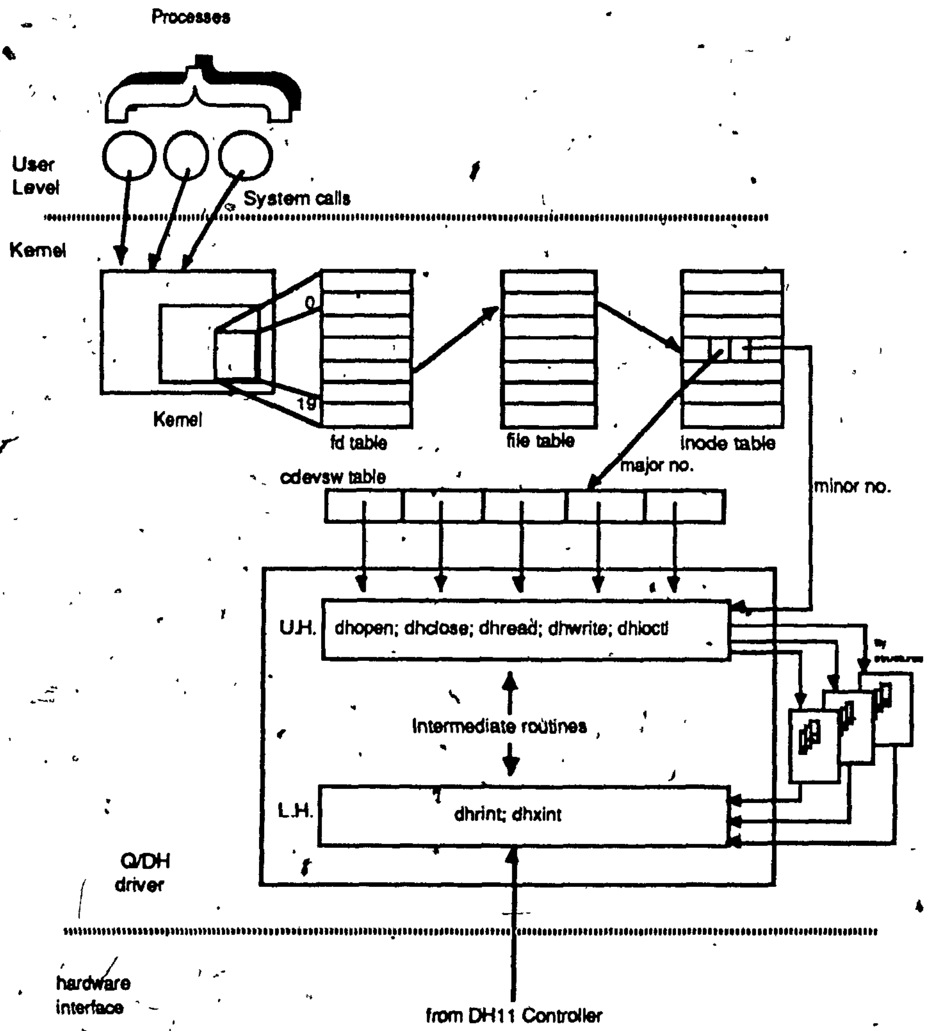

Fig 4.1
Kernel Data Structures for Driver Linkage

-it establishes a software connection between the calling process and the opened device by accessing the appropriate dh_tty() entry
-it initializes the terminal control modes

-it sets a hardware register in Q/DH device called lpr (line parameter register) based on the control modes

-it puts the process to sleep in the case that no hardware connection is detected

-when wakenup, it calls a line discipline routine to allocate a receiver buffer in tty structure.

In practice user programs do not usually open these terminal files. They are normally opened by the getty process in the login procedure (described in Appendix 3). Getty invokes open system calls from a standard library function fopen to set up the standard input, standard output and standard error fd's.

4.2. The Main Data Structures of the Terminal Driver.

The driver routines and the main structures they are using are spread over a few files:

-dh.c containing the upper and lower half routines, which are device dependent

-tt0.c containing the line discipline routines which interpret input and output

-tty.c containing some general driver routines for the device independent code in support of the routines in dh.c and tt0.c files

-tty.h containing the basic structures needed for normal terminal I/O.

The main data structure is called tty and it is described in appendix 1.

4.3. The Implicit Multiplexing/Demultiplexing Feature of the Q/DH Driver.

Fig.4.2. shows a simplified diagram of the driver data flow for both input and output emphasizing the main routines which participate in data transmission from user space to the interface, and vice versa.

On the left hand side of Fig.4.2. we see a read system call belonging to a user program which requests a read of n characters from the terminal (stdin) into the program buffer buff.

After retrieving the major and the minor numbers from the inode corresponding to stdin, the system call *read* invokes the upper half driver routine *dhread* with the minor number dev as parameter. Based on dev, *dhread* sets a pointer tp to the corresponding tty structure and calls the upper half line discipline routine *ttread* with tp as a parameter.

The major function of *ttread* is to copy data from the canonical queue into the process user space at the location u.u_base. If the canonical queue is empty, *ttread* calls a support function *canon* which checks the raw queue to see whether it is empty or not.

If the raw queue is non-empty and the line discipline flag ICANON is set indicating that canonical processing is requested, *canon* will transfer characters from the raw queue to the canonical queue until it encounters a delimiter such as newline or end of file. If ICANON is not set, *canon* will satisfy the read requests directly from the raw queue if at least MIN characters have been received on this queue.

If the raw queue is empty, *canon* puts the process to sleep on the raw queue. The process will be awakened later on by a low level line discipline routine *ttin* when a delimiter has been encountered. This condition allows fast bursts of input characters in the case of a read requesting a number of bytes exceeding a cblock size CLSIZE. This improves the efficiency of the transfer which normally is done character by character. The routine *ttin* is called by the receiver interrupt routine *dhrint*.

After copying all the characters in a cblock from the canonical queue to the user's memory space, *ttread* returns it to the cfreelist.

Looking on the interface side, the Q/DH device driver polls all terminal lines to see if they have user typed characters. If yes, the hardware stores any character found from a terminal in an interface buffer called silo, in a two byte field. The higher byte contains in bit positions 8-11 the line number on which the character has been received and the lower byte contains the character itself.

When the level of the silo register exceeds the value set in the silo status register, the hardware transfers one character from the silo to the nxch register, setting bit 15 of the latter to 1 to indicate that the character is valid. It then sends an input interrupt. The system interrupt handler, in turn issues a call to the low-level driver interrupt routine *dhrint*.

The routine *dhrint* gets the register set base address in order to access the contents of nxch register. If the character in this register is valid, *dhrint* starts a loop in which it reads and processes one character at a time as long as the condition holds. For each valid character, *dhrint* extracts the terminal number from bits 8-11 of the upper byte to select the tty structure corresponding to that terminal and sets a pointer tp to it. This pointer is subsequently used for the following purposes:

-to access other fields on the same tty structure in order to take appropriate action according to the information contained in those fields and to the meaning of the character. For example it might resume or suspend scrolling or strip off or not the character to 7 bits depending on the input modes of the termio structure.

-to access the receiver buffer called rxbuff in Fig.4.2. through the t_rbuf control blo~~ck of the~~ tty structure in order to put the character in.

-as a parameter to the low level line discipline routine *ttin*.

The routine *ttin* extracts the character from the rx buffer, then uses the pointer tp to access the input control value t_iflag of the tty structure for the terminal. This value tells *ttin* how to process each input character depending on the setting of different bit flags before putting it on the raw input queue. For example if IUCLC is set, then it will do upper/lower case mapping or if INLCR is set, it will insert a new line character after a carriage return.

After placing the character on the raw queue, *ttin* accesses through the tp pointer the t_lflag of the tty structure, which it is used to control the terminal line discipline functions.

If ISIG is set and the input character matches one of the control characters INTR or QUIT, *ttin* performs the function associated to that character, else if ISIG is not set no checking is done.

If ICANON is set, canonical processing is enabled. This refers to the treatment of erase and kill characters and the incrementation of the t_delct field of the tty structure when encountering a delimiter. If t_delct is non zero and the upper half is asleep in *canon* as mentioned previously, *ttin* will wake it up. Following this, *canon* behaves as for the case of a non-empty raw queue as already explained.

If ECHO is set, the characters are placed upon reception on the output queue. This is done through a call to *ttxputi* which saves the characters to be echoed on a temporary queue in case output is in progress at the time the echoing is requested. Otherwise *ttxputi* calls *ttxput* to put the characters on the output queue.

The routine *ttxput* looks at c_oflag field of the tty structure which specifies the treatment of output. We should mention that the routine *ttxput* is also called from *ttwrite* when writing to a terminal.

If OPOST is set, output characters are postprocessed as indicated in c_oflag. This means that delays may be added, tabs expanded, newlines mapped to carriage return, etc.. Otherwise, if OPOST is not set, the characters are placed on the output queue without change.

If ECHO is set, after placing the input characters on the output queue, *ttin* also calls *dhproc()* with the command parameter T_OUTPUT in order to initiate the transmission of the first packet contained in a cblock from the head of the output queue. Additional explanation is given when describing the transmission part.

In summary, the multiplex/demultiplex functions for input characters take place in the flow of data as illustrated in Fig.4.2. The terminal inputs are multiplexed by the hardware of the interface card, with the characters being stored together in the silo register. These characters are acquired by the driver on interrupt, one by one, and are buffered in the receiver buffer rx. The receiving software of the driver described above demultiplexes them on the basis of bits 8-11, and distributes them to their individual queue sets, from which they are transferred to the respective user spaces by the upper half routines.

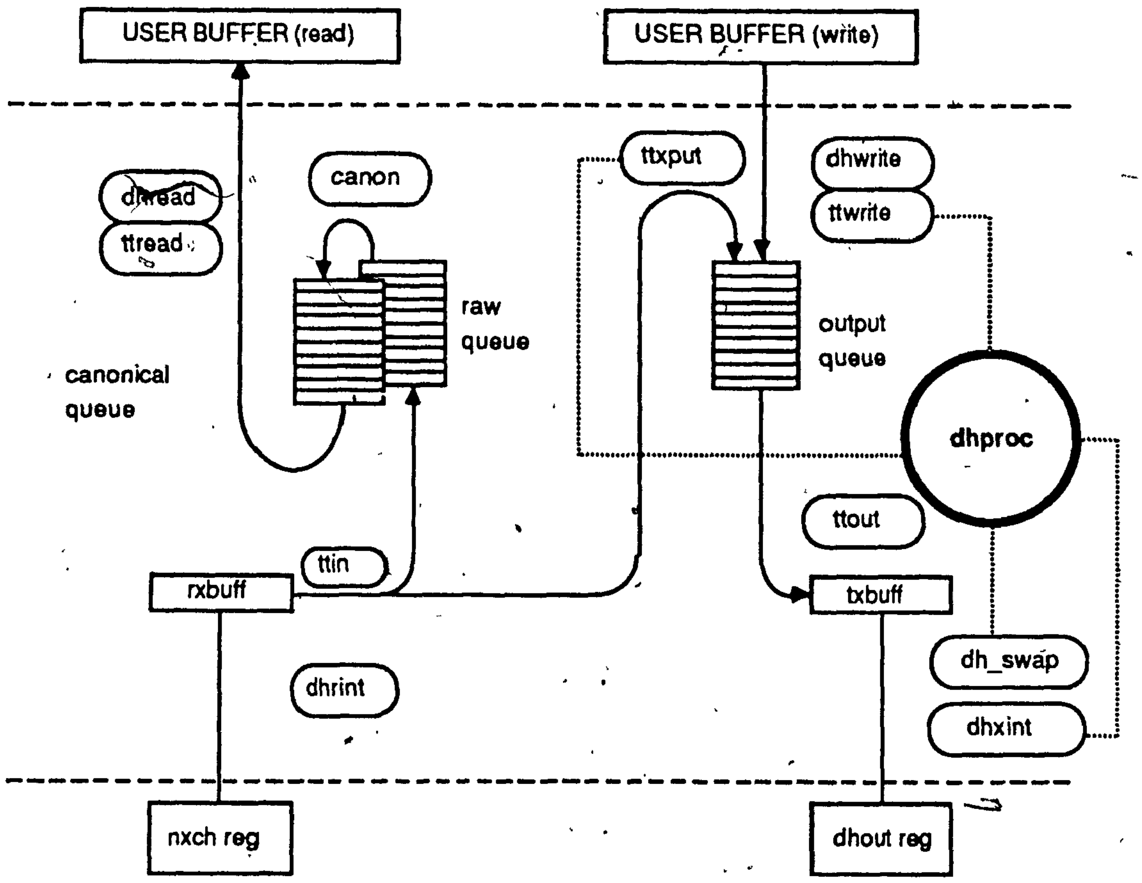

Fig 4.2
The Information Flow in the Q/DH Driver

The following material describes how transmission is accomplished. On the right hand side of Fig.4.2. we see a *write* system call belonging to a user program which requests to write on the terminal (stdout).

The *write* system call places the user data at the location u.u_base. Then it calls the upper half driver *dhwrite* which computes the tty entry corresponding to the terminal, after which it invokes the upper half line discipline routine *ttwrite*.

The routine *ttwrite* normally takes the user data from the location u.u_base, one character at a time and places it on the output queue by invoking *ttxput*. *Ttwrite* can also do block processing if the number of characters to be written is greater than half of a cblock. In this case it gets a free cblock from the cfreelist, it copies the number of characters from u.u_base into it and it calls *ttxput* to put the number of characters in the cblock rather than one single character on the output queue. The role of *ttxput* was presented above in the description of the receiving flow.

While placing character(s) from u.u_base to the output queue, *ttwrite* checks the high water mark (HWM) representing a system specified level of characters in the output queue. If it is exceeded, *ttwrite* takes appropriate action to shrink the queue by initiating the transmission of the first packet (a cblock) from its head. This is done by calling a lower half general purpose device dependent routine *dhproc* with the command parameter T_OUTPUT.

The routine *dhproc* is called from different places to do different jobs. If *dhproc* is BUSY doing other work, for example servicing another write call, or if the user typed a cntrl-s character to stop the output to his terminal, *dhproc* will not initiate transmission and it simply returns to the place of the call in *ttwrite*. Consequently *dhproc* will not shrink the output queue. Therefore *ttwrite* makes a second check of the HWM and if it is still exceeded, it goes to sleep on the output queue. *Ttwrite* is awakened from the low level line discipline *ttout* when the number of characters in the output queue drops below the low water mark (LWM). This system specified limit represents the level characters which triggers the wakeup of the routines to fill the queue. When awakened, *ttwrite* resumes execution of the *dhproc* routine with the command parameter T_OUTPUT.

The routine *dhproc* does the following:

1. Calls *ttout* in order to retrieve a cblock from the head of the output queue. It then stores the cblock in the transmission buffer txbuff of Fig.4.2. If needed, *ttout* will also wake up the upper half *ttwrite*.

2. Calls *dhswap* which copies the characters from the tx buffer into the low level transmission buffer dhout returning its address. While copying *dhswap* swaps

each pair of bytes, as the order of bytes in the interface is the reverse of that in the memory.

3. Sets the proper bits of the Q/DH interface registers so as to create the conditions for a subsequent 22 bit DMA transfer. It then marks the terminal line as being BUSY in order to allow the transmission to continue without interruption.

The next packet from the output queue will be transmitted when the lower half transmission routine *dhxint* is invoked. This occurs after the transmission of the first packet following a hardware transmission interrupt (bit 15 of the system control register set by Q/DH device).

The routine *dhxint* scans all the lines of the interrupting Q/DH device in order to determine which ones are between packet transmissions. For those lines, it calls *dhproc* with the argument T_OUTPUT to get the next packet and to initiate DMA transfer to the specified line.

In summary, the multiplex/demultiplex functions on output, can be characterized as illustrated in Fig.4.2. The outputs of the user processes are transferred by the upper half routines into the output queues of the respective terminals. These queues are read by lower half routines and are formatted and stored in the single output buffer dhout of the DH11 interface.

We may notice that for both input and output, the software connection is achieved by accessing the same tty structure for one same operation from both the upper and the lower half routines.

The inherent multiplexing/demultiplexing functions of the driver provide the support for a set of terminals controlled through a single physical channel (the I/O bus) to a single interface (the multiplexer board). The driver includes the code both to control the terminal and to interpret and map characters where appropriate. By extending this architecture to one further level of multiplexing many of the functions provided here may be used to control I/O through windows, which may be treated in a manner analogous to the handling of the instances of terminals in the driver presented above.

Chapter 5

The Design of the Window Driver

5.1. General Presentation of Our Approach.

This chapter presents a proposal for implementing a window facility for a UNIX System V facility. This study is based on the driver design for the Cadmus computer at Concordia University. The Cadmus driver is quite standard and its organization is typical for a tty driver.

The Cadmus driver, like most tty drivers in contemporary systems, drives a multiplexor board with a number of tty ports (8 in this case). The driver design reflects this fact, with a capacity for multiplexing and demultiplexing I/O from the board to the kernel data structures which represent instances of the tty driver. In this chapter, we present a design which extends this multiplexing to a second level: windows on each individual terminal associated with processes. We refer to this design concept as the **Window Driver**, as the major part of the window control functions are designed into a modified terminal driver.

In the Window Driver, each window is a pseudo-terminal represented by a modified kernel tty structure called wtty (see appendix 1) allocated from a common pool. This structure will be accessed by both upper and lower driver half routines in a manner similar to the method used in the Q/DH tty driver design. The details are discussed in section 5.3.

As each window is implemented through an instance of a modified tty structure, we need to associate to it a special file name, as is done for the physical ports. These window file names have to be opened in order to be used. To remain consistent with the structure of UNIX, the processes should be spawned from the user level. A user level window manager is therefore still required in this design to handle window initialization, termination and process spawning. We should still emphasize that in this approach the manager code will be rather small as we do not have to duplicate character processing. This is already done by the driver line disciplines. This in an advantage of our approach over user-level implementations.

5.2. Functional Overview.

The architecture of the proposed window system is illustrated in Fig.5.1. The window driver is an extended version of a standard driver for a tty multiplexer board.

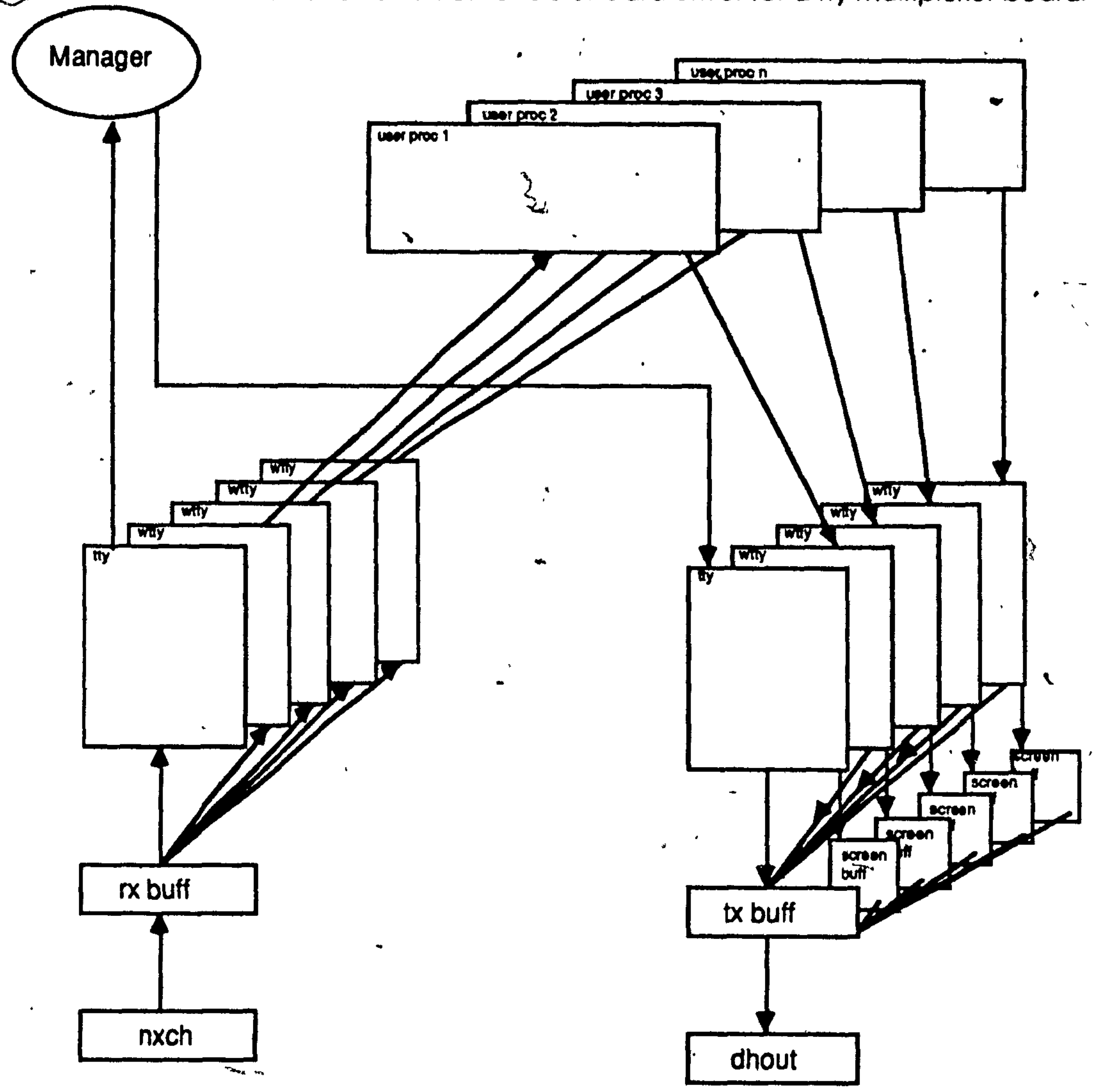

Fig.5.1.
Functional Diagram for the Kernel Window Facility

As in the standard driver, it has a set of data structures for control and I/O to each terminal of the multiplexer. In addition, however, the window driver has a pool of structures for window control. Each instance of the modified structure has all the features of a standard tty structure, with several additional items added for window display and control, such as a window buffer.

Control of the windows from the kernel side is effected by additional code in the lower half of the driver, as well as a modified line discipline to handle the window control.

The driver provides two modes of operation, standard and windowed. In the **standard mode** of operation, the terminal is handled by the standard driver routines and data structures. The **windowed mode** is initiated by the user entering a WIN command in his shell. This command spawns a process, executes a window manager which then in turn initializes a window on the screen. This initialization process is described in detail below but in brief, it consists of the opening of a wtty structure, the spawning of a process, the execution of a shell on that process, and the required linking, both in user mode and in the kernel, of the windowing facilities to the requesting terminal. It can be seen from this architecture that the setting up and control of a window is a task divided between the kernel and a user level process. Once the window is set up, however, I/O is entirely handled by the kernel, in the same way as for a standard terminal.

Each terminal may potentially have a window manager. This manager handles user-level window functions for all the windows associated with that terminal, and has no effect on the other terminals. The pool of wtty structures is, of course, a limited resource. The structures are allocated on a first come, first served basis, and when all are busy, a request for a window simply fails.

The design issues, the decisions and the details of the design are presented in the following.

5.3. The User Interface.

In order to focus on the aspect of multiplexing I/O to the windows and to highlight the problem of window control, we have restricted the scope of this proposal to full screen windows rather than tiled or overlapping windows. The control of window sizing and overlapping would require the addition to the kernel of functions analogous to part of the curses package. In a first view we prefer to separate these from the actual function of window control in the kernel.

The design of our system provides some flexibility for the user by allowing him to work with a variable number of windows within a fixed global limit. Admittedly, on a small system such as the CADMUS, the practical limit on the window pool is rather small, but even contemporary personal computers now have this level of power. As an example for this instance, we have selected 8 as an acceptable limit but with more computing power this limit can be easily changed.

There are three window commands in this approach:

1.Open a window which translates to the creation of a blank screen with a cshell process in it waiting for user commands. The newly created window becomes the **current window.**

2.Close a window which applies only to the current window. In this case the window structure is deallocated and returned to the pool, its process killed and the window preceding the one being closed is displayed. An existing window is made current by typing the control character corresponding to the third operation below.

3.Switch a window which selects next window on the queue, displays the contents of its screen buffer and flags it as "current".

In the proposed design the window facility provides the possibility of running concurrent programs while using the whole screen area, within the limitations of screen I/O. An inactive window will suspend processing on screen I/O unless the user specifies otherwise.

In the design of the user interface, we may consider one of two possible methods to implement the above commands :

1. The user may type three distinct control characters, one for each of the three operations above.

2. The user may type a certain control character followed by a distinct integer value according to the operation desired for the window. This solution presents the following advantages over the first:

- the user always types the same control character but he has to remember the integer value for the desired operation.

-It allows easy expansion of the facility if new window operations are to be provided.

The disadvantage presented by the second method consists of the fact that the modifications needed in the *dhrint()* routine are complex. In the original design one character at a time is taken from the nxch register of the hardware interface, put in the receiver buffer pointed from the tty structure and subsequently processed and sent to the appropriate queues by different routines which all use the same pointer as *dhrint* does in order to access the appropriate dh_tty() entry. While this is adequate for a design where we have one port per terminal, in the multiplexed version the reception of the control character does not supply enough information for the system to access the dh_tty() entry corresponding to the window terminal until the modified *dhrint* routine intercepts the above mentioned character.

We retain the first method, in which the user will type a distinct control character for each designed window operation, as the simplest solution to implement, and the most convenient user interface.

Note.In order to distinguish between a real and a window tty device we will use the terms wtty and tty in order to refer to two structures for a window "device" and a physical terminal device respectively (see appendix 1).

5.4. The Structures Needed in Our Approach.

The basic structure is a modified tty structure (see Appendix 1) which includes information needed for managing the window devices. Instances of this structure may be accessed from both the upper and the lower driver routines.

With this architecture, which defines windows as devices and which uses the terminal management routines of the device driver to handle I/O, the management of windows is similar to the management of the terminals in a conventional system.

Clearly there are many alternatives. The devices could be phantom devices which do not exist in the /dev file, but instead constitute a pool of resources managed solely by the window driver. The implication in this case is the addition of non standard UNIX system calls to access them, which will lead to undesirable complexity and an asymmetry in the handling of I/O through windows.

The choice of defining wtty devices in /dev skirts this problem, and requires only the addition of new driver functions which conform to the standard interface of the UNIX system. This approach has no undesirable side effects, and is in complete harmony with the UNIX concept. It leaves open two alternatives for coordination of the window manager and the lower half multiplexer: manager selection and kernel selection of the device.

Conventionally, a device is selected from the user level by a user process. If the device is a window, however, some means must be provided to indicate to the device driver which physical device is windowed. In the system proposed here, the pool of wtty devices must be managed in such a way that any one of 8 terminal users may access the next available device. If the window manager in the user level of a given terminal has the responsibility of selecting a window device, it must do this by accessing the wtty devices sequentially, and verifying whether they have already been opened (an open inode will be in-core, and will have a non-zero reference count). Great care must be taken to avoid race conditions. The manager must then convey an identifier of this device to the low level multiplexer in answer to the request formulated from the user generated control character. This can be done through an *ioctl()* call (Appendix 2), as it is important to leave standard character I/O with strict UNIX functionality. The implication is an addition of an ioctl command, and a corresponding expansion of the tty structure.

Kernel selection of the device is simpler. In this case, the lower half of the driver manages an array of structures which represent the window devices. The dimension of the array is a system configuration parameter. On reception of a 'new window' control character, the driver selects a free window device, links it to a queue associated with the requesting terminal, and outputs its identifier to the window manager process on the standard terminal device. This can be in the form of a character string configured as a control sequence. The window manager then opens the file of the identified wtty device, spawns a process, links it to the wtty, and invokes a shell. Race conditions are not a problem, as the wtty number is assigned by a single kernel routine. The implementation details of the kernel selection alternative is discussed below. Some details of manager selection are discussed in Appendix 2.

5.5. The Window Driver.

The Window Driver is essentially a modified and expanded tty multiplexer board driver. While the data structures of a window present some differences from those of a terminal, they are sufficiently alike to be managed by the same set of functions, expanded to handle the modifications. The standard driver structures and functions were described in chapter 4. Here we present only the modifications.

Note that both device types can be managed by a single driver, due to the similarity of the functions. The extra operations can be handled either by new entry points or conditional selection of the specific functions.

Appendix 1 illustrates the data structures of the window. The major additional element added is the window buffer, which retains an image of the contents being displayed in a window. When the window is active, these contents are identical to the window on the terminal screen. When inactive, they store the record of activity which is suspended once the buffer is full, or which continues if specified by the user.

In addition to the wtty structure (Appendix 1) there is a global array of pointers to linked lists of wttys (Fig. 5.2), having one entry for each terminal. The pointers to the receiver and transmitter buffers in both wtty and tty structures can be initialized to point to a single receiver and transmitter buffer respectively. This way the lower half routines *dhrint()* and *dhxint()*do not have to be changed. The modifications needed can be implemented in the line discipline routines.

```
extern struct wqueue {
  int head;
  int tail;
  int curr;
  struct wtty
fifo[NWTTY]
```

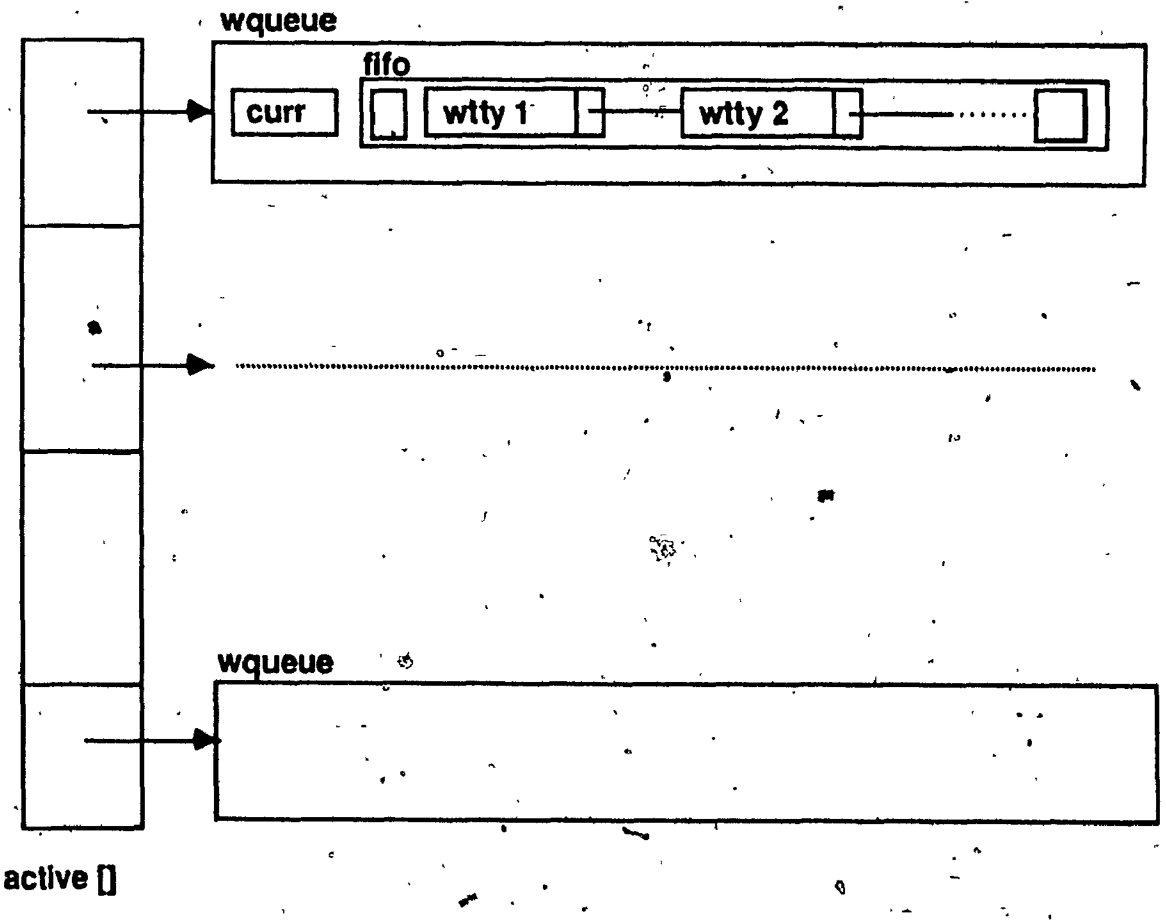

Fig 5.2
Window Driver Data Structures

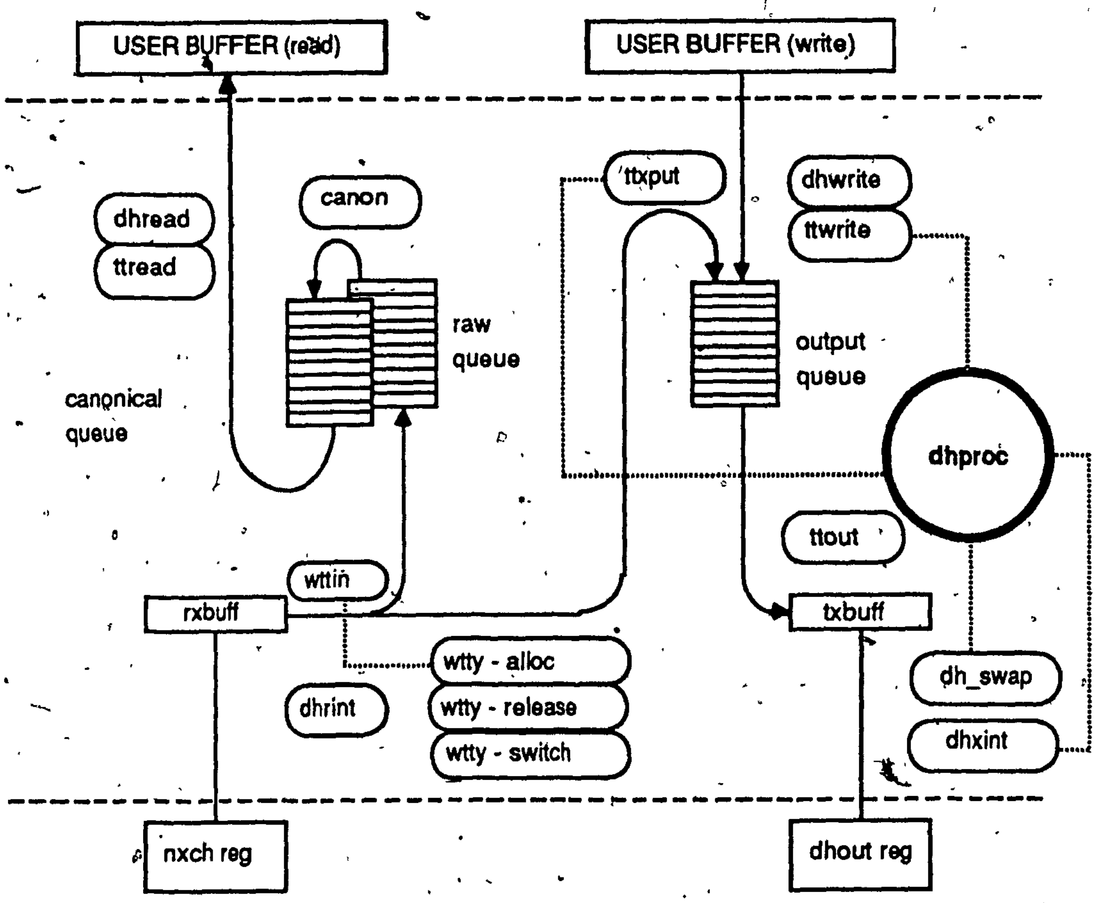

Fig 5.3
Modified tty Control Routines

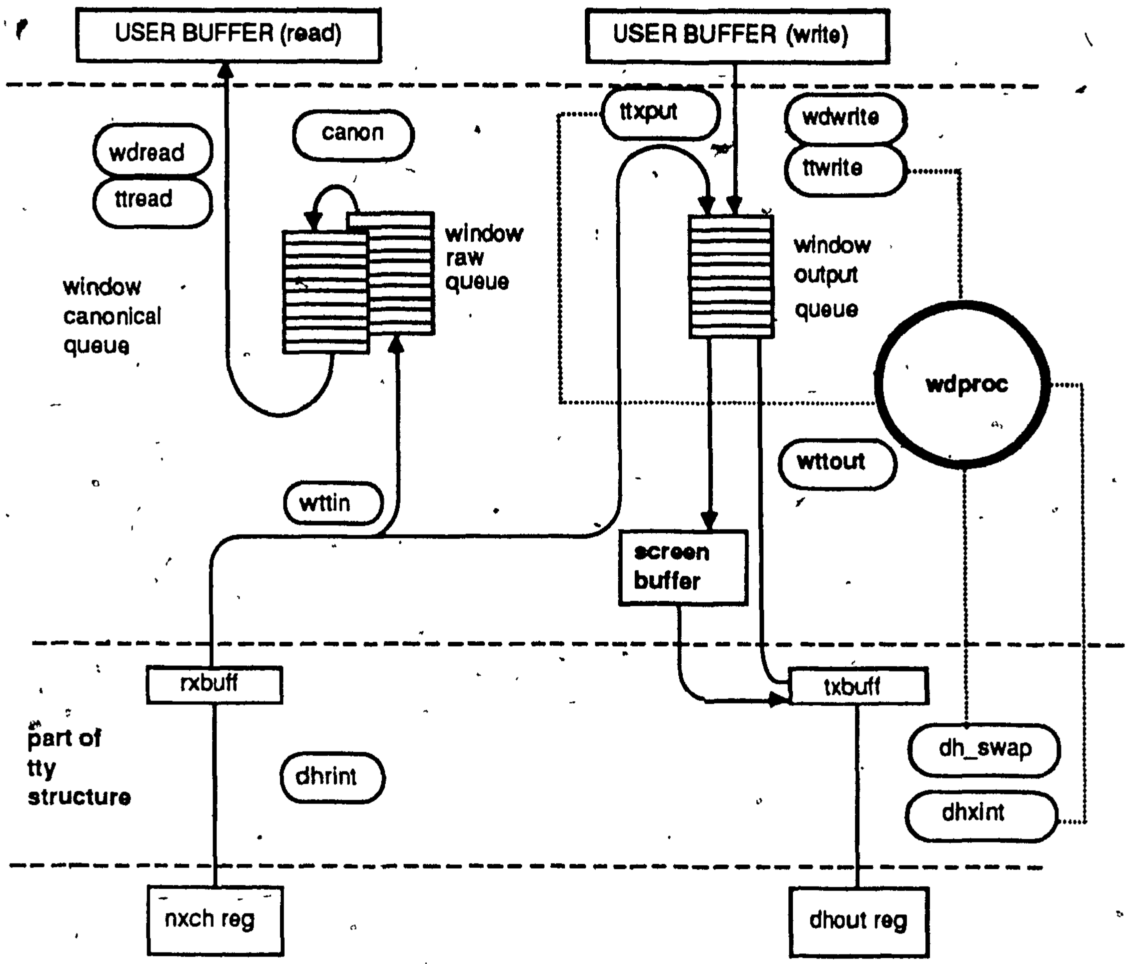

Fig 5.4

Window Structure with Modified tty Control Routines

For the input, *ttin* will contain additional code, thus becoming *wttin*. Before starting the character processing according to the setting of the input flags, *wttin* will check to see if the character corresponds to one of the window control characters.

If the character is an open command for a new window, *wttin* calls a new function, *wtty_select*. This function does the following:

-it scans the entries in the common pool of wttys

-it picks up the first one which is not busy by checking the field "alloc" and marks it busy

-it enters the tty number which has acquired it in the new field "ttyline"

-it links the wtty structure into the linked list belonging to the terminal entry in the array active

-it updates the field "curr" of the same entry to contain the wtty integer number

-it places a byte coding the "open" operation and the integer id of the acquired wtty structure on the raw queue of the terminal in order to be sent to the Window Manager.

If the character is a command to close a window, *wttin* calls a new function *wtty_release*. This function does the following:

-it issues a kill signal to the process group in control of the window.

-it deallocates the current wtty structure for the terminal entry and it marks its field "alloc" as free

-it updates the field "curr" with the integer number of the wtty preceding the one being closed

-it calls the routine *wtty_flush* which flushes the buffer of the new current window on the screen

Note that there is no direct communication with the Window Manager. It is necessary for the process group using the window to be killed before it has a chance to attempt I/O on the unlinked window structure. The only way to achieve this is to kill it from the driver itself. The Window Manager process will receive a signal informing it of the death of a child, and will take the usual actions in this case. It does not attempt to respawn the child.

If the character is to switch a window, *wttin* calls a new function *wtty_switch* which:

-selects the next wtty structure on the linked list for the terminal

-updates the field "curr" accordingly

-flushes the buffer of the current window as above

Note that the user level window manager is not involved in a switch between windows.

If the character typed is not a window control character, *wttin* sets the pointer tp to the current wtty structure for the terminal entry by accessing the array "active", puts the character on the raw queue of that wtty structure after having processed it in the same way as for the terminal.

For the transmission part, *dhproc* requires changes only in the statements which calculate the physical line number. This is to be retrieved from the field "ttyline" mentioned above. The routine *ttout* which is invoked by *dhproc* becomes *wttout.* Before putting the character into the transmitter buffer, *wttout* needs to update the output screen buffer when echoing the input or when the window receives output. Additionally, it invokes a scrolling function for the screen buffer. The screen buffer is to be maintained as a circular buffer with pointers to the top line, bottom line and the cursor.

As for the upper half window driver routines, they are to be found in an extra entry in the cdevswitch table. This is because the wtty files to be opened will be found in /dev with a distinct major number from that of the tty files.

The routine *wdopen* will check whether the minor number is valid, then it sets a pointer tp to the corresponding entry of the wtty array, marks the field t_state as being open and calls *ttopen* to allocate the receiver buffer.

The routines *wdclose, wdread, wdwrite* differ from their analogous tty routines only with respect to the tp pointer which is set to select the appropriate entry of the wtty array. Then they call *dhclose, dhread* and *dhwrite* respectively. The fact that the modifications are not dramatic is normal, because these routines refer to the management of the queues which is the same for both the terminal and the windows.

5.6. The Window Manager.

The Window Manager is a program which is basically responsible for opening the window files and spawning the processes for them. It does the following:

1. It issues a blocking read for two bytes corresponding to the designed control sequence. The proposal is that the first byte codes the operation and the second one the window id
2. If the operation is "open", it issues an open call on the /dev/wtty device corresponding to the window id.
3. It opens that file for read and write and it redirects the fd to stdin, stdout and stderr
4. It sets the wtty parameters in the termio structure, speed, new line discipline, etc. as for a real tty
5. It spawns a process, and executes a shell from it.

Other operation codes are not used in the present proposal, but are available for future extensions of the system.

5.7. Overhead Considerations.

As a general rule, it is preferable to add features to Unix on the user level rather than in the kernel. It is therefore appropriate to provide a clear rationale for any modifications made to the kernel, as proposed in this design. It should be noted, of course, that all the proposed modifications are restricted to a driver, which follows the standard Unix calling conventions with the exception of one additional ioctl call. In this sense, it can be argued that the main body of the Unix kernel is not being modified. The principal reason for proposing this design option is, however, the reduced overhead promised by this implementation.

A comparison of the overhead between a driver implementation and a user level implementation can be estimated on the basis of two factors: system calls and data transfers. System calls involve heavy overhead. Entry and exit from the kernel requires invocation of trap processing together with the library routines and functions which support this operation. It is also quite probable that a context switch may also occur if the system is not rather lightly loaded. Data transfer operations refer to the copying of data from one buffer to another. Already in a standard terminal driver, this operation occurs up to four times: interface to buffer, buffer to clist, clist to canonical clist, and canonical clist to user buffer. This structure has evolved, with its high overhead, from the design goal of presenting an absolutely standard interface to user level programs. Any windowing system will add to this overhead, as seen from the specifications below.

To estimate the overhead of a user level system, we have taken the example of WOW, the system designed and implemented (in pre-prototype form) in Concordia. The reason for using this example is its relative simplicity and the accessibility of its code and design philosophy. Referring back to figure 3.3, it can be seen that the architecture of this system consists of a control-multiplexer process connected to the window processes by virtual tty pipe pairs.

The operations involved in the case of a <u>read</u> from a window consist therefore of a

1.a read on the vtty from the window process
2.a write on the vtty by the multiplexer-manager
3.a read on the driver by the multiplexer-manager
4.any other operations required to control and synchronize the above.

The actual system calls are as follows:

-read(STDIN, ch, 1) in the routine getkbd()
-write(table(where)(src_input), pkt, strlen(pkt)) in the routine write_to_term
-read(fd, &ch, 1) in the routine get_ch, where fd represents the file descriptor of the outpipe
-reading user commands from the cshell which has been previously spawned for the respective window on its creation.

The actual data transfers for input are as follows:

-*(v -> ichar++) = c, which copies the user typed character in the input buffer of the VTERM structure. The pointer ichar points to the current position of pkt(PACKETSIZE) with PACKETSIZE defined as 255.
-win -> _y(y)(x++) = c, in the routine waddch which echoes the character to the current window
-while(*(packet++) = *(p++)) which copies the packet from screen buffer pointed to by p to the multiplexor buffer called packet().

A write operation is essentially the mirror image of the above.

The overhead involved in the establishment of a window consists of

1.a read of the control characters requesting a window
2.creation of vtty
3.creation of a child process
4.linkage of the vtty to the child
5.invokation of the cshell
6.establishment of this child process as the current window.

The system calls involved are as follows:

-read(STDIN, ch, 1) in the routine getkbd
-pipe(p.inpipe) to create an input pipe for the child
-pipe(p.outpipe) to create an output pipe for the child
-process_id = fork() to create a child process
-ioctl(p.outpipe(WR), PIOCVTTY, (srtuct sgttyb*) &p) where p is a pointer to input and output pipe structure. This links the pipes into a vtty.

-12 close() and dup() system calls for appropriate redirection of the file descriptors for the pipes in both the child and the parent
-execlp("/bin/csh","-csh","-i","-s",NULL) which spawns a cshell for the window. All these system calls are done from the routine enter_new_user.
-setpgrp() to ensure appropiate distribution of the interrupts.

For the proposed driver implementation we refer to figures 5.3 and 5.4. In this case, the user level software is minimal, and is only required to create and terminate the processes on which the windows are invoked. As a result, the read and write operations on the user level have an overhead which is identical to a regular system. The number of system calls is therefore identical. On the other hand, the increased complexity of the driver implies some overhead in data transfer. This is principally the maintenance of the screen buffer, which is done inside the driver. This involves an extra copy of both incoming and outgoing data, and the adjustment of pointers to identify the screen top and bottom and the cursor position.

The actual additional data transfers are as follows:
-for the window control, copying the tty number acquiring a new window into the "ttyline" field of the wtty structure, copying the acquired wtty integer number into the field "curr", screen buffer flushing of a current window.
-for the I/O to the current window, copying of each character from the window output queue to the screen buffer and adjustment of the buffer pointers.

Compared with a standard driver, there are a number of additional tests which must be carried out on every character. These consist of testing for all additional control characters to distinguish between a window command and all other characters, text and control. In a user implementation, these operations are done in the multiplexor or by virtual tty calls. As the driver is normally set to raw mode, this testing is not duplicated. It may therefore be surmised that there is no significant difference in character testing overhead between the two implementations, although one is done on the user level and the other in the kernel.

A summary of the I/O operations in the two methods is given below. It is clear that the driver implementation is far more efficient from the point of view of operations on a single window.

I/O

	Window Driver	W O W
System calls	direct *read* by user process	-*read* on the input side of the vtty from the window process, -*write* on the input side of the vtty by the multiplexer-manager -*read* from stdin by the multiplexer-manager -system calls to support multiplexing operations: non-blocking *reads* on all vtty's, *kill*(0) to see if process still alive.
Data Transfers	copying in the kernel of the incoming and outgoing data to the screen buffer at the current cursor position	-from the read buffer to the VTERM buffer -from the read buffer to the screen buffer -from the VTERM buffer to the packet buffer -from the packet buffer to the virtual terminal pipes

5.8. Extension to Multiple Windows.

The present proposal has been restricted thus far to single full screen windows. This restriction has permitted us to highlight the essential differences between multiplexing in a driver in the kernel level and multiplexing on the user level. This does not imply, however, that multiple overlapping window implementation is impractical in the window driver. In fact, a multiple window display is principally a problem of cursor control, and the management of display rewriting on switching or resizing of windows.

We shall discuss the essential functions required in the following, without entering into detail, as most of these operations are implemented in an analogous form on the user level by the <u>curses</u> package.

Communication with the current window presents no problem, as the cursor position can be controlled by appropriate cursor positioning commands. The size of the window can be stored in parameters associated with the screen buffer of each window. These can be used to generate the appropriate commands to keep text output within the window. Scrolling within a window presents a problem, as its control in software implies rewriting the entire window contents with each line move. The displaying would be limited practically to a non scrolled output.

Display rewriting can be managed by inspection of the queue of displays. Each screen affected may be rewritten from lowest to the highest, or current window.

As the techniques for this type of window management are well known and currently used, it is not appropriate here to describe them in detail, but simply to indicate that there is no problem in their implementation in the driver apart from the volume of additional code added to the kernel. The level of control described above, however, constitutes a small subset of the curses package, and would not represent a disproportionate addition.

Chapter 6

Conclusions

This report has presented a design proposal for a window facility based on the extension of the multiplexing capability of the Q/DH multiplexer board for UNIX System V on a Cadmus computer to a second level: sets of windows for each terminal, allocated from a common pool. Although some of the driver routines, namely *dhread, dhwrite* and *dhproc* and the line discipline routines *ttin* and *ttout* are modified, the UNIX system interface remains the same. No additional system calls are required, and all modified ones obey the standard calling conventions. Although this design is based on modifications in the kernel, a window manager is still required, but its code is minimal and it handles only window initialization, process spawning and the usual termination operations on "death of a child process". No character processing is required at this level because it is already done by the line disciplines.

The modifications mentioned in Chapter 5 refer to new entries in the cdevswitch and the lineswitch tables, some modified driver and line discipline routines, and a user level program in /bin.

With respect to the overhead, our proposal takes advantage of the natural level of multiplexing in the driver, thus avoiding the many system calls needed in a user implementation of a window facility like WOW, particularly for the window control, as seen from the summary on page 54. As a result, the overhead in our proposal is reduced compared to a user level implementation.

While we did not consider overlapping and outlining of windows in detail, the techniques are well known, and would not add any original aspects to the design. It is clear, however, that overlapping windows in our approach would increase the driver size and overhead. Actually as most terminals are not designed for overlapped windows, a full screen window facility with icons which indicate the presence of other windows is far more practical. The icons can be labelled with integers instead of graphic symbols. Even so, the functionality does not compare with the power of windows on personal computers which now cost hardly more than a terminal. Notwithstanding these aspects, the increased power provided to a terminal with relatively little overhead is an attractive option.

The significance of this work also relates to other applications which use multiplexed I/O. The same architecture is applicable to communications on a shared medium such as a Local Area Network where several users have access to character I/O through the same port. It is also applicable for character I/O to intelligent terminals as in BLIT or to PC workstations, which can use their own bit mapped drivers to display the multiplexed data as windows.

6.1. Future Development

It seems unlikely that the future of the UNIX user interface will be based on windowed ASCII terminals. Prices of personal computers have dropped so rapidly in the past few years, that it is already hard to justify the purchase of a terminal rather than an intelligent device with its own bit mapped display. The software architecture presented in this report is, however even more appropriate to communication with such devices, as multiplexed output on a serial port can be demultiplexed far more easily on a PC than on an ASCII terminal. Some modifications in the design presented here would be required, but these would lead to simplification rather than increased complexity. The local memory in the PC removes the need for a screen buffer in the driver, and the transfer of other functions could raise the speed potential of I/O.

Many interesting developments in the area of cooperation between a UNIX system and a PC user interface are possible. The ideas developed in this work could form the basis for simple and efficient designs.

REFERENCES.

(And86) Anderson R., "UNIX through windows", COMUNIX 1986, London, England, 3-5 June 1986, pp.115-121.

(Bac86) Bach J.M., " The Design of the UNIX Operating System " , Prentice - Hall Inc.,Englewood Cliffs NJ 1986.

(Bre84) Bresnahan, J.B., Barnard D.T., and Macleod J.A., "WSH - A New Command Interpreter for UNIX", Software Practice and Experience, Vol. 14(12), December 1984, pp.1197-1205.

(Bou78) Bourne S.R., " The UNIX Shell " Bell System Technical Journal Vol.57 No.6 Part 2 July - August 1978.

(Bro86) Brown D. " The Q/DH Device Driver " A Concordia Project June 1986.

(Gam84) Gammill R., Prithvi R., "VT - A Virtual Terminal Window Package for Unix", ACM SIGSMALL, May 1984, pp.21-30.

(Ker 84) Kernighan B.W. and Pike R., " The UNIX Programming Environment " Prentice-Hall Englewood Cliffs NJ 1984.

(Hol86) Holcomb R., Tharp A.L., "The Effect on Man-machine Interfaces (or opening doors with windows)", ACM SIGDOC Asterik (USA), Vol. 12, No. 3, October 1986, pp.9-20.

(Jac84) Jacob R.J.K., "User - level Window Managers for UNIX", Proc. UniForum International Conference on Unix", Washington DC, January 1984.

(Ker78) Kernighan B.W. and Ritchie D.M., "The C Programming Language " Prentice - Hall Englewood Cliffs NJ 1978.

(Lac87) Lacroix R. " WOW " A Concordia Project, May 1987.

(Lan79) Lantz K.A., Rashid R.F., "Virtual Terminal Management in a Multiple Process Environment", Proceedings of the Seventh Symposium on Operating Systems Principles, December 1979, pp.86-97.

(Mey81) Meyrowitz N., Moser M., "BRUWIN": An Adaptable Design Strategy for Window Manager / Virtual Terminal Systems", ACM SIGOPS Conference, Vol 15, No 5, December 1981, pp180-189.

(Pik84) Pike R., " The Blit : A Multiplexed Graphics Terminal " , AT&T Bell Laboratories Technical Journal Vol.63 No.8 Part 2 October 1984 pp1607-1632.

(Pik85) Pike R. and Locanthi B., " Hardware / Software Trade-offs for Bitmap Graphics on the Blit " , Software Practice and Experience Vol.15(2) pp131-151 Feb.1985.

(Qua85) Quarterman J.S., Silberschatz A., Peterson J.L., "4.2BSD and 4.3BSD as Examples of the Unix System", Computing Surveys (USA), vol. 17, No.4, December 1985, pp.379-418.

(Rit78) Ritchie D.M., " A Retrospective " The Bell System Technical Journal, Vol.57 No.6 Part 2 July-August 1978, pp.1947-1970.

(Rit81) Ritchie D.M. and Thompson K. " Some Further Aspects of the UNIX Time-Sharing System " , Mini-Micro Software Vol.6 No.3 1981, pp. 9-12.

(Rit84a) Ritchie D.M.," The Evolution of the UNIX Time-Sharing System ", AT&T Bell Laboratories Technical Journal Vol.63 No.8 Part 2 October 1984 pp1577-1594.

(Rit84b) Ritchie D.M., " A Stream Input Output System " , AT&T Bell Laboratories Techical Journal Vol.63 No.8 Part 2 October 1984 pp1897-1910.

(Tat82) Tate A., "A Window Manager for the UCSD p-System", SIGSMALL Newsletter, 8, (1), march 1982, pp.14-21.

(Tei81) Teitelman W. and Masinter L., "The Interlisp Programming Environment", Computer 14 No. 4, april 1981, pp.25-33.

(Tes81) Tesler Larry, "The Smalltalk Environment", Byte Vol. 6, No. 8, August 1981, pp.90-147.

(Thi87) Thimbley H., " The Design of a Terminal Independent Package ", Software-Practice and Experience, Vol.17(5) pp351-367 May 1987.

(Unia) UNIX System V User Reference Manual.

(Unib) UNIX BSD 4.3. User Reference Manual.

(Unic) UNIX System V Driver Source Code.

(Uni86) UNIX BSD 4.3. Window Facility Source Code 1986.

(Uni87a) UNIX System V STREAMS Primer, AT&T, Prentice-Hall Inc, 1987.

(Uni87b) UNIX System V Streams Programmer's Guide, AT&T, Prentice-Hall Inc, 1987.

(War79) Warren S.K. and Abbe D., "Rosetta smalltalk: a conversational, extensible microcomputer language", SIGSMALL Newsletter Vol. 5, No.2, 1979, pp.36-45.

(Wei85) Weiser M., "CWSH: The Windowing Shell of the Maryland Window System", Software-Practice and Experience, Vol. 15(5), May 1985, pp.515-522.

Appendix 1

A wtty structure is needed for each window terminal. This structure is given below. It represents an extension of the tty structure. The new fields which do not appear in the tty structure are in bold characters.

```
#define NCC 13
#define BUFSIZE 256

struct wtty {
        struct clist t_rawq;              /*raw input queue*/
        struct clist t_canq;              /*canonical queue*/
        struct clist t_outq;              /*output queue*/
        struct buff{                      /*screen buffer*/
                int nr;                   /*no. of rows*/
                int nc;                   /*no. of columns*/
                int t,b;                  /*top, bottom*/
                int l,r;                  /*left, right*/
                char y(BUFFERSIZE);       /*buffer*/
                int *firstch;             /*pointer to the first character*/
                int *lastch;              /*pointer to the last character*/
                bool scroll;              /*flag for scrolling full screen window*/
                struct cursor{            /*cursor control */
                        int r;            /*row*/
                        int c;            /*column*/
                } scrbuff;
        struct ccblock t_tbuf;            /*tx control block*/
        struct ccblock t_rbuf;            /*rx control block*/
        int (* t_proc)();                 /*routine for window functions*/
        ushort t_iflag;                   /*input modes*/
        ushort t_oflag;                   /*output modes*/
        ushort t_cflag;                   /*control modes*/
        ushort t_lflag;                   /*line discipline modes*/
        int alloc;                        /*wtty allocated*/
        int ttyline;                   /*to which line the window is tied to*/
        short t_state;                 /*internal state*/
        short t_pgrp;                  /*process group name*/
        char t_line;                   /*line discipline*/
        char t_delct;                  /*delimiter count*/
        char t_term;                   /*terminal type*/
        char t_tmflag;                 /*terminal flags*/
        char t_col;                    /*current column*/
        char t_row;                    /*current row*/
        char t_vrow;                   /*variable row*/
        char t_lrow;                   /*last physical row*/
        char t_hqcnt;                  /*no. high queue packets on t_outq*/
        char t_dstat;                  /*used by terminal handlers and LD*/
        unsigned char t_cc(NCC);       /*settable control chars*/
};
```

Appendix 2

In the Manager Alternative, in step 1, the Window Manager issues a blocking read for only one character. Then the steps 2, 3, 4, and 5 are the same as for the Driver Alternative. After step 5, it executes a modified ioctl call as described below in order to communicate the id of the acquired wtty file to the global structure "active". The effect of this ioctl call is to update the field "curr" of the entry corresponding to the terminal with the window id and to call wtty_flush to flush the screen buffer, in this case a blank screen.

The Modification of the *ioctl* System Call.

ioctl() is a system call controlling a device. It actually transfers information from the user buffer to the driver or vice-versa. The standard function is as follows.

After extracting the major and minor numbers as done from any system call, *ioctl()* invokes *dh_ioctl()* upper half driver routine. The *dh_ioctl()* routine in its turn calls a function *ttiocom()* which carries out one of the device dependent actions specified in the argument "cmd" of the call. A process normally uses an *ioctl()* system call in the following way:

-it issues an *ioctl()* with a particular command in order to get the terminal parameters by accessing the appropriate tty driver structure fields

-it sets the terminal parameters to the desired values

-it issues another *ioctl()* with a appropriately chosen command which restores the parameters in the corresponding fields of the tty driver structure. The syntax of the *ioctl()* call is given below:

ioctl(fildes, cmd, arg), where "arg" may be declared in two ways:

1. **As a pointer to a termio structure** defined as

```
struct termio {
        unsigned short c_iflag        /*input  modes*/
        unsigned short c_oflag        /*output  modes*/
        unsigned short c_cflag        /*control  modes*/
        unsigned short c_lflag        /*local  modes*/
        char c_line                   /*line  discipline*/
        char c_cc(NCC)                /*control  characters*/
```

};

An example of a command using this form is TCGETA which gets from the system the parameters associated with the terminal and stores them in the termio structure referenced by arg.

2. **As an integer**. An example of command using this form is TCSBRK which waits for the output to drain. If "arg" is equal to 0, it sends a break signal to the terminal. For our purpose, in order to transmit an integer value to the lower half *dhrint()* routine we need to use the latter of the two forms.

The proposed modification involves the following:

-a new command to be coded in termio.h, let us call it MPXOPT (for multiplexed option)

-a few lines of code have to be added in *ttiocom()* function to process this new command, as outlined below:

```
----------
register struct tty *tp;
char * arg;
boolean flag = FALSE ; integer line ;
----------
case MPXOPT
if(hiword(arg) is nonnegative) {
line = tp - dh_tty  /*calculate the port number */
active(line) -> curr = hiword(arg)
wtty_flush(tp) ;  /*this flushes the wtty buffer on the screen  */
return(flag = TRUE)
}
```

where **active(line)** is the kernel 8 by 1 integer array mentioned previously which is accessible by both upper and lower half driver routines

-when returning from ttiocom, *dh_ioctl()* invokes another routine *dhparam()*. This routine sets the line parameter register for the terminal based on the control modes of the corresponding tty structure. As our modification does not refer to any hardware interface we may insert an if(not flag) statement before calling *dhparam()*.

Appendix 3

The INIT-LOGIN-GETTY-CSHELL Process Sequence in Controlling a Set of Terminals in UNIX V.

In this appendix, we examine the sequence which initializes user processes on the terminals of a system configured for multiple users. Some aspects of the design of the Window Manager are related to the functions of this program, in particular the initialization sequence of the Manager.

The initialization of control processes for multiple terminals is handled by a set of programs sequenced by init. The set up of processes in a multiple window system can follow an analogous procedure, with appropriate changes, of course, to the levels on which the programs will be executed, and details in their execution.

Init ((Unia), (Bac86)) is invoked in UNIX V as the last step in the boot procedure. In multi-user operation, init continously reads the file /etc/inittab and forks (i.e. spawns a child) as many times as the number of physical terminals in this file such that a process is created for each such terminal. This process becomes the control process meaning that it will control all the processes initiated by the user from that terminal after he has succesfully logged in. Typically the control process is the login shell.

Once in the shell, the user may execute commands. Each command usually represents a program which runs from a child spawned by the shell.

When typing input data from the keyboard, the user may press certain keys like "delete" or "break" by mistake or on purpose. These may be interpreted by the kernel line disciplines as signals which are to be sent to user processes. This implies a uniform treatment of all processes associated with a terminal, which has been made possible in UNIX by including them into the same group distinguished by a group process id, with the parent being the process group leader.

A process group is established by issuing a *setpgrp()* system call which will initialize the p_pgrp field of the kernel process structure to the pid of the parent. All processes spawned from the process group leader will inherit its group pid unless they are specifically released following a *setpgrp()* system call (one such call for each process to be released!). This will make the process executing the call a group leader for its future descendants, a feature which is used in the initialization of windows as mentioned in section 3.1.

If the process group leader opens a terminal (i.e. a /dev/tty* file corresponding to a terminal), the line discipline routines will associate it with the terminal if not already previously done and thus it will become a control process for the terminal hereafter called the control terminal.

The /etc/inittab file contains a table which is illustrated below.

Getty is a program which in a multiuser environment initializes individual terminal lines where users might log in. It can be found in /etc directory.

Format : identifier, state, action, process specification
Fields separated by colons
Comment at end of line preceded by '#'

```
1 is   : s   : initdefault :
- --   : -   : --------- :
- --   :   - : --------- :
7 co  : 34 : respawn : getty Sys'con console adm3a #
- --      : - - : ---------- : ------------------------ #
9 t11  : 34 : repsawn : getty /dev/tty11 exta vt100 #
10 t12 : 34 : respawn : getty /dev/tty12 exta vt100 #
--'---' : -- : ----------- : ------------------------ #
id   rstate  action          process
```

Fig.A1. A Sample Inittab File.

The significance of each field is given below:

id - identifies the entry

rstate - defines the run-level (one or multiple) which is viewed as a software system configuration allowing only a selected group of processes spawned by init to exist at any given time; all processes with run levels differing from the target run level are forced to terminate after a certain grace period.

action - indicates how to treat processes in the process field .

process - specifies a program name and its respective parameters which will be passed as a command to a forked shell after being prefixed by "exec" character string

To each physical terminal in the system corresponds an inittab entry for which:

1) The action field typically contains the character string"respawn" which when interpreted by init will have the following effect. If the process field for that entry does not exist, init will create it, after which it will continue scanning the inittab file without waiting for its termination. When such a process dies, it will recreate it. Else, if the process currently exists init will do nothing, but continue scanning the inittab file.

2) The process field contains the string "getty" followed by a number of arguments as seen in Fig.A.1. above.

Getty must have at least one argument, the tty line. If it is not provided, it will return a message error "no terminal line specified" and it will exit. Init will respawn the process as mentioned earlier.

After setting character pointers to each argument, getty does the following:

- changes the directory to /dev
- attempts to open the file specifying the particular terminal line for read and write as described previously in section 4.1.; if unsuccessful, getty exits and will try again when respawned by init, else
- returns the file descriptor for the opened line redirected as stdin, stdout, stderr
- appropriately initializes terminal parameters (via several ioctl calls) to either the specified arguments or the default ones
- writes a login message for the respective line requesting a user name and waits for an input
- while reading the user name one character at a time, getty attempts to adapt the system to the speed and type of terminal in use
- if no name was supplied within a certain delay or a bad name was given it will try again; else (if successful),
- executes the login program with the user supplied name as argument

The login process writes a request for a password. When it receives the password, it checks it against the password file and if correct, it executes a "csh" program which will prompt the user; if the operation is not succesful, depending on the Unix version, it might disconnect the user or allow him to retry.

The csh is the last process in the sequence init -getty - login - shell. In this chain, with each successful process execution, the previous process gets overlaid, except for init which executes getty from a child. In fact the connection to the driver is achieved by getty with a successful open for the line and it continues to exist as long as the user executes commands from his login shell.