SOMMAIRE

DEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTES
DEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTES
DEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTES

Developer's Handbook for the Apple II MouseText Tool Kit

DEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTES
DEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTES
DEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTES

Ø3Ø-Ø891-2

NOTICE

Apple Computer, Inc. reserves the right to make improvements in the product described in this manual at any time and without notice.

DISCLAIMER OF ALL WARRANTIES AND LIABILITY

APPLE COMPUTER, INC. MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL OR WITH RESPECT TO THE SOFTWARE DESCRIBED IN THIS MANUAL, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. APPLE COMPUTER, INC. SOFTWARE IS SOLD OR LICENSED "AS IS". THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE IS WITH THE BUYER. SHOULD THE PROGRAMS PROVE DEFECTIVE FOLLOWING THEIR PURCHASE, THE BUYER (AND NOT APPLE COMPUTER, INC., ITS DISTRIBUTOR OR ITS RETAILER) ASSUMES THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION AND ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES. IN NO EVENT WILL APPLE COMPUTER, INC. BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN THE SOFTWARE, EVEN IF APPLE COMPUTER, INC. HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES OR LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010

Contents

Table of Contents

List of Figures and Tables

Foreword

This is the developer's handbook for Version 2.1 of the MouseText Tool Kit for the Apple II. The main purpose of the handbook is to tell you how to use the Tool Kit routines in your application programs. In addition, the handbook includes appendixes that contain information about the mouse itself and about the hardware and software that make it work.

Version 2.1 of the Apple II MouseText Tool Kit provides support for mouse-operated menus and windows using the text display. It uses the Mouse Text icon characters available on the Apple IIc. The icon characters are available on the Apple IIe only with an updated character ROM. Another tool kit, The Mouse Graphics Tool Kit, will support the double high-resolution graphics displays on the Apple IIc and the Apple IIe.

Note to Users: This is not the owner's manual for AppleMouse II. That manual, The AppleMouse II User's Manual, tells how to install the mouse on the Apple II and describes the demonstration program that comes with the mouse. This handbook tells you how to use the MouseText Tool Kit routines in programs that you write yourself.

Chapter 1 outlines the features of the MouseText Tool Kit and tells you what the Tool Kit routines will do for your application programs.

Chapter 2 gives complete specifications for the MouseText Tool Kit commands.

Chapters 3, 4, and 5 describe how to use the MouseText Tool Kit with application programs written in each of three different languages: Chapter 3 describes the command calls in machine language, Chapter 4 describes the command procedures in Pascal, and Chapter 5 describes the

ampersand commands used in Applesoft.

The appendixes provide additional information about using the AppleMouse II. Appendix A describes the interface card that supports the operation of the mouse hardware. Appendix B describes the interface to the mouse firmware (the level of communication and control between the hardware and the Tool Kit routines). Appendix C tells you how to install the Pascal Attach Driver that adds mouse communications to the Pascal BIOS. Appendix D contains programming examples using the MouseText Tool Kit. Appendix E describes the special Mouse Text characters. Appendix F is a combined list of the error codes returned by the Tool Kit commands.

Chapter 1

Introduction: The MouseText Tool Kit

The Apple II MouseText Tool Kit is a set of software routines that you can use to implement mouse-controlled menus and text windows for your application programs. This version of the Tool Kit provides commands for displaying and controlling pull-down menus, including

- cursor selection and display
- menu bar displays
- menu item selection.

This version of the MouseText Tool Kit also has window-handling commands used in desk-top displays for handling folders and the like. These commands perform functions such as

- window selection and display
- window dragging and size changing
- writing text in windows.

The Tool Kit also provides support for programs to perform functions like scrolling windows through documents.

Special Characters: The character generator in the Apple IIc includes special characters called MouseText that can be used for cursor displays. A new character-generator ROM will be available to provide the MouseText characters on the Apple IIe. The MouseText characters are described in Appendix E.

The Tool Kit supports 8Ø-column displays on the Apple IIc and, via the Apple 8Ø-column text card or equivalent cards, on the Apple IIe.

Features Supported by the Tool Kit

The commands in the MouseText Tool Kit enable your program to support several kinds of features:

- The Cursor
- Events
- Menus
- Windows
- Control Regions consisting of Scroll Bars with Thumbs

The sections that follow outline these features and mention some of the individual commands your program calls. Tables 1-1 and 1-2 list all of the commands; they are described individually in Chapter 2.

The Cursor

The cursor is the character that moves on the display as the user moves the mouse. Cursor commands enable the program to select the character displayed as the cursor and to turn the cursor on and off. The Tool Kit uses the mouse to control the position of the cursor. There is also a means of controling the cursor and the Tool Kit functions by pressing keys: see the section "Mouse Emulation" later in this chapter.

Events

The Tool Kit deals with four kinds of events: mouse events, keyboard events, update events, and application events (optional with the application program). Mouse events are button pressed (down), button released (up), and moving the mouse with the button held down (drag). Mouse motion with the button up is not an event, but the program can obtain the most recent mouse position even if no event has occurred. Keyboard events are keypresses and are optional; that is, the program specifies that it handles keypresses itself or that the Tool Kit deals with them.

Update events are a special case, provided for those applications with windows that can't be refreshed automatically. Please see the description in Chapter 2 at the GetEvent command.

Precedence of Events: If the mouse button is down, the Tool Kit ignores keypresses.

The Tool Kit's event-handling commands maintain a queue of events.

The program detects mouse events by calling GetEvent. With the Tool Kit running in Passive Mode, GetEvent automatically issues an internal call to CheckEvents. The CheckEvents command posts mouse events and keypress events in the queue and updates the mouse position. If the event queue is empty, the GetEvent command simply returns the most recent mouse position.

In Interrupt Mode, the Tool Kit's interrrupt handler calls CheckEvents 6Ø times per second, synchronized with the display vertical blanking (VBL). In Passive Mode, the application program must call CheckEvents or GetEvent often enough to obtain smooth cursor motion. Also, the application program can put its own events into the queue by calling the PostEvent command.

CheckEvents is the only command that reads the mouse; if it is never called, either directly, indirectly by GetEvent, or (in Interrupt Mode) by the Tool Kit itself,the cursor will never move.

If the event queue fills up, the Tool Kit ignores new events until there is room for them in the queue. To empty the queue, the program calls the FlushEvents command.

Note: Frequent calls to CheckEvents also provide a type-ahead feature by posting keyboard events in the queue until the program can process them.

Menus

The Tool Kit's menu management commands enable programs to provide pull-down menus. The visible components of a menu are

- a menu bar at the top of the display, showing the menu titles

- the menu items that appear, one to a line, when a menu pops down.

When the user moves the cursor onto a title in the menu bar and presses the button on the Mouse, the application program calls the

MenuSelect command, which displays a menu and tracks the mouse as long as the mouse button stays down. The user doesn't literally pull it down: instead, it pops down as soon as the application program determines that the cursor has moved onto the title. As the user keeps the button pressed and moves the cursor down the menu, the Tool Kit highlights the item the cursor is pointing to by displaying it in inverse video.

When the user releases the button, the item that the cursor was pointing to is selected and the menu disappears. To tell the user that something is happening, the Tool Kit leaves the menu title in the menu bar highlighted. The application program turns off the highlighting of the title as soon as it finishes performing the selected operation.

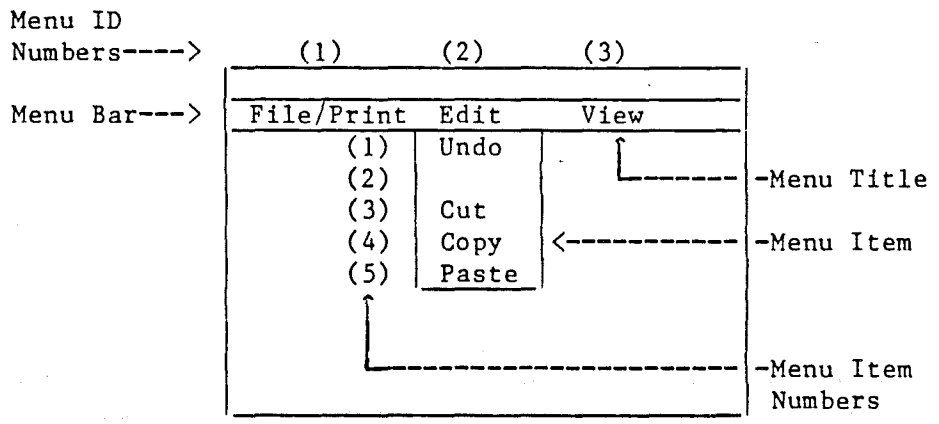
The data structures the Tool Kit uses to manage the menu information also contain information that is not displayed, but is returned to the program when a menu item is selected:

- a menu ID number for each menu

- a menu item number for each item

The program can set individual items or an entire menu to the disabled state. Disabled items or menus are not highlighted when the cursor moves onto them, and they cannot be selected.

Figure 1-1 Menu Components. Note: Numbers in parentheses are menu ID and menu item numbers and do not appear in the display. The menu item numbers are always sequential starting with 1, but the menu ID numbers can be in any order, as long as they're between 1 and 255.



The application program calls the SetMenu command with data structures containing the menu information the Tool Kit needs, and the Tool Kit displays the menus. The program can call SetMenu during the course of operation to change the contents of menus. The menu data structures are described in the "Menu Commands" section in Chapter 2.

When the FindWindow command detects the mouse button pressed in row Ø (the menu bar), the program calls the MenuSelect command. MenuSelect takes care of the entire selection process: it displays the menus and tracks the mouse position with the cursor for as long as the user holds down the mouse button. If the user selects a menu item, the MenuSelect command highlights the menu's title in the menu bar and returns the menu item number and the menu ID number. If the user doesn't select a menu item, the MenuSelect command returns a menu ID value of Ø.

Keeping the selected menu title highlighted while the operation is being performed gives useful feedback to the user. After the program has carried out the selected operation, it should call HiLiteMenu with menu ID set to Ø to un-highlight the menu title.

For menu items that are used often, the program can provide fast item selection; it does this by allowing the user to press keys instead of moving the mouse. To do this, the program specifies the keys the user can press to select the items in the menus. When the GetEvent command

returns a keypress, the program calls the MenuKey command. MenuKey gets the menu ID and the item number by searching the menu data structure for a matching key, and then highlights the selected menu title the same way MenuSelect does. After the operation has been performed, the program must use the HiLiteMenu command to turn off the highlighting of the selected title.

Windows

The Tool Kit's window commands make it possible for programs to use the mouse to control multiple windows. Figure 1-2 shows how windows appear on the display screen.

Parts of a Window

Each window has several parts, as shown in Figure 1-3. The two main parts of a window are the drag bar at the top, including the title of the window, and the content region, where the application displays information. The drag bar is used for moving the window around on the display. To move the window, the user positions the cursor on the drag bar and holds down the mouse button while moving the cursor to the desired position. The drag bar also contains the Close Box, or Go-Away Box. To close the window, the user clicks and releases the mouse while in the Go-Away Box.

The lower-right corner of the window contains the Grow Box, which is used to change the size of a window. To do this, the user presses the mouse button when the cursor is in the Grow Box, then holds the button down while moving the mouse. The display shows the new size of the window as an outline that moves around as the mouse moves. When the user releases the mouse button, the Tool Kit redisplays the window with its new size but without contents. The program puts appropriate text into the re-sized window by calling window commands or its own window subroutines.

Figure 1-2
Typical Display With Windows

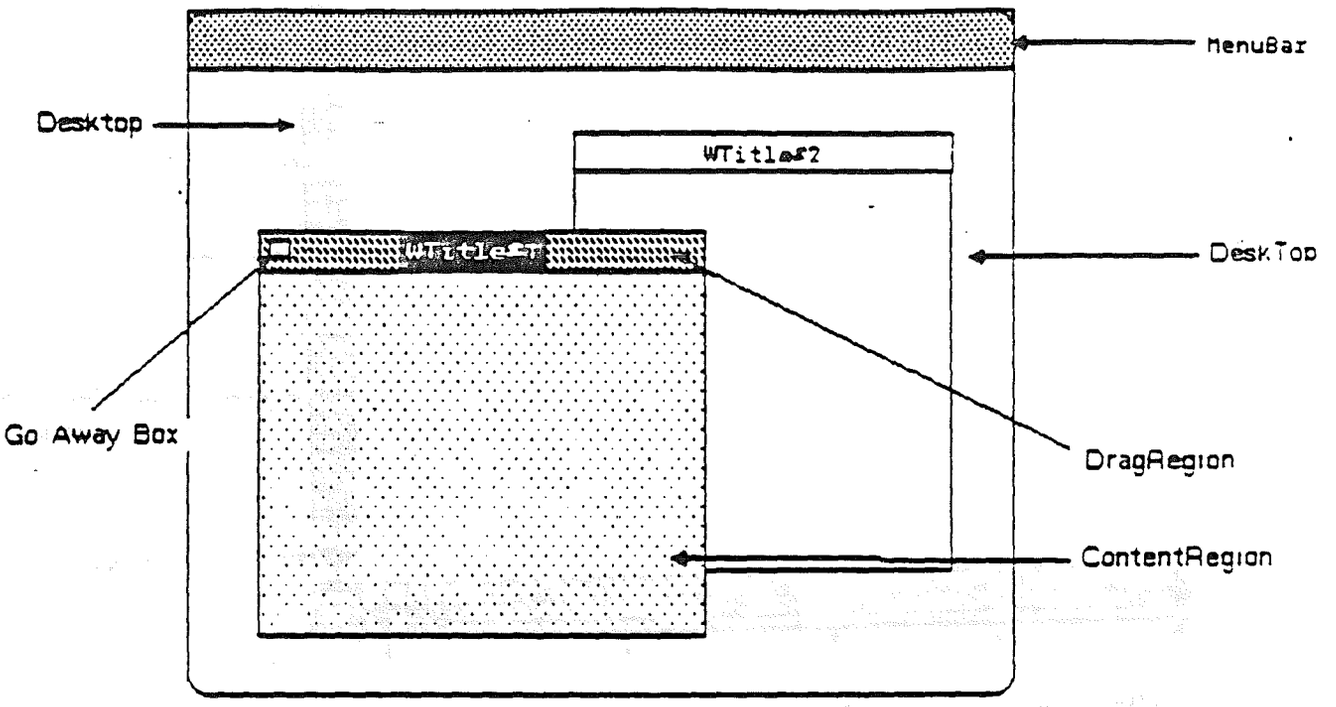


Figure 1-3 Parts of a Window

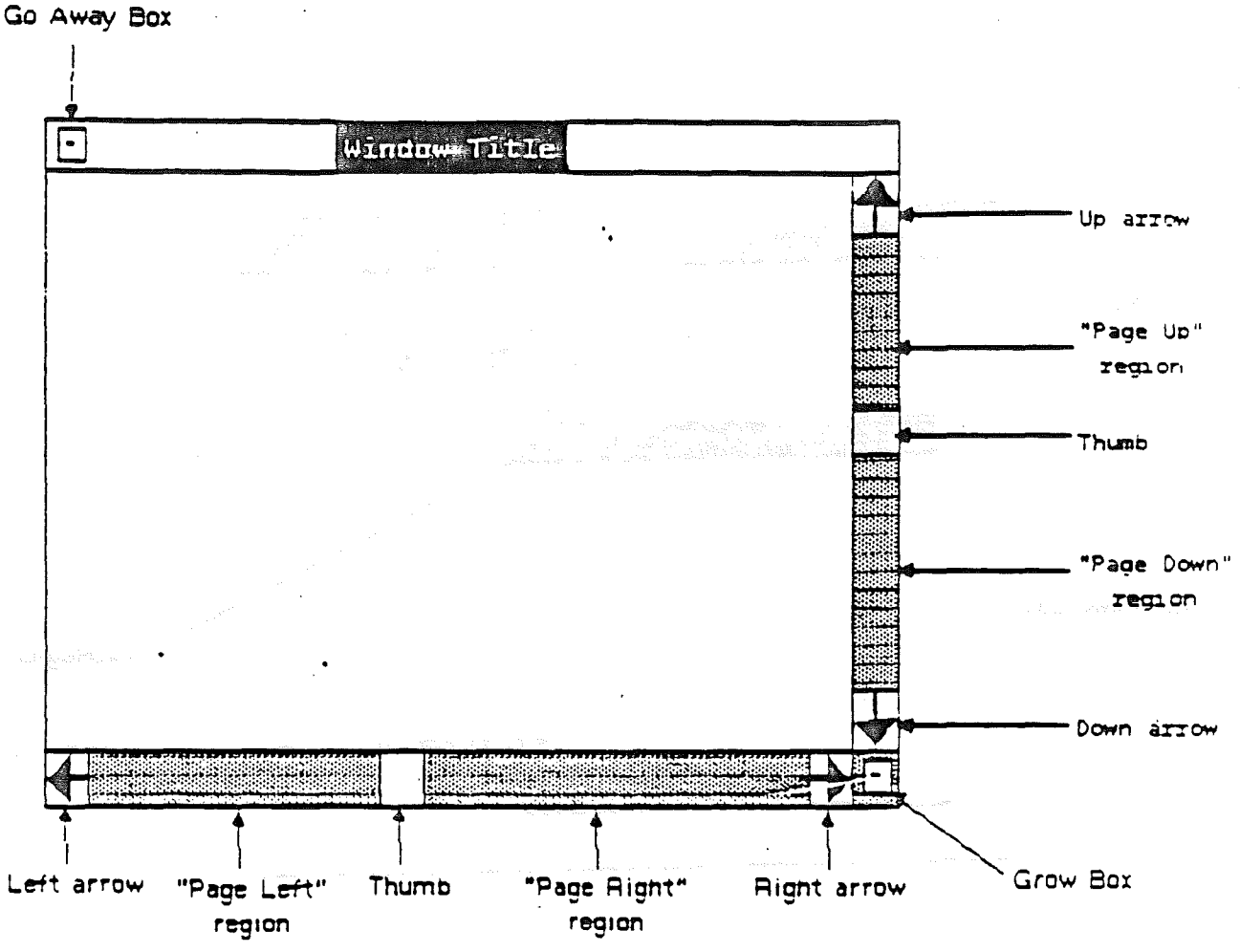
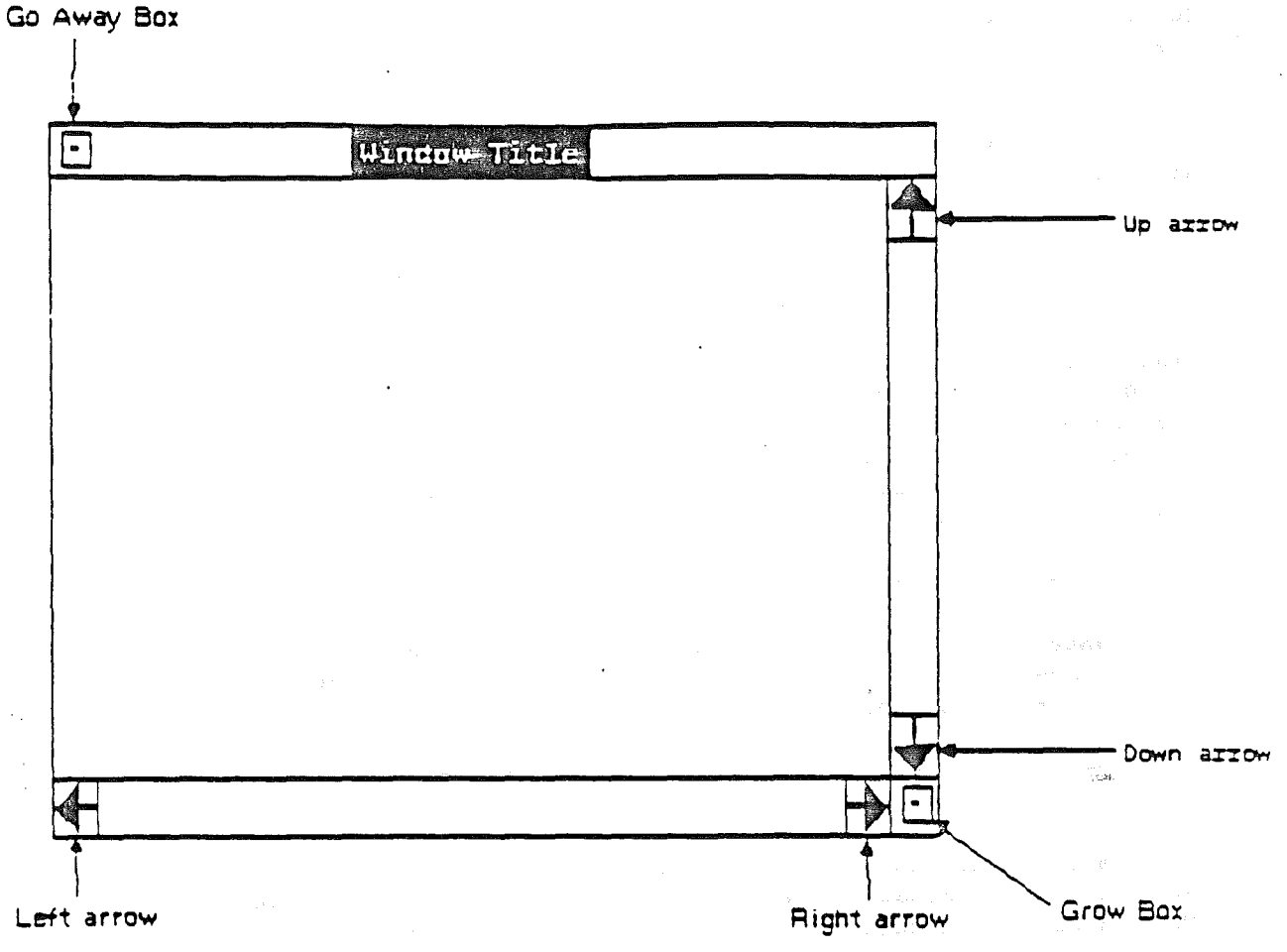


Figure 1-4
Window With Inactive Scroll Bars



Window Coordinates

Three different coordinate systems are used with the window commands:

- mouse coordinates, with X from Ø to 79 and Y from Ø to 23
- screen coordinates, with X from -8Ø to 159 and Y from -24 to 47
- window coordinates, with X from -8Ø to 159 and Y from -24 to 47

The mouse coordinates correspond to the absolute range of the display screen and are expressed as unsigned byte quantities. The window and screen coordinates are represented as two-byte signed quantities.

It is important to be aware of the ranges of the signed two-byte quantities because the Tool Kit routines make certain assumptions about the high byte. The only time the high byte is not simply the sign extension of the low byte's sign bit is when the value is in the range 128 to 159 for the X axis. The Y-axis quantities are also two-byte quantities for the sake of consistency. The only legal values of the high byte are $ØØ and $FF.

To be visible, characters must be in the top window, and their screen coordinates must be in the range from Ø to 79 in the X axis and Ø to 23 in the Y axis. What's more, if the width of the window is W and the length of the window is L, characters are visible only if their window coordinates are in the range from Ø to W - 1 in the X axis and Ø to L - 1 in the Y axis. The scroll bars are considered to be in the content area, so the useful content area range is from Ø to W - 3 if the vertical scroll bar space is used. Similarly, if there is a horizontal scroll bar, the useful content area range is from Ø to L - 2 in the Y axis.

Note: If a Grow Box is present, the vertical scroll bar space is used even if the scroll bar is not present. This ensures that the useful content area is always rectangular.

There must be at least one character in the window's content area for a Window Information Data Structure to be displayed correctly. The window length must be at least 1, or 2 if there is a horizontal scroll bar. Window width must be at least one, or three if there is a vertical scroll bar or a Grow Box. The maximum window width is 8Ø. The maximum length is 22 for normal windows, 23 for dialog windows.

Note: It is a good idea to keep window width greater than 3, else you can have a window whose title does not show, or even a window that cannot be dragged, but only closed, because there is only space enough for the Close Box.

A window can be placed in any position on the screen, including positions that make part of the window invisible. This is the reason for the ranges of the screen and window coordinates. Even though the ranges normally used are positive, you can get meaningful negative

values when you convert from one coordinate system to another. For example, a window's drag bar is always in the negative range of the window's Y axis.

Note: Windows are output-only devices; the Tool Kit will not copy their contents into user memory. The application program must ensure that the information in the content memory area and the contents of the window agree.

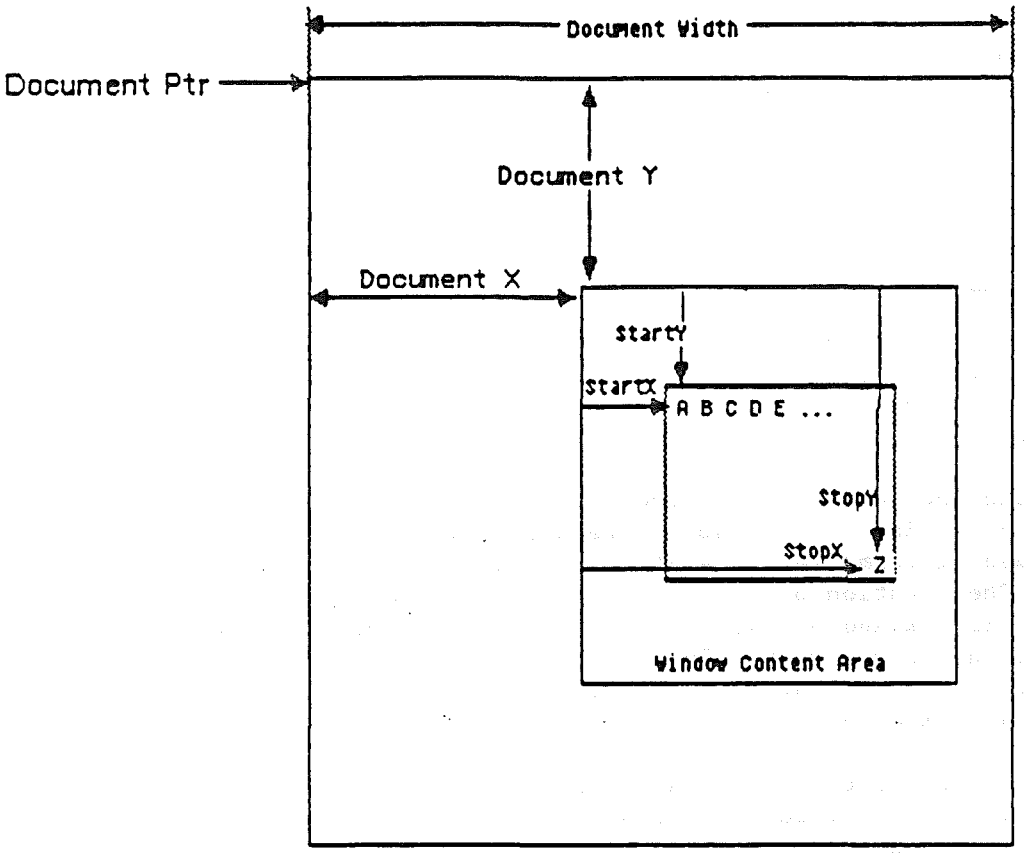
Document Information

The only document display feature built into the Tool Kit is a screen image of the text. Each line is padded with spaces on the right, and there are no special line delimiters; the number of characters per line is fixed.

To support the document display, the window management part of the Tool Kit needs certain information about the document. This information is in the Document Information Data Structure (Dinfo), described in Chapter 2. The location of the window in the document is specified by Dinfo quantities Dx and Dy (see Figure 1-5). The window can be placed anywhere within the document. In this sense, the document dimensions can be considered as a fourth coordinate system in which the window coordinates are embedded.

Other kinds of document displays are possible, but the routines to create them must be provided by the application program. For information about adding display routines, see the "SetUserHook" section in Chapter 2.

Figure 1-5
Location Parameters in a Document



Whenever a window is dragged, the Tool Kit must redisplay the content areas of the windows. The application program can override the Tool Kit's document display feature by having a routine that is called by the Tool Kit whenever the window is to be redisplayed. The program passes the address of this assembly language routine to the Tool Kit as part of the Window Information Data Structure, described in Chapter 2. Because of the way the Tool Kit saves zero-page locations, the program's routine cannot rely on the contents of those zero-page locations. Furthermore, the routine can only call the Tool Kit's window update commands to update the content region. These commands are WinChar, WinString, WinText, WinBlock, and WinOp. (Note: WinBlock uses a Document Information Data Structure.)

In the case where the window should not be refreshed automatically, the Tool Kit uses a type of event called an update event to signal the application that the window needs to be refreshed. The application specifies that a window is of this type by making the two-byte DInfo pointer (in the Window Data Structure) equal to zero. Please see the description in Chapter 2, under the GetEvent command.

Control Regions: The Scroll Bar

The only window control regions supported by the Tool Kit are the scroll bars displayed in the content region of the front (active) window (see Figure 1-3). Either horizontal or vertical scroll bars or both may be present.

An active scroll bar has several components, as shown in Figure 1-3. There are arrows at either end, an open box called the Thumb, and gray regions in between called Page-Up and Page-Down Regions in a vertical scroll bar, and Page-Left and Page-Right Regions in a horizontal scroll bar. (Sometimes the gray regions are called Page-Up and Page-Down Regions even in a horizontal scroll bar.)

An application program should support three different ways of scrolling the window contents using the scroll bars:

- Pressing the mouse button with the mouse in the arrows to scroll continuously as long as the button is down. The thumb moves to indicate the relative position of the window in the document.

- Pressing the mouse button in the Thumb itself, and dragging the Thumb to cause the corresponding scrolling of the document.

- Pressing the mouse button in a Page-Up or Page-Down Region to scroll the window up or down a page--that is, a window full--at a time.

The Thumb should appear right next to one of the arrows only when the first or last character of the document appears in the window. This ensures that the user can always page and that the Thumb can be used to get to the first and last characters of a document.

If the full width or length of a document appears in the window, the program should make the scroll bars reflect this by putting them into the inactive state, which shows the arrows, but no page regions or Thumb (see Figure 1-4).

If the window is so narrow that fewer than three character cells are available for the page regions and the Thumb, the Tool Kit will not display the Thumb. If fewer than three cells are available for the entire scroll bar, not even the arrows will show, and the user will be unable to scroll. Instead, the Tool Kit will display a gray region if the scroll bar is active, or a hollow region if it is inactive.

Interrupts and the Tool Kit

Tollow this sequence of steps to start the mouse:

(1) (For Pascal only) Call PascIntAdr to get the address of the Tool Kit's interrupt handler.

(2) (For Pascal only) Pass the interrupt address to the mouse firmware by calling SetMouse as described in Appendix B, "The Mouse Firmware Interface." Mouse Mode should be set to passive.

(3) Call StartDesktop with the UseInterrupts parameter set the way you want it for your program.

(4) (optional) Call SetUserHook to pass the addresses of your program's interrupt handlers, if any, to the Tool Kit.

The Tool Kit saves the interrupt state of the machine when your program calls the StartDeskTop command. When the program calls the StopDeskTop command, the Tool Kit sets the state of the machine to the state saved by StartDeskTop.

When you use the Tool Kit in Interrupt Mode, the Tool Kit provides the interrupt handler. In addition, the Tool Kit allows the application program to have interrupt handler subroutines that are called by the Tool Kit. The program passes each subroutine's address to the Tool Kit as a parameter by calling the SetUserHook command. This feature makes it possible for the application program to perform tasks at interrupt time.

A user hook routine that is called at interrupt time cannot call most Tool Kit commands. Doing so could put the Tool Kit into an unknown state. If a program needs to generate calls to the Tool Kit because of an interrupt, the interrupt routine should set a flag that the program checks during its main polling loop.

Lists of Tool Kit Commands

Tables 1-1 and 1-2 list all of the Tool Kit commands by name and function. For complete descriptions of the commands, please see Chapter 2. For the commands as called in the different languages, please see the chapter devoted to the appropriate language interface.

Table 1-1a.
Alphabetical List of Tool Kit Commands

Name	Number	Type
ActivateCtl	43	Control Region Commands
CheckEvents	5	Event-Handling Commands
CheckItem	16	Menu Commands
CloseAll	25	Window Commands
CloseWindow	24	Window Commands
DisableItem	15	Menu Commands
DisableMenu	14	Menu Commands
DragWindow	30	Window Commands
FindControl	39	Control Region Commands
FindWindow	26	Window Commands
FlushEvents	07	Event-Handling Commands
FrontWindow	27	Window Commands
GetEvent	6	Event-Handling Commands
GetWinPtr	45	Window Commands
GrowWindow	31	Window Commands
HideCursor	4	Cursor Commands
HiLiteMenu	13	Menu Commands
InitMenu	9	Menu Commands
InitWindowMgr	22	Window Commands
KeyboardMouse	48	Startup Commands
MenuKey	12	Menu Commands
MenuSelect	11	Menu Commands
ObscureCursor	44	Cursor Commands
OpenWindow	23	Window Commands
PascIntAdr	17	Startup Commands

Table 1-1b.
Alphabetical List of Tool Kit Commands, continued

Name	Type	
PeekEvent	21	Event-Handling Commands
PostEvent	46	Event-Handling Commands
ScreenToWindow	33	Window Commands
SelectWindow	28	Window Commands
SetCtlMax	4Ø	Control Region Commands
SetCursor	2	Cursor Commands
SetKeyEvent	8	Event-Handling Commands
SetMark	2Ø	Menu Commands
SetMenu	1Ø	Menu Commands
SetUserHook	47	Startup Commands
ShowCursor	3	Cursor Commands
StartDeskTop	Ø	Startup Commands
StopDeskTop	1	Startup Commands
TrackGoAway	29	Window Commands
TrackThumb	41	Control Region Commands
UpDateThumb	42	Control Region Commands
Version	19	Startup Commands
WinBlock	36	Window Commands
WinChar	34	Window Commands
WinOp	37	Window Commands
WindowToScreen	32	Window Commands
WinString	35	Window Commands
WinText	38	Window Commands

Table 1-2a. Startup Commands

Number	Name	Description
Ø	StartDeskTop	Activates the mouse and the Tool Kit routines
1	StopDeskTop	Inactivates the mouse and the Tool Kit routines
17	PascIntAdr	Gets the interrupt handler address for Pascal (not applicable to BASIC)
47	SetUserHook	Sets the address of the user's interrupt handler
19	Version	Returns the version and revision numbers of the Tool Kit
48	KeyboardMouse	Conditions Tool Kit to perform next operation in emulation mode

Table 1-2b. Cursor Commands

Number	Name	Description
2	SetCursor	Sets the character used for displaying the cursor
3	ShowCursor	Makes the cursor visible
4	HideCursor	Makes the cursor invisible
44	ObscureCursor	Makes the cursor invisible until the mouse moves

Table 1-2c. Event-Handling Commands

Number	Name	Description
5	CheckEvents	Reads the mouse, moves the cursor to the new position, and posts event, if any
6	GetEvent	Gets next event; if none, gets mouse position
46	PostEvent	Posts an event in the event queue
7	FlushEvents	Empties the event queue
8	SetKeyEvent	Specifies whether Tool Kit handles keyboard events
21	PeekEvent	Returns event data without removing it from the queue

Table 1-2d. Menu Commands

Number	Name	Description
9	InitMenu	Allocates memory for temporary screen save
1Ø	SetMenu	Initializes a menu bar data structure and displays the menu bar
11	MenuSelect	Interacts with mouse to display menu and return selection, if any
12	MenuKey	Selects a menu item to match a keypress
13	HiLiteMenu	Turns highlighting of menu title on or off
14	DisableMenu	Inhibits highlighting and selection over a whole menu
15	DisableItem	Inhibits highlighting and selection of a menu item
16	CheckItem	Turns checkmark next to item on or off
2Ø	SetMark	Sets the character to use as checkmark

Table 1-2e. Window Commands

Number	Name	Description
22	InitWindowMgr	Initializes the open window list and buffer area
23	OpenWindow	Passes the Tool Kit a pointer to a Window Information Data Structure
24	CloseWindow	Deletes a window
25	CloseAll	Deletes all windows
45	GetWinPtr	Gets the pointer to the Window Information Data Structure (not applicable to Pascal)
26	FindWindow	Finds the window region that contains a given point
27	FrontWindow	Returns the ID number of the front window
28	SelectWindow	Makes a window the front (active) window
29	TrackGoAway	Returns whether the mouse button was released in a Go-Away Box
3Ø	DragWindow	Displays window outline during drag, then redisplays windows
31	GrowWindow	Displays window outline during grow, then redisplays windows
32	WindowToScreen	Converts window coordinates to screen coordinates
33	ScreenToWindow	Converts screen coordinates to window coordinates
34	WinChar	Writes a character in a window
35	WinString	Writes a string in a window
38	WinText	Writes text in a window
36	WinBlock	Writes a block of text in a window
37	WinOp	Clears all or part of a window

Table 1-2f. Control Region Commands

Number	Name	Description
39	FindControl	Returns whether the mouse is in a control region
4Ø	SetCtlMax	Sets the range of a scroll bar
41	TrackThumb	Tracks the Thumb until the mouse button is released
42	UpdateThumb	Displays the Thumb in a given position
43	ActivateCtl	Changes the state of a scroll bar (active or inactive)

Mouse Emulation

Although the menu and window capabilities of the Apple II MouseText Tool Kit are normally used with the AppleMouse II, it is possible to run a program using the Tool Kit on a computer that doesn't have a mouse. It is also possible to use the keyboard to control the menus and windows, even on a computer that has a mouse. Even when mouse emulation is going on, the Tool Kit still responds to mouse movement and button operation.

The first method of mouse emulation is called Keyboard Mouse Mode. It enables the application to support menu selection and window manipulation by means of keyboard commands. Note that the application must include the appropriate calls to provide this feature, in contrast to the Safety-Net Mode, which is transparent to the application.

The second method of mouse emulation, Safety-Net Mode, is provided for use with a computer that doesn't have a mouse. This might happen, for example, when a dealer needs to demonstrate an application and a mouse-equipped computer is inoperative or is not available.

Keyboard Mouse Mode

The Keyboard Mouse Mode of mouse emulation makes it possible for applications to provide keyboard commands for operations that normally require the mouse: selecting menus, and dragging and changing the size of windows. The choice of commands for selecting these mouseless operations is up to the application program. The recommended key sequences for use in English-speaking countries are:

ESC to get the menu display.
OPEN-APPLE-D or SOLID-APPLE-D to drag the window.
OPEN-APPLE-G or SOLID-APPLE-G to grow the window.

When the Tool Kit is in Keyboard Mouse Mode, it is performing one of those three operations and remains in Keyboard Mouse Mode until the operation is completed. Unlike Safety-Net Mode, the user doesn't have to hold a key down.

When the user initiates Keyboard Mouse Mode, the Tool Kit makes the cursor visible, even if it was previously hidden or obscured. When the keyboard operation is completed, the Tool Kit returns the cursor to its previous state of visibility.

In Keyboard Mouse Mode, the cursor keys move the cursor around on the display. If the user is doing a drag or grow, the OPEN-APPLE key acts as an accelerator for the cursor keys. With the OPEN-APPLE key down, pressing left or right arrow keys moves the cursor sideways by 10 spaces at a time. Likewise, the up and down arrow keys move the cursor up and down 5 rows at a time.

The user can terminate a Keyboard Mouse Mode operation in four different ways:

by pressing the ESC key.
by pressing the RETURN key.
by pressing a valid command key.
by pressing and releasing the mouse button.

When the user presses the ESC key, the Tool Kit cancels the operation and returns the cursor to its former position.

When the user presses the RETURN key, the Tool Kit completes the operation.

When the user presses a valid command key, the Tool Kit terminates the operation and then posts an event for the command key. If the operation was a menu selection, the Tool Kit cancels the operation. If the operation was a drag window or a grow window, the Tool Kit completes the operation. In any case, the Tool Kit returns the cursor to its original position.

When the user presses and releases the mouse button, the mouse button up event signals completion of the operation and initiates execution of the selected command, just as if the mouse had been used throughout.

After menu selection, the Tool Kit records the position of the cursor (that is, the item that is highlighted) and returns to that position (and item) when the user selects the menu again.

The three operations that can be performed in Keyboard Mouse Mode are selecting from the menu, dragging a window, and growing window. To perform these operations normally, the application calls the appropriate command: MenuSelect, DragWindow, or GrowWindow, respectively. To perform one of them in Keyboard Mouse Mode, the application must first call the KeyboardMouse command, then call the MenuSelect, DragWindow, or GrowWindow command. This causes the Tool Kit to perform the command in Keyboard Mouse Mode, functioning in the manner described above. The KeyboardMouse command has no parameters.

There is an alternative way for the application to get into Keyboard Mouse Mode, and that is calling the MenuKey command with ESC as the keystroke. That has the same effect as calling KeyboardMouse followed by MenuSelect: it initiates a menu select operation in Keyboard Mouse Mode.

Safety-Net Mode

Safety-Net Mode uses inputs from the keyboard to provide the usual mouse operations of moving the cursor around on the desktop and selecting menus. When the Tool Kit is in Safety-Net Mode, the application program works normally: all command calls are the same, and the program need not even take into account the fact that there is no mouse.

The user puts the Tool Kit into Safety-Net Mode by pressing the OPEN-APPLE key and holding it down, then pressing and releasing the SOLID-APPLE key. The Tool Kit generates a click to acknowledge that it is in Safety-Net Mode. The Tool Kit remains in Safety-Net Mode as long as the user continues to hold down the OPEN-APPLE key.

In Safety-Net Mode, the cursor keys take the place of the mouse for moving the cursor around. Each time you press a cursor key, the cursor moves one space in the direction indicated on the key. The cursor keys do not have wrap-around: when you have moved the cursor all the way to a screen edge, pressing the same cursor key again has no effect.

In Safety-Net Mode, the SOLID-APPLE key takes the place of the mouse button. Pressing the SOLID-APPLE key is like pressing the mouse button.

Note: All during Safety-Net Mode, the Tool Kit reads the cursor keys and the SOLID-APPLE key even if the application program has specified that the keyboard is to be ignored.

Chapter 2

Specifications of the Commands

This chapter gives the command number and the contents of the command list for each of the Tool Kit commands. Each command occupies a separate page and has the format shown here:

Name

Function:

A brief statement of the command's function

Command Number:

The number, in decimal and hex, that specifies the command

Parameter List:

A description of the parameters being passed

Description:

A full description of the command

Error Codes:

A list of error codes for the command

In addition to the error codes listed with the commands, there are three generic error codes that can be returned by any command. These error codes are

1 ($Ø1) Illegal command number

2 ($Ø2) Wrong number of parameters

3 ($Ø3) StartDeskTop hasn't been called

All of the error codes are listed together in Appendix F.

Startup Commands

A program will normally call these commands once to set up the operating environment for the program. For example, calling the Version command tells the program which version of the Tool Kit it is using.

Your program calls the StartDeskTop command to activate the mouse and set the Operating Mode for the Tool Kit, and the StopDeskTop command to deactivate the mouse and the Tool Kit.

Pascal programs must also call the PascIntAdr command to get the address of the Tool Kit's interrupt handler so the Pascal interface can install the interrupt handler. (See Chapter 4, "The Pascal Interface.")

StartDeskTop

Function:

StartDeskTop initializes the mouse and the Tool Kit routines.

Command Number: Ø ($ØØ)

Parameter List:

6 (input, byte) the number of parameters

id (input, byte) machine ID byte: $Ø6 = Apple IIe or IIc

sid (input, byte) subsidiary ID byte:
$EA = Apple IIe
$EØ = Apple IIe with revised ROM
$ØØ = Apple IIc

op (input, byte) operating system byte:
Ø = ProDOS
1 = Pascal

s# (input or output, byte) slot number of the mouse card

int (input or output, byte) interrupt usage:
Ø = Passive Mode only
1 = use interrupts

col (input, byte) number of text columns:
Ø = 4Ø columns
1 = 8Ø columns

Description:

StartDeskTop saves the current state of the computer, initializes the Tool Kit routines, and activates the mouse card. If the calling program specifies a slot number of Ø, StartDeskTop will check the slots for a mouse card and use the first one it finds, returning its slot number in s#. If no mouse card is found, StartDeskTop will set Passive Mode and return the int parameter as Ø.

If the program requires that the mouse card be present, it should set the high bit of the s# parameter on before calling StartDeskTop. When that bit is set, StartDeskTop will return an error condition if it doesn't find a mouse card.

If the program uses interrupts, it must set the int parameter to 1.

The ID bytes are the values found at locations $FBB3 and $FBCØ in the Apple IIe and Apple IIc. Version 2 of the MouseText Tool Kit requires the machine ID byte to be $Ø6.

The Tool Kit doesn't do anything about the 8Ø-column firmware. The program has to activate the firmware if it is needed.

StartDeskTop sets the cursor to the arrowhead character (ASCII value $Ø2) and sets it hidden; after calling StartDeskTop, an application program can call ShowCursor immediately.

Error Codes:

4 ($Ø4)	Machine or operating system not supported
5 ($Ø5)	Invalid slot number (less than Ø or greater than 7)
6 ($Ø6)	Card not found
11 ($ØB)	Could not install interrupt handler

StopDeskTop

Function:

StopDeskTop deactivates the mouse and the Tool Kit routines.

Command Number: 1 ($Ø1)

Parameter List:

Ø (input, byte) the number of parameters

Description:

StopDeskTop hides the cursor, removes the link to the interrupt handler, and sets the mouse to an inactive state. StopDeskTop then restores the computer to the initial state that was saved by StartDeskTop.

Error Codes:

(none)

PascIntAdr

Function:

PascIntAdr returns the address of the Tool Kit's interrupt handler.

Command Number: 17 ($11)

Parameter List:

1 (input, byte) the number of parameters

Adr (output, word) the address of the interrupt handler

Description:

PascIntAdr returns the address of the Tool Kit's interrupt handler in the Adr parameter. Your Pascal program can pass that address on to the Mouse Attach Driver when it calls SetMouse. The SetMouse call should always specify Passive Mode along with the interrupt address. The program should do this before calling StartDesktop, which will enable interrupts if its Int parameter is set to 1.

See also Chapter 4, "The Pascal Interface."

Note: This command is used only in Pascal programs.

Error Codes:

(none)

SetUserHook

Function:

SetUserHook sets the address of the user's interrupt handler.

Command Number: 47 ($2F)

Parameter List:

2 (input, byte) the number of parameters

Id (input, byte) the ID number of the interrupt routine

Adr (input, word) the address of the interrupt routine

Description:

The SetUserHook command sets the starting address of the application program's interrupt handler routine so that the Tool Kit can pass control to it whenever CheckEvent is called. In Interrupt Mode, the Tool Kit calls CheckEvent internally during interrupt servicing, so routines installed by SetUserHook become interrrupt service routines for the application.

CheckEvent can pass control to the program's interrupt routine either before or after it checks events. The Id parameter determines at which point CheckEvent will call the interrupt routine. If Id = Ø, CheckEvent will call the interrupt routine before checking events, and if Id = 1, CheckEvent will call it after checking events. In this way there can be an interrupt routine either before or after event checking, or there can be two user routines, one before and one after event checking.

If the interrupt routine that is called before event checking (Id = Ø) returns to the Tool Kit with the carry flag clear, CheckEvent will not check events. This allows the application program to handle event checking itself and bypass event checking by the Tool Kit.

If the Adr parameter is set to Ø, SetUserHook removes any routine previously installed.

WARNING
The user interrupt routine can call only Tool Kit commands PostEvent, ShowCursor, HideCursor, and SetCursor. Calling any other commands from the user interrupt routine could put the Tool Kit into an unknown and bizarre state.

Error Codes:

21 ($15) Illegal Id parameter (must be Ø or 1)

Version

Function:

Version returns the Tool Kit's version and revision numbers.

Command Number: 19 ($13)

Parameter List:

2 (input, byte) the number of parameters

Ver (output, byte) the version number of the Tool Kit

Rev (output, byte) the revision number of the Tool Kit

Description:

The Version command returns the version and revision numbers of the Tool Kit. The program can use these numbers to determine compatibility.

Error Codes:

(none)

KeyboardMouse

Function:

KeyboardMouse makes the next command work in mouse emulation mode, if the command is one of the three that work in that mode.

Command Number: 48 ($3Ø)

Parameter List:

Ø (input, byte) the number of parameters

Description:

The KeyboardMouse command is a function call; it has no parameters.

The KeyboardMouse command is used in conjunction with the three commands that can operate in mouse emulation mode: MenuSelect, DragWindow, and GrowWindow. To make one of these three commands operate in emulation mode, all you have to do is call the KeyboardMouse command just before calling one of them.

An application can also get this form of mouse emulation on the MenuKey command by calling the command with the ESC key as the keystroke. That has the same effect as calling KeyboardMouse and then calling MenuSelect.

Error Codes:

(none)

Cursor Commands

Your program calls these commands to select the character to display as the cursor and to turn the cursor on and off.

SetCursor

Function:

SetCursor sets the character used for displaying the cursor.

Command Number: 2 ($Ø2)

Parameter List:

l (input, byte) the number of parameters

cc (input, byte) character to use as cursor

Description:

SetCursor sets the character displayed as the cursor. Characters normally used as the cursor include the following Mouse Text characters:

Arrowhead (ASCII value Ø2 $Ø2)
Hourglass (ASCII value Ø3 $Ø3)
Checkmark (ASCII value Ø4 $Ø4)
Text Cursor (ASCII value 2Ø $14)
Cell Cursor (ASCII value 29 $1D)

If the cursor is visible, it changes to the new character as soon as SetCursor is called. Each time the cursor is moved, if it is visible, the Tool Kit saves the character at the new cursor position and replaces it with the character specified by SetCursor.

Error Codes:

(none)

ShowCursor

Function:

ShowCursor makes the cursor visible.

Command Number: 3 ($Ø3)

Parameter List:

Ø (input, byte) the number of parameters

Description:

ShowCursor makes the cursor visible. If the cursor is obscured, ShowCursor has no effect.

Error Codes:

(none)

HideCursor

Function:

HideCursor makes the cursor invisible.

Command Number: 4 ($Ø4)

Parameter List:

Ø (input, byte) the number of parameters

Description:

HideCursor makes the cursor invisible. If the cursor is obscured, ShowCursor has no effect.

Error Codes:

(none)

ObscureCursor

Function:

ObscureCursor makes the cursor temporarily invisible.

Command Number: 44 ($2C)

Parameter List:

Ø (input, byte) the number of parameters

Description:

ObscureCursor makes the cursor invisible until the mouse moves; then the cursor reappears. An appropriate time to use OscureCursor is when text is being entered, to keep the cursor from obstructing the view of the text. As soon as the user moves the mouse to perform another task, the cursor reappears.

Error Codes:

(none)

Event-Handler Commands

The Tool Kit's event-handler commands maintain an event queue for mouse and keyboard events. The CheckEvents command posts events in the queue and updates the mouse position. The GetEvents command gets the next event in the queue.

CheckEvents

Function:

CheckEvents reads the mouse, moves the cursor to the new mouse position, and posts an event, if any.

Command Number: 5 ($Ø5)

Parameter List:

Ø (input, byte) number of parameters

Description:

CheckEvents reads the mouse and posts a mouse event if the button state changed. If a key on the keyboard is down and keypress events are to be checked, CheckEvents posts a keypress event and clears the keyboard strobe. (If a previous call to SetKeyEvent has disabled keypress events, CheckEvents ignores the keypress.) CheckEvents also updates the cursor position to the X and Y values from the mouse.

If the program is using Interrupt Mode, the interrupt handler calls CheckEvents. If the program calls CheckEvents in Interrupt Mode, the Tool Kit returns an error.

In Passive Mode, the GetEvent command calls CheckEvents internally. If the program is using Passive Mode, it should call CheckEvents or GetEvent often to ensure smooth cursor motion.

Remember: CheckEvents is the only command that reads the mouse and updates the cursor position.

An application program can have an interrupt-service routine of its own that augments or even replaces the functions of CheckEvents. CheckEvents can pass control to the routine either before or after event checking. The program can even have two interrupt routines, one called before event checking and one after. See the SetUserHook command in the "Startup Commands" section to see how to do this.

Error Codes:

7 ($Ø7) Interrupt Mode in use. (Program specified Interrupt Mode in StartDeskTop, so it can't call CheckEvents.)

GetEvent

Function:

GetEvent fetches the next event from the event queue. If there is none, GetEvent returns the mouse position. In Passive Mode, GetEvent calls CheckEvents.

Command Number: 6 ($Ø6)

Parameter List:

3 (input, byte) number of parameters

et (output, byte) event type:
Ø = no event
1 = button down
2 = button up
3 = key pressed
4 = drag event
5 = Apple key down
6 = update event

eb1 (output, byte) event byte 1: X coordinate or key value

eb2 (output, byte) event byte 2: Y coordinate or key modifier

Description:

GetEvent fetches the next event from the event queue so the program can respond to the pressing of a key or the mouse button. In Passive Mode, GetEvent calls CheckEvent internally to make sure the lastest event gets processed.

The event-type variable is a byte that indicates what happened to cause the event. If the event type is Ø, 1, 2, 4, or 5, the event bytes are the X and Y coordinates of the mouse position from the last call to CheckEvents. If the event type is 3, the event bytes are the key and the key modifier. The high bit of the key value is Ø. The key modifier values are:

Ø = no modifier
1 = OPEN-APPLE pressed
2 = SOLID-APPLE pressed
3 = both Apple keys pressed

The drag event (et parameter = 4) is similar to a no event except that the mouse button is still down. After getting a button-down event,

the program should get drag events or a button-up event. If the program gets a no event while waiting for a button-up event, that indicates that a mouse-up event was missed and that you don't know what the mouse position was at that time (you only know its present position). If this happens, the program must cancel any operation that is in progress.

The Apple-key down event indicates that one of the Apple keys was down when the mouse button was pressed.

By the Way: This is similar to the shift click event on a Lisa or a Macintosh. We can't read the shift keys on the Apple II, but we can read the Apple keys.

An event type of 6 indicates an update event. This indicates that a window that cannot be automatically refreshed needs updating. The window ID is returned in ebl, the key value parameter. This event only occurs when the application has set the DInfo pointer in the Window Data Structure to zero, indicating that the window cannot be automatically refreshed.

Error Codes:

(none)

PostEvent

Function:

PostEvent posts an event into the event queue.

Command Number: 46 ($2E)

Parameter List:

3 (input, byte) number of parameters

et (input, byte) event type:
Ø = no event
1 = button down
2 = button up
3 = key pressed
4 = drag event
5 = Apple key down
6 = update event

eb1 (input, byte) event byte 1: X coordinate or key value

eb2 (input, byte) event byte 2: Y coordinate or key modifier

Description:

PostEvent posts an event into the event queue. The parameter list is the same as for GetEvent except that all of the parameters are inputs.

PostEvent can have an event type like the ones returned by GetEvent (et = Ø, 1, ...5) or it can have a type defined by the application program (et = 128, 129, ...255). Any other value for the et parameter is illegal. The Tool Kit ignores events of type 128-255.

Error Codes:

19 ($13) The event queue is full; the event was not posted.

2Ø ($14) Illegal event type; the event was not posted.

FlushEvents

Function:

FlushEvents empties the event queue.

Command Number: 7 ($Ø7)

Parameter List:

Ø (input, byte) number of parameters

Description:

FlushEvents empties the event queue.

Error Codes:

(none)

SetKeyEvent

Function:

SetKeyEvent specifies whether Tool Kit treats keypresses as events.

Command Number: 8 ($Ø8)

Parameter List:

l (input, byte) number of parameters

sk (input, byte) set keyevent:
Ø = don't check keyboard,
1 = check the keyboard

Description:

SetKeyEvent specifies whether Tool Kit posts keypresses as events. If the value of sk is 1, the Tool Kit reads the keyboard. If a key is pressed, the Tool Kit posts a key event and clears the key strobe. If the value of sk is Ø, the Tool Kit doesn't handle keypresses. At start up, the Tool Kit is set to post keyboard events.

The Tool Kit handles keypresses as events in the queue, providing a form of type-ahead. This means that Pascal programs don't need the Keypress function in the Applestuff Unit as long as they're using the Tool Kit.

Error Codes:

(none)

PeekEvent

Function:

PeekEvent reports on the next event without removing it from the queue.

Command Number: 21 ($15)

Parameter List:

3 (input, byte) number of parameters

et (output, byte) event type:
Ø = no event
1 = button down
2 = button up
3 = key pressed
4 = drag event
5 = Apple key down
6 = update event

eb1 (output, byte) event byte 1: X coordinate or key value

eb2 (output, byte) event byte 2: Y coordinate or key modifier

Description:

PeekEvent returns information from the next event in the event queue, but does not remove the event from the queue. The parameters are the same as for the GetEvent command, described earlier.

Error Codes:

(none)

Menu Commands

The Tool Kit's menu commands provide menu display and selection functions. Once you have set up the menu data structures with SetMenu, the MenuSelect command will display a menu, track the mouse and move the cursor, highlight menu items as the cursor moves onto them, return with the menu ID and item numbers selected, and leave the menu title highlighted. Other menu commands inhibit menus or menu items and display a checkmark beside specified menu items.

It is the responsibility of the application program to ensure that menu titles do not extend past the right edge of the screen. The program must make sure that a menu's width is always less than the screen width minus two (38 or 78) and that a menu's length is always less than screen length minus two (22). Otherwise, the menu routines can write into main memory when they should be writing to the display, thereby clobbering screen holes or program memory.

Keys in Menus

The MenuKey command gives your program the ability to use keypresses to select menu items. Typically, you use a combination keypress consisting of a letter key plus one of the Apple keys. Menu items that can be selected in this way are indicated by the OPEN-APPLE or SOLID-APPLE icon and the specified letter or other key displayed to the right of the menu item. If an item can be selected using either type of apple icon, the OPEN-APPLE icon appears with the letter in the menu.

You can also specify a control character as the keypress that selects a menu item. You do this by setting either Character 1 or Character 2 in the Menu Item Block to a value from 1 to 31, corresponding to a control character. (Menu Item Blocks are defined in Table 2-4.) You need not set the modifier bits in the Item Option Byte.

When you specify a control key to select an item, the Tool Kit displays a diamond icon and the key to the right of the menu item. Only the character in Character 1 will be used, even if you made Character 2 a control character.

Keypresses with the CONTROL key are easier to touch-type than those with the Apple keys, but you should still use the Apple-key combinations for most items and reserve the use of control keys for high-speed or repetitive functions where the ability to touch-type the command is important.

The user will expect control keys to be used for the same functions across different products. Apple has defined the menu functions of most of the control keys, as shown in Table 2-1.

If the user presses a key other than one of those specified in the menu, the Tool Kit generates a beep.

Table 2-1.
Control Keys for Menu Items

Control Key	Function
CTRL-B	Boldface
CTRL-C	Copy
CTRL-D	Delete
CTRL-E	Editing type, insert or overstrike cursor
CTRL-F	Forward delete
CTRL-H	Left arrow
CTRL-I	Tab
CTRL-J	Down arrow
CTRL-K	Up arrow
CTRL-L	Begin or end underline
CTRL-M	Return
CTRL-P	Print
CTRL-U	Right arrow
CTRL-V	Paste
CTRL-X	Cut
CTRL-Z	Zoom
CTRL-[	Escape

InitMenu

Function:

InitMenu establishes an area of memory that will be used to save the part of the display obscured by menus.

Command Number: 9 ($Ø9)

Parameter List:

2 (input, byte) number of parameters

sa (input, word) save area: pointer to reserved memory area

sas (input, word) save area size: number of bytes reserved

Description:

During calls to MenuSelect, the part of the display obscured by a menu must be saved so that it can be replaced when the menu goes away. The application program must provide memory space and reserve it for use by the Tool Kit.

You can determine the amount of memory space to reserve for menu displays by calculating the screen area of the largest menu in the program. The largest menu could have a large screen area because it has many items, or it could have only a few items, each of which is very long.

You calculate the screen area of a menu by taking the product of the number of items in the menu, plus 1, times five bytes more than the length of the longest item string in that menu. If you are using keys to select items, each item string must include three bytes to display a space, an Apple icon, and the key that selects the item.

When the program calls the SetMenu command to initialize a menu bar, SetMenu checks whether the amount of memory reserved by InitMenu is enough for a particular menu and returns an error if it is not.

Error Codes:

(none)

SetMenu

Function:

SetMenu initializes the menu bar data structure and displays the menu bar.

Command Number: 1Ø ($ØA)

Parameter List:

1 (input, byte) number of parameters

mbs (input, word) pointer to menu bar structure

Description:

SetMenu initializes a menu bar data structure and displays the menu bar. Given a pointer to a menu bar structure (see Tables 2-2 and 2-4), SetMenu fills in the data required by the menu commands and saves the pointer for their use. Once SetMenu has been called, the program must not move the data structure.

SetMenu checks to make sure that the memory area reserved by InitMenu is enough to handle the display area that will be obscured by the menu bar specified by the data structure. If it is not, SetMenu returns an error, but it still displays the menu bar.

Error Codes:

1Ø ($ØA) Save area (from InitMenu) is too small.

Table 2-2. Data Structure for a Menu Bar

Parameter Function	Parameter Size	Note
Number of Menus	1 byte	
Reserved for Future Use	1 byte	
First Menu Block:		
Menu ID (can't be Ø)	1 byte	
Menu Option Byte	1 byte	
Pointer to Title String	2 bytes	
Pointer to Menu Data Structure	2 bytes	
X Position for Title Display	1 byte	*
Left for HiLite and Select	1 byte	*
Right for HiLite and Select	1 byte	*
Reserved for Future Use	1 byte	*
Second Menu Block:		
(same structure as First Menu Block)		
.		
.		
Last Menu Block		
(same structure as First Menu Block)		

*Indicates items filled in by Tool Kit.

Table 2-3. Contents of Option Byte in Each Menu Block (see Table 2-2)

Bit Number	Bit Function	Note
7	Disable Flag	*
6	Reserved for Future Use	
5	Reserved for Future Use	
4	Reserved for Tool Kit	
3	Reserved for Tool Kit	
2	Reserved for Future Use	
1	Reserved for Future Use	
Ø	Reserved for Future Use	

*Disable Flag is updated by DisableMenu command. By setting the flag

off before calling SetMenu, the program can make the menu start out disabled.

Table 2-4. Data Structure for a Menu

Parameter Function	Parameter Size	Note
Number of Items	1 byte	
Left Column of Save Box	1 byte	1
Right Column of Save Box	1 byte	1
Reserved for Future Use	1 byte	1
First Menu Item Block:		
Item Option Byte	1 byte	
Mark Character	1 byte	2
Character 1 (high bit off)	1 byte	3
Character 2 (high bit off)	1 byte	3
Pointer to Item String	2 bytes	
Second Menu Item Block:		
(Same structure as First Menu Item Block)		
.		
.		
Last Menu Item Block:		
(Same structure as First Menu Item Block)		

1 Indicates items filled in by Tool Kit.
2 Updated by the SetMark command. The program can set the initial mark character in the data structure, but after that it should change the mark character only by calling SetMark.
3 The program should set this byte to Ø if not using characters.

Table 2-5. Contents of Option Byte in Menu Data Structure

Bit Number	Bit Function	Notes
7	Disable Flag	1,5
6	Item Is Filler	2
5	Item Is Checked	3,5
4	Reserved for Tool Kit	
3	Reserved for Tool Kit	
2	Item Has Mark	3,5
1	Modifier Is SOLID-APPLE Key	
Ø	Modifier Is OPEN-APPLE Key	

1 Updated by the DisableItem Command.

2 If the "Item Is Filler" bit in the Option Byte is on, then Character 1 of the Menu Item Block (see Table 2-4) is the character to use for filler; otherwise, Character 1 and Character 2 are the uppercase and lowercase values of the key that identifies the item when MenuKey is called.

3 Updated by the CheckItem command.

4 Used only with Applesoft BASIC; set to Ø otherwise. See Appendix F, "Applesoft String Options."

5 The program can set the initial states of these flags in the data structure before calling the SetMenu command. After that, the program should update the flags only by calling the appropriate commands.

MenuSelect

Function:

MenuSelect interacts with the mouse to display a menu and return the selection, if any.

Command Number: 11 ($ØB)

Parameter List:

2 (input, byte) number of parameters

id (output, byte) menu ID, Ø = no menu item chosen

in (output, byte) menu item number, undefined if id = Ø

Description:

MenuSelect performs the interactive display of menus while the user keeps the mouse button depressed. MenuSelect does not return until the user releases the button and a button-up event occurs.

The application program calls MenuSelect whenever the user presses the mouse button on line Ø of the display. As the user moves the mouse up and down the menu display, MenuSelect tracks the mouse and updates the cursor. When the cursor moves onto a menu item, MenuSelect highlights the name of the item.

When the user releases the mouse button while the cursor is on a menu item, MenuSelect removes the menu from the display, highlights the menu title, and returns the menu ID number and the item number. After the program finishes performing the selected operation, it must call HiLiteMenu to turn off the highlighting of the menu title.

An application can also use the MenuSelect command in keyboard mouse emulation mode by calling it immediately after calling the KeyboardMouse command. In that mode, the Tool Kit tracks the cursor while the user presses cursor keys to move the cursor. The user indicates a menu selection by pressing the RETURN key or by pressing and releasing the mouse button. The user can also press an appropriate command key. Pressing the ESC key terminates the command.

Error Codes:

(none)

MenuKey

Function:

MenuKey finds the menu item that matches a key.

Command Number: 12 ($ØC)

Parameter List:

4 (input, byte) number of parameters

id (output, byte) menu ID, Ø if no item selected

in (output, byte) item number, undefined if id = Ø

k (input, byte) key: the character typed

km (input, byte) key modifier, as returned by GetEvent:
Ø = no modifier,
1 = OPEN-APPLE key
2 = SOLID-APPLE key
3 = either Apple key

Description:

After the user presses a key, MenuKey searches the menu data to find a menu item that has a matching key. If it finds a match, it highlights the menu title and returns the menu ID number and item number the same way MenuSelect does.

Also like MenuSelect, MenuKey leaves the selected menu highlighted; the progam must call HiLiteMenu to turn off the highlighting.

If you set the key modifier parameter to 3, either Apple key will serve to modify a matching keypress.

If an item is disabled, its menu key or keys will not select it.

As a special case, the MenuKey command can operate like MenuSelect does in keyboard mouse emulation mode. Calling MenuKey with ESC as the key initiates that mode of operation. The Tool Kit tracks the cursor while the user presses cursor keys to move the cursor. The user indicates his selection by pressing the RETURN key or pressing and releasing the mouse button. The user can also press an appropriate command key. Pressing the ESC key terminates the command.

Error Codes:

(none)

HiLiteMenu

Function:

HiLiteMenu turns highlighting of a menu title on or off.

Command Number: 13 ($ØD)

Parameter List:

1 (input, byte) number of parameters

id (input, byte) menu ID: Ø = turn highlighting off

Description:

HiLiteMenu turns highlighting of a specified menu title in the Menu Bar on. Call HiLiteMenu with id = Ø to turn off highlighting after a call to MenuSelect or MenuKey.

Error Codes:

8 ($Ø8) Menu ID was not found.

DisableMenu

Function:

DisableMenu disables or enables selection and highlighting over a whole menu.

Command Number: 14 ($ØE)

Parameter List:

2 (input, byte) number of parameters

id (input, byte) menu ID

dis (input, byte) disable:
1 = disable
Ø = enable

Description:

DisableMenu disables or enables selection and highlighting over a whole menu. If the menu has been disabled, none of the items can be selected, either by MenuSelect or by MenuKey. The menu will still appear when the user moves the mouse onto the menu title, but the title will not be highlighted, and none of the items in the menu will be highlighted when the mouse moves onto them.

When a call to DisableMenu enables a menu, any items that were individually disabled remain disabled. (See the DisableItem command.)

By setting the Disable Flag in the Menu Block's Option Byte when you set up the Menu Bar data structure, your program can make the menu start out disabled. After that, the program should use the DisableMenu command to disable and enable menus.

Error Codes:

8 ($Ø8) Menu ID was not found

DisableItem

Function:

DisableItem disables or enables selection and highlighting of a menu item.

Command Number: 15 ($ØF)

Parameter List:

3 (input, byte) number of parameters

id (input, byte) menu ID

in (input, byte) item number

dis (input, byte) disable:
1 = disable item
Ø = enable item

Description:

DisableItem disables or enables selection and highlighting of a menu item. If an item is disabled, it cannot be selected, either by MenuSelect or by MenuKey, and it will not be highlighted when the mouse moves onto it.

To enable an item, call DisableItem with the disable parameter set to Ø.

By setting the Disable Flag in the Menu Item Block's Item Option Byte when you set up the menu data structure, your program can make the menu item start out disabled. After that, the program should use the DisableItem command to disable and enable menu items.

Calling DisableItem with item number set to zero generates error 9, Item Number Not Valid.

Error Codes:

8 ($Ø8) Menu ID was not found

9 ($Ø9) Item Number not valid

CheckItem

Function:

CheckItem turns the checkmark displayed next to item on or off.

Command Number: 16 ($1Ø)

Parameter List:

3 (input, byte) number of parameters

id (input, byte) menu ID

in (input, byte) item number

ck (input, byte) checkmark:
Ø = turn checkmark off
1 = turn checkmark on

Description:

CheckItem turns the checkmark displayed next to item on or off. The checkmark appears in the blank column in the left edge of the menu.

Your program can call the SetMark command to change the checkmark to any ASCII character.

Calling CheckItem with item number set to zero generates error 9, Item Number Not Valid.

Error Codes:

8 ($Ø8) Menu ID was not found

9 ($Ø9) Item Number not valid

SetMark

Function:

SetMark enables a program to select the character to display for items that are checked in a menu.

Command Number: 2Ø ($14)

Parameter List:

4 (input, byte) number of parameters

id (input, byte) menu ID

in (input, byte) item number

mk (input, byte) checkmark:
Ø = use checkmark character
1 = install new character

char (input, byte) character to display for this item

Description:

SetMark sets the character that is displayed when a program calls CheckItem. The default character is the checkmark.

Error Codes:

8 ($Ø8) Menu ID was not found

9 ($Ø9) Item Number not valid

Window Commands

The Tool Kit's window management commands provide the functions your application program needs to set up and display windows. Once you have set up the window information structure with OpenWindow, you can use these commands to select a window, bring it to the front of the display, put text into it, drag it, change its size, or close it.

Each open window must have a unique ID number in the range from 1 through 255. An attempt to open a second window with the same ID number as one already open will return an error.

A window ID number of Ø is not valid because FrontWindow returns ID = Ø when no window is open. An attempt to open a window with ID number of Ø will return an error.

With some of the Tool Kit commands, you can use an ID number of Ø to indicate the front window. If there is no front window when you do this, these commands return an error. The commands that interpret ID = Ø to mean the front window are

- CloseWindow
- GetWinPtr
- SelectWindow
- DragWindow
- GrowWindow
- WindowToScreen
- ScreenToWindow
- WinChar
- WinString
- WinText
- WinBlock
- WinOp

Note: The use of ID = Ø to select the front window is only a convenience. You can always use the actual ID number of the front window instead.

InitWindowMgr

Function:

InitWindowMgr initializes the internal list of open windows and establishes an area of memory that will be used to save parts of the display while a window is being dragged or grown.

Command Number: 22 ($16)

Parameter List:

2 (input, byte) number of parameters

ptr (input, word) pointer to reserved memory area

size (input, word) size of reserved memory area, in bytes

Description:

InitWindowMgr resets the pointers to the first and last entries in the internal linked list of open windows and establishes an area of memory that will be used to save parts of the display while a window is being dragged or grown.

During calls to DragWindow and GrowWindow, the Tool Kit must save the part of the display obscured by the outline of the window so that it can be replaced when the window operation is finished. The application program must provide the necessary memory space and reserve it for use by the Tool Kit.

The amount of memory space required is determined by the perimeter of the largest window (the sum of twice the window's width plus twice its length).

Note: This memory area can be the same as the area reserved by InitMenu.

Error Codes:

(none)

OpenWindow

Function:

OpenWindow opens a window by supplying a pointer to the window's Information Data Structure.

Command Number: 23 ($17)

Parameter List:

l (input, byte) number of parameters

ptr (input, word) pointer to Window Information Data Structure

Description:

OpenWindow passes window information to the Tool Kit via a pointer to a Window Information Data Structure, or Winfo Data Structure (see Table 2-6). The Winfo Data Structure must reside at a fixed location in memory while the window is open.

The Window Information Data Structure includes a pointer to a Document Information Data Structure (Dinfo Data Structure) that the Tool Kit uses to obtain the text to display in the window (see Table 2-1Ø). Each call to OpenWindow makes that window the front, or active, window.

OpenWindow forces X and Y position coordinates to valid values. It also forces the Thumb positions to be no greater than the maximums. However, OpenWindow does not check to make sure that window minimums are less than maximums or that current window size is between the maximum and the minimum.

The application program can substitute its own routine for OpenWindow. The program passes the address of its open routine in the Winfo Data Structure in place of the pointer to the Dinfo Data Structure and set bit 7 of the Window Option Byte. The Tool Kit will pass control to the program's routine whenever the contents of the window need to be changed.

Because the user routine is called from within the Tool Kit, it cannot rely on the zero-page locations the Tool Kit uses (currently $ØØ to $18). When the Tool Kit calls the user routine, the register contents are

- accumulator: window ID number
- X register: low byte of Winfo address
- Y register: high byte of Winfo address

The routine can only call the Tool Kit commands whose names start with Win- to update the content region of the window it was requested to update. Any other calls can put the Tool Kit into an unknown state.

Error Codes:

12 ($ØC) A window with the same ID is already open

13 ($ØD) InitWindowMgr buffer too small for this window

14 ($ØE) Bad Winfo--tried to open with ID=Ø, or conflicting maximum and minimum width or length

17 ($11) Error returned by user hook

Table 2-6. Information Structure for a Window

Parameter Function	Parameter Size	Note
Window ID Number (not Ø)	1 byte	
Window Option Byte	1 byte	
Title String Pointer	2 bytes	
Window Position X Coordinate	2 bytes	1,2
Window Position Y Coordinate	2 bytes	1,2
Current Content Width	1 byte	1,3
Current Content Length	1 byte	1,3
Minimum Content Width	1 byte	
Maximum Content Width	1 byte	4
Minimum Content Length	1 byte	
Maximum Content Length	1 byte	4
Document Information Structure Pointer	2 bytes	
Horizontal Control Option Byte	1 byte	
Vertical Control Option Byte	1 byte	
Horizontal Scroll Maximum	1 byte	
Current Horizontal Thumb Position	1 byte	1,5
Vertical Scroll Maximum	1 byte	
Current Vertical Thumb Position	1 byte	1,5
Window Status Byte	1 byte	1
Reserved for Future Use	1 byte	6
Pointer to Next Winfo Structure	2 bytes	6
Reserved for Tool Kit	2 bytes	6
Screen Area Covered	4 bytes	6

1 Program sets initial values, Tool Kit updates these.
2 Initial values determine initial position of window.
3 Initial values determine initial window size.
4 Document width and length determine maximum content width and length.
5 Initial values determine initial position of thumb.
6 Items filled in by Tool Kit.

Table 2-7. Contents of Window Option Byte in Window Information Structure

Bit Number	Bit Function	Notes
7	Document Pointer Function	1
6	Reserved for Future Use	
5	Reserved for Future Use	
4	Reserved for Tool Kit	2
3	Reserved for Tool Kit	2
2	Grow Box is present	3
1	Close Box is present	3
Ø	Window is Dialog or Alert Box	3

1 This bit indicates the function of the Document Pointer.
Ø = Pointer to Document Information Structure
1 = Pointer to User Window Routine
2 The program must set these bytes to Ø.
3 These items set the initial appearance of the window. They cannot be changed when the window is open; instead, you must close the window, change the values, then open the window again.

Table 2-8. Contents of Horizontal or Vertical Control Option Byte in Window Information Structure

Bit Number	Bit Function	Notes
7	Scrollbar is present	1
6	Thumb is present	1
5	Reserved for Future Use	
4	Reserved for Future Use	
3	Reserved for Future Use	
2	Reserved for Future Use	
1	Reserved for Future Use	
Ø	Scrollbar is active	2

1 These items set the initial appearance of the window. They cannot be changed when the window is open; instead, you must close the window, change the values, then open the window again.
2 Initial value set by program; after that, use ActicateCtl to change it.

Table 2-9. Contents of Window Status Byte in Window Information Structure

Bit Number	Bit Function	Note
7	Window is open	*
6	Reserved for Future Use	
5	Reserved for Future Use	
4	Reserved for Future Use	
3	Used by Tool Kit	
2	Used by Tool Kit	
1	Used by Tool Kit	
Ø	Used by Tool Kit	

* Program can read to determine state of window.

Table 2-10. Information Structure for a Document

Parameter Function	Parameter Size	Note
Document Pointer	2 bytes	1
Reserved (set to Ø)	1 byte	
Document Width	1 byte	
Document X Coordinate	2 bytes	2
Document Y Coordinate	2 bytes	2
Reserved for Tool Kit	4 bytes	3

1 See bit 7 of the Window Option Byte
2 Set to Ø or set initial position in the document.
3 The program must set these bytes to Ø.

CloseWindow

Function:

CloseWindow removes the window with given ID number and redisplays the screen.

Command Number: 24 ($18)

Parameter List:

l (input, byte) number of parameters

id (input, byte) ID number of window to close

Description:

CloseWindow removes the window with the given ID number from the list of open windows and redisplays the screen with the window removed. Setting ID = Ø selects the top window as the window to be closed.

Error Codes:

15 ($ØF) Window ID not found

17 ($11) Error returned by user hook

CloseAll

Function:

CloseAll closes all open windows and redisplays the screen.

Command Number: 25 ($19)

Parameter List:

Ø (input, byte) number of parameters

Description:

CloseAll removes all windows from the list of open windows and redisplays the screen.

Error Codes:

(none)

GetWinPtr

Function:

GetWinPtr returns the pointer to the Winfo structure of the open window that has the specified ID number.

Command Number: 45 ($2D)

Parameter List:

2 (input, byte) number of parameters

id (input, byte) ID number of window

ptr (output, word) pointer to Winfo Data Structure

Description:

GetWinPtr returns the pointer to the Window Information Data Structure (Winfo) of the open window that has the specified ID number. Setting ID = Ø selects the top window.

Error Codes:

15 ($ØF) Window ID not found

FindWindow

Function:

FindWindow returns the ID number of the window that contains the given point.

Command Number: 26 ($1A)

Parameter List:

4 (input, byte) number of parameters

px (input, byte) X mouse-coordinate of point

py (input, byte) Y mouse-coordinate of point

type (output, byte) type of area point is in:
Ø = Desktop
1 = Menu Bar
2 = content region
3 = drag region
4 = Grow Box
5 = Close Box

id (output, byte) ID number of window the point is in (Ø if point is in desktop or menu bar).

Description:

FindWindow returns the ID number of the window that contains the given point and returns the type of region the point is in: Menu Bar, content region, drag region, Grow Box, or Close Box. The point is specified in mouse coordinates. If the point is not in a window, FindWindow returns an ID number of Ø and a region type of desktop.

If the point is in the content region, the application program should call the FindControl command with window coordinates of the point to determine whether the point is in a scroll bar.

Error Codes:

(none)

FrontWindow

Function:

FrontWindow returns the ID number of the front window.

Command Number: 27 ($1B)

Parameter List:

l (input, byte) number of parameters

id (output, byte) ID number of front window

Description:

FrontWindow returns the ID number of the front, or active, window. It returns Ø if no windows are open.

Error Codes:

(none)

SelectWindow

Function:

SelectWindow activates the window with the given ID number.

Command Number: 28 ($1C)

Parameter List:

1 (input, byte) number of parameters

id (input, byte) ID number of window

Description:

SelectWindow makes the window with the given ID number the front, or active, window and redisplays the screen. The window that was active becomes the second window in the list. Setting ID = Ø selects the front window. If the window selected is already the front window, the Tool Kit does not redisplay the screen.

Error Codes:

15 ($ØF) Window ID not found

17 ($11) Error returned by user hook

TrackGoAway

Function:

TrackGoAway tracksthe mouse and indicates whether the mouse button was released in the Go-Away Box.

Command Number: 29 ($1D)

Parameter List:

l (input, byte) number of parameters

go (output, byte) Go-Away status:
Ø = not in Go-Away Box
1 = mouse was in Go-Away Box

Description:

TrackGoAway tracks the mouse until the mouse button is released. If the mouse is in the Go-Away Box when the button is released, the return status is 1; if not, it is Ø.

The application program should call TrackGoAway when it detects that the mouse button is down with the mouse in the Go-Away Box of the front window. If the return status indicates that the button was released in the Go-Away Box, the application program should then call the CloseWindow command.

Error Codes:

16 ($1Ø) There are no windows

DragWindow

Function:

DragWindow displays the outline of the window being dragged, then redisplays it in its new position.

Command Number: 3Ø ($1E)

Parameter List:

3 (input, byte) number of parameters

id (input, byte) ID number of window being dragged

mx (input, byte) X mouse coordinate of starting position

my (input, byte) Y mouse coordinate of starting position

Description:

DragWindow displays the outline of the window being dragged until the user releases the mouse button, whereupon DragWindow clears the display area previously occupied by the window and redisplays the windows from back to front.

The application program should call the DragWindow command when it detects that the mouse button is down in the drag region of a window. In addition to the ID number of the window, the DragWindow command also needs the mouse coordinates of the position returned as px and py by the FindWindow command. In this it differs from the TrackGoAway and GrowWindow commands; while the Go-Away Box and the Grow Box consist of only one character each, the drag bar consists of several characters, and the mouse could be in any of them when the user starts dragging the window.

Setting ID = Ø selects the front window.

An application can also use the DragWindow command in keyboard mouse emulation mode by calling it immediately after calling the KeyboardMouse command. In that mode, the Tool Kit tracks the cursor and moves the window outline while the user presses cursor keys. The user indicates the completion of the move by pressing the RETURN key or by pressing and releasing the mouse button. Pressing the ESC key terminates the command and redisplays the window in its original position.

Error Codes:

15 ($ØF) Window ID not found

17 ($11) Error returned by user hook

22 ($16) Operation cannot be performed

GrowWindow

Function:

GrowWindow displays the outline of the window being grown, then redisplays an empty window with the new size.

Command Number: 31 ($1F)

Parameter List:

1 (input, byte) number of parameters

stat (output, byte) return status:
Ø = window did not change size
1 = window did change size

Description:

GrowWindow displays the outline of the window being grown until the user releases the mouse button, whereupon GrowWindow clears the display area previously occupied by the window and redisplays the windows from back to front.

The application program should call the GrowWindow command when it detects that the mouse button is down in the Grow Box of the front window.

GrowWindow leaves the content area of the front window blank because it can't determine whether the bottom of the document has been passed and whether the content area should be shifted. If the return status indicates that GrowWindow changed the size of the window, the application must redisplay the content area and update the scroll bars.

An application can also use the GrowWindow command in keyboard mouse emulation mode by calling it immediately after calling the KeyboardMouse command. In that mode, the Tool Kit tracks the cursor and draws the window outline in diffferent sizes while the user presses cursor keys. The user indicates the completion of the resizing by pressing the RETURN key or by pressing and releasing the mouse button. Pressing the ESC key terminates the command and redisplays the window in its original size.

Error Codes:

16 ($1Ø) There are no windows

17 ($11) Error returned by user hook

22 ($16) Operation cannot be performed

WindowToScreen

Function:

WindowToScreen converts window coordinate values to screen coordinates.

Command Number: 32 ($2Ø)

Parameter List:

5 (input, byte) number of parameters

id (input, byte) ID number of window to use

wx (input, word) X coordinate in the window

wy (input, word) Y coordinate in the window

sx (output, word) X coordinate for the screen

sy (output, word) Y coordinate for the screen

Description:

WindowToScreen converts passed coordinate values from window coordinates to screen coordinates.

Setting ID = Ø selects the front window.

Error Codes:

15 ($ØF) Window ID not found

ScreenToWindow

Function:

ScreenToWindow converts screen coordinate values to window coordinates.

Command Number: 33 ($21)

Parameter List:

5	(input, byte) number of parameters
id	(input, byte) ID number of window to use
sx	(input, word) X coordinate for the screen
sy	(input, word) Y coordinate for the screen
wx	(output, word) X coordinate in the window
wy	(output, word) Y coordinate in the window

Description:

ScreenToWindow converts passed coordinate values from screen coordinates to window coordinates.

Setting ID = Ø selects the front window.

Error Codes:

15 ($ØF) Window ID not found

WinChar

Function:

WinChar writes a character in a window.

Command Number: 34 ($22)

Parameter List:

4 (input, byte) number of parameters

id (input, byte) ID number of window

wx (input, word) X coordinate in window

wy (input, word) Y coordinate in window

char (input, byte) character to display

Description:

WinChar writes a character at a given position in a window. If the position given is not inside the window, WinChar does not write the character.

WinChar does not update the document.

Setting ID = Ø selects the front window.

Error Codes:

15 ($ØF) Window ID not found

WinString

Function:

WinString writes a string in a window.

Command Number: 35 ($23)

Parameter List:

5	(input, byte)	number of parameters
id	(input, byte)	ID number of window
wx	(input, word)	X coordinate in window
wy	(input, word)	Y coordinate in window
ptr	(input, word)	pointer to the string
res	(input, byte)	must be Ø.

Description:

WinString writes a string at a given position in a window. WinString does not wrap around; if the string extends past the right edge of the window, WinString just truncates it. WinString does not display any characters in the string that fall outside the edges of the window.

WinString does not update the document.

Setting ID = Ø selects the front window.

Error Codes:

15 ($ØF) Window ID not found

WinText

Function:

WinText writes ASCII characters in a window.

Command Number: 38 ($26)

Parameter List:

5 (input, byte) number of parameters

id (input, byte) ID number of window

wx (input, word) X coordinate in window

wy (input, word) Y coordinate in window

ptr (input, word) pointer to the first character of text

len (input, byte) number of characters to display

Description:

WinText writes ASCII characters at a given position in a window. WinText does not wrap around; if the characters extend past the right edge of the window, WinText just truncates them. WinText does not display any characters that fall outside the edges of the window.

WinText does not update the document.

Setting ID = Ø selects the front window.

Error Codes:

15 ($ØF) Window ID not found

WinBlock

Function:

WinBlock writes a block of text in a window.

Command Number: 36 ($24)

Parameter List:

6 (input, byte) number of parameters

id (input, byte) ID number of window

ptr (input, word) pointer to Document Information Data Structure for the text to be displayed. If ptr = Ø, WinBlock uses the Dinfo pointer from the Winfo specified by the window ID. (Document Information Data Structure definition is Table 2-9.)

startx (input, word) X coordinate of upper-left corner of display window position within the document window (see Fig. 1-4)

starty (input, word) Y coordinate of upper-left corner of display window position within the document window (see Fig. 1-4)

stopx (input, word) X coordinate of lower-right corner of display window position within the document window (see Fig. 1-4)

stopy (input, word) Y coordinate of lower-right corner of display window position within the document window (see Fig. 1-4)

Description:

WinBlock writes a block of text in a window. Startx, starty, stopx, and stopy define a rectangle in the window where the characters are displayed; WinBlock does not alter anything outside this rectangle.

WinBlock does not update the document.

Setting ID = Ø selects the front window.

Error Codes:

15 ($ØF) Window ID not found

WinOp

Function:

WinOp performs an operation on a window.

Command Number: 37 ($25)

Parameter List:

4 (input, byte) number of parameters

id (input, byte) ID number of the window

wx (input, word) X window coordinate

wy (input, word) Y window coordinate

op (input, byte) operation to perform:

26 ($1A) = clear to start of window*
27 ($1B) = clear to start of line*
28 ($1C) = clear window
29 ($1D) = clear to end of window
3Ø ($1E) = clear line
31 ($1F) = clear to end of line

* Operations do not clear the character at position X,Y.

Description:

WinOp clears all or a portion of a window, depending on the operation code. Except for operation code 28, clear window, WinOp clears the characters from position X,Y to the end of the area indicated by the operation. Notice that the forward clears include the character at position X,Y, but the backward clears—that is, clear to start of window and clear to start of line--do not. You can think of the latter operations as "clear from start of area up to, but not through, position X,Y."

Setting ID = Ø selects the front window.

Error Codes:

15 ($ØF) Window ID not found

Control Region Commands

These commands deal with the control regions in the front window: the horizontal and vertical scroll bars, including the Thumbs.

FindControl

Function:

FindControl indicates which control region of a window a given point is in.

Command Number: 39 ($27)

Parameter List:

4 (input, byte) number of parameters

wx (input, word) X window coordinate of point

wy (input, word) Y window coordinate of point

ctl (output, byte) control region the point is in:
- Ø = content region
- 1 = vertical scroll bar
- 2 = horizontal scroll bar
- 3 = none of the above (dead zone)

part (output, byte) part of control region the point is in:
- 1 = Up-Arrow of vertical scroll bar, Left-Arrow of horizontal scroll bar
- 2 = Down-Arrow of vertical scroll bar, Right-Arrow of horizontal scroll bar
- 3 = page-up region of vertical scroll bar, page-left region of horizontal scroll bar
- 4 = page-down region of vertical scroll bar, page-right region of horizontal scroll bar
- 5 = Thumb of scroll bar

Description:

FindControl indicates which control region of a window a given point is in. The application program should call FindControl when it determines, by means of a call to FindWindow, that the mouse is in the content region of the front window. Depending on the control and part codes returned by FindControl, the application should then take

appropriate action--for example, if the mouse is in a page-up or page-down region or in an Up-Arrow or Down-Arrow, the application scrolls the contents of the window, then calls UpdateThumb to make the Thumb reflect the new position in the file.

The application program must make sure that the wx and wy values are converted to window coordinates before calling FindControl.

Note: This is different from FindWindow, which takes mouse coordinates.

Error Codes:

16 ($1Ø) There are no windows

SetCtlMax

Function:

SetCtlMax changes the range of the scroll bar of the front window.

Command Number: 4Ø ($28)

Parameter List:

2 (input, byte) number of parameters

ctl (input, byte) control region to update max value for:
1 = vertical scroll bar
2 = horizontal scroll bar

max (input, byte) new maximum value (must be greater than 1)

Description:

SetCtlMax changes the range of the scroll bar of the front window. If the current Thumb position is greater than the new maximum, SetCtlMax sets the Thumb to the new maximum and calls UpdateThumb to display it at the proper position. SetCtlMax changes the control max value and (if necessary) Thumb position in the Winfo Data Structure.

The program normally calls SetCtlMax whenever the size of a window changes (for example, by GrowWindow).

Maximum values depend on the application; a typical maximum value for the horizontal scroll bar would be calculated as the document width, minus the content width, plus twice the width of the vertical scroll bar or grow box. Likewise, a typical maximum value for the vertical scroll bar would be calculated as the document length, minus the content length, plus the height of the horizontal scroll bar.

Error Codes:

16 ($1Ø) There are no windows

18 ($12) Bad control ID (not 1 or 2)

TrackThumb

Function:

TrackThumb tracks the thumb until the mouse button is released, then it updates the data in the Winfo.

Command Number: 41 ($29)

Parameter List:

3 (input, byte) number of parameters

ctl (input, byte) the control region whose Thumb is moving:
1 = vertical scroll bar
2 = horizontal scroll bar

pos (output, byte) position the Thumb moved to

stat (output, byte) return status:
Ø = Thumb didn't move, pos is not valid
1 = Thumb did move

Description:

TrackThumb tracks the Thumb until the mouse button is released. The application program should call TrackThumb when FindControl indicates that the mouse button is down in the Thumb. When the mouse button is released, TrackThumb updates the position information in the Winfo Data Structure and returns the new position of the Thumb. If the value of the return status is Ø, the Thumb is in the same position it started in, and the value of pos is not valid.

The Thumb position is a number in the range from Ø to the maximum position on the horizontal or vertical scrolling bar. A position of Ø means the first character of the document should be made visible; a position equal to the maximum means the last character of the document should be made visible.

If the Thumb position is the same as it was when TrackThumb is called, it is treated as if it had not moved. If the Thumb does move, TrackThumb updates the Thumb position in the Winfo Data Structure.

TrackThumb operates only on the front window.

Error Codes:

16 ($1Ø) There are no windows

18 ($12) Bad control ID (not 1 or 2)

UpdateThumb

Function:

UpdateThumb redisplays the Thumb in the designated position.

Command Number: 42 ($2A)

Parameter List:

2 (input, byte) number of parameters

ctl (input, byte) control region whose Thumb is being moved

pos (input, byte) new position of Thumb

Description:

UpdateThumb redisplays the Thumb in the designated position and updates the position value in the Winfo Data Structure. UpdateThumb operates only on the front window.

The program should call UpdateThumb after scrolling or paging.

Error Codes:

16 ($1Ø) There are no windows

18 ($12) Bad control ID (not 1 or 2)

ActivateCtl

Function:

ActivateCtl changes the state of a scroll bar.

Command Number: 43 ($2B)

Parameter List:

2 (input, byte) number of parameters

ctl (input, byte) which control region to change

state (input, byte) state to make control region:
Ø = inactive
1 = active

Description:

ActivateCtl changes the state of a scroll bar and updates the Control Option Byte in the Winfo Data Structure. An active scroll bar shows the Thumb and page regions; an inactive bar shows a hollow page region.

ActivateCtl operates only on the front window.

Error Codes:

16 ($1Ø) There are no windows

18 ($12) Bad control ID (not 1 or 2)

Chapter 3

The Machine Language Interface

This chapter tells you what you need to know to use the MouseText Tool Kit with your machine language programs. For descriptions of the individual Tool Kit commands, see Chapter 2.

Installing the Machine Language Tool Kit

The MouseText Tool Kit is a relocatable package of machine language subroutines. To use the Tool Kit in a Pascal environment, you'll need to load the routines as a library unit, as described in Chapter 4. To use the Tool Kit in a ProDOS environment, you'll need to load the routines with the relocating loader that comes with the ProDOS Assembly Tools.

By The Way: Make sure you get up-to-date documentation and code for Rload; the early versions don't work.

Version 2.1 of the MouseText Tool Kit is designed to run only on the Apple IIe and only in the primary 64K memory space; later versions may be able to take advantage of auxiliary memory.

Important: Version 2.1 of the MouseText Tool Kit must reside in the main 64K memory bank, and all calls must be made from main memory.

All calls to the MouseText Tool Kit go through a single entry point named ToolKit. In addition to necessary housekeeping functions, the

main entry point of the MouseText Tool Kit saves the X and Y index registers and saves the locations in zero page that it uses for temporary storage.

The exit routine for the Tool Kit also performs housekeeping functions, as well as restoring the contents of the zero-page locations, restoring the previous contents of the X and Y index registers, and setting the carry flag to reflect the error status. The exit routine also loads the error status into the accumulator, thereby setting the 65Ø2's N and Z flags.

Syntax of Machine Language Calls

A machine language call to the MouseText Tool Kit looks like this:

```
JSR     TOOLKIT         ;main Tool Kit entry point
DB      CMDNUM          ;command number of
                        ;routine being called
DW      CMDLIST         ;pointer to parameter list
BNE     ERROR           ;optional error handling
```

For programming examples showing calls to the MouseText Tool Kit, refer to Appendix D.

By The Way: Calls to the MouseText Tool Kit have the same syntax as calls to the ProDOS Machine Language Interface, which is described in the ProDOS Technical Reference Manual.

After a return from a call to the Tool Kit, the value of the program counter is six bytes beyond the location of the calling JSR, and the accumulator contains the error code. The index registers and the stack pointer are unchanged. If the called routine generated an error, the carry bit is on and the zero bit is off; if it did not generate an error, the zero bit is on and the carry bit is off. Table 3-1 gives a summary of the return status for Tool Kit calls.

Table 3-1. Processor Status After Return from Tool Kit. A bit value of x means the bit is undefined.

	Processor Status Bits N	Z	C	D	V	Accumulator Contents	Program Counter
Successful Call	Ø	1	Ø	Ø	x	Ø	Calling JSR + 6
Unsuccessful Call	Ø	Ø	1	Ø	x	Error Code	Calling JSR + 6

The Machine language Commands

Here are the command numbers and parameter lists for each of the Tool Kit commands.

Startup Commands

A program normally calls appropriate startup commands once to set up its operating environment.

StartDeskTop

```
StartDesktop  equ   Ø        ; command number

start.parms   db    6        ; parameter list for StartDeskTop
start.mid     db    Ø        ; machine id byte
start.msid    db    Ø        ; machine subid byte
start.opsys   db    $ØØ      ; using ProDOS
start.slotn   db    $ØØ      ; slot no. for mouse (Ø = check all slots)
start.int     db    $Ø1      ; using Interrupt Mode
start.col     db    $Ø1      ; using 8Ø columns
```

StopDesktop

```
StopDesktop   equ   1        ; command number

stop.parms    db    Ø        ; parameter list for StopDeskTop
```

PascIntAdr

```
PascIntAdr    equ    17        ; command number

pasc.parms    db      1        ; parameter list for PascIntAdr
pasc.addr     dw      Ø        ; address of int handler
```

SetBasAdr

```
SetBasAdr     equ    18        ; command number

setb.parms    db      1        ; parameter list for SetBasAdr
setb.addr     dw      Ø        ; base address of string area
```

Version

```
Version       equ    19        ; command number

ver.parms     db     2         ; parameter list for Version
ver.ver       db     Ø         ; version number
ver.rev       db     Ø         ; revision number
```

SetUserHook

```
SetUserHook   equ    47        ; command number

shook.parms   db     2         ; parameter list for SetUserHook
shook.id      db     Ø         ; user's routine ID
shook.addr    db     Ø         ; starting address of user's routine
```

KeyboardMouse

```
KeyboardMouse equ    48        ; command number

kdbms.parms   db     Ø         ; parameter list for KeyboardMouse
```

Cursor Commands

These commands control the appearance of the cursor.

SetCursor

```
SetCursor     equ    2         ; command number

setc.parms    db     1         ; parameter list for SetCursor
setc.char     db     $ØØ       ; character to use for cursor
```

ShowCursor

```
ShowCursor    equ    3          ; command number

showc.parms   db     Ø          ; parameter list for ShowCursor
```

HideCursor

```
HideCursor    equ    4          ; command number

hidec.parms   db     Ø          ; parameter list for HideCursor
```

ObscureCursor

```
ObscureCursor equ    44         ; command number

obscc.parms   db     Ø          ; parameter list for ObscureCursor
```

Event-Handling Commands

These commands deal with events in the event queue.

CheckEvents

```
CheckEvents   equ    5          ; command number

chke.parms    db     Ø          ; parameter list for CheckEvents
```

GetEvent

```
GetEvent      equ    6          ; command number

evt.parms     db     3          ; parameter list for GetEvent
evt.type      db     Ø          ; the event type
evt.eb1       db     Ø          ; event byte 1 (x or key)
evt.eb2       db     Ø          ; event byte 2 (y or modifier)

evt.x         equ    evt.eb1    ; x pos of mouse
evt.y         equ    evt.eb2    ; y pos of mouse
evt.key       equ    evt.eb1    ; key input by user
evt.keymod    equ    evt.eb2    ; modifier to key input by user
```

FlushEvents

```
FlushEvents    equu    7          ; command number

flshe.parms    db      Ø          ; parameter list for FlushEvents
```

SetKeyEvent

```
SetKeyEvent    equ     8          ; command number

setkey.parms   db      1          ; parameter list for SetKeyEvent
setkey.sk      db      Ø          ; set key event
```

PeekEvent

```
PeekEvent      equ     21         ; command number

pke.parms      db      3          ; parameter list for PeekEvent
pke.type       db      Ø          ; the event type
pke.eb1        db      Ø          ; event byte 1 (x or key)
pke.eb2        db      Ø          ; event byte 2 (y or modifier)

pke.x          equ     pke.eb1    ; x pos of mouse
pke.y          equ     pke.eb2    ; y pos of mouse
pke.key        equ     pke.eb1    ; key input by user
pke.keymod     equ     pke.eb2    ; modifier to key input by user
```

PostEvent

```
PostEvent      equ     46         ; command number

post.parms     db      3          ; parameter list for PostEvent
post.type      db      Ø          ; the event type
post.eb1       db      Ø          ; event byte 1 (x or key)
post.eb2       db      Ø          ; event byte 2 (y or modifier)

post.x         equ     post.eb1   ; x pos of mouse
post.y         equ     post.eb2   ; y pos of mouse
post.key       equ     post.eb1   ; key input by user
post.keymod    equ     post.eb2   ; modifier to key input by user
```

Menu Commands

These commands deal with menu selection and display.

InitMenu

```
InitMenu      equ    9          ; command number

im.parms      db     2          ; parameter list for InitMenu
im.sarea      dw     savearea   ; area to use for saving screen under menu
im.ssize      dw     savesize   ; size of save area
```

SetMenu

```
SetMenu       equ    1Ø         ; command number

sm.parms      db     1          ; parameter list for SetMenu
sm.mbar       dw     mymenu     ; pointer to Menu Data Structure
```

MenuSelect

```
MenuSelect    equ    11         ; command number

ms.parms      db     2          ; parameter list for MenuSelect
ms.mid        db     Ø          ; menu ID returned
ms.item       db     Ø          ; item number returned
```

MenuKey

```
MenuKey       equ    12         ; command number

mkey.parms    db     4          ; parameter list for MenuKey
mkey.mid      db     Ø          ; menu ID returned
mkey.item     db     Ø          ; item number returned
mkey.key      db     Ø          ; key user typed
mkey.mod      db     Ø          ; modifier of key
```

HiliteMenu

```
HiliteMenu    equ    13         ; command number

hili.parms    db     1          ; parameter list for HiliteMenu
hili.mid      db     Ø          ; menu ID (Ø for all)
```

DisableMenu

```
DisableMenu   equ    14         ; command number

dism.parms    db     2          ; parameter list for DisableMenu
dism.id       db     Ø          ; menu ID
dism.dis      db     Ø          ; disable code
```

DisableItem

```
DisableItem   equ    15          ; command number

ditm.parms    db     3           ; parameter list for DisableItem
ditm.id       db     Ø           ; menu ID
ditm.item     db     Ø           ; item number
ditm.dis      db     Ø           ; disable code
```

CheckItem

```
CheckItem     equ    16          ; command number

chki.parms    db     3           ; parameter list for CheckItem
chki.id       db     Ø           ; menu ID
chki.item     db     Ø           ; item number
chki.chk      db     Ø           ; checkmark code
```

SetMark

```
SetMark       equ    2Ø          ; command number

setm.parms    db     4           ; parameter list for SetMark
setm.id       db     Ø           ; menu ID
setm.item     db     Ø           ; item number
setm.chk      db     Ø           ; checkmark code
setm.char     db     Ø           ; character to use as checkmark
```

Window Commands

These commands deal with window selection and display.

InitWindowMgr

```
InitWindowMgr equ    22          ; command number

iwm.parms     db     2           ; parameter list for InitWindowMgr
iwm.sarea     dw     savearea    ; area to use when saving window screen
iwm.ssize     dw     savesize    ; size of save area
```

OpenWindow

```
OpenWindow    equ    23          ; command number

open.parm     db     1           ; parameter list for OpenWindow
open.wind     dw     Ø           ; pointer to Winfo Data Structure
```

CloseWindow

```
CloseWindow   equ   24      ; command number

cw.parms      db    1       ; parameter list for CloseWindow
cw.id         db    Ø       ; ID number of window to close
```

CloseAll

```
CloseAll      equ   25      ; command number

cla.parms     db    Ø       ; parameter list for CloseAll
```

GetWinPtr

```
GetWinPtr     equ   45      ; command number

gwip.parms    db    2       ; parameter list for GetWinPtr
gwip.id       db    Ø       ; window ID number
gwip.winfo    dw    Ø       ; pointer to Winfo Data Structure
```

FindWindow

```
FindWindow    equ   26      ; command number

fdw.parms     db    4       ; parameter list for FindWindow
fdw.x         db    Ø       ; X coordinate of mouse
fdw.y         db    Ø       ; Y coordinate of mouse
fdw.type      db    Ø       ; type of region mouse is in
fdw.window    db    Ø       ; window ID number (Ø = desktop)
```

FrontWindow

```
FrontWindow   equ   27      ; command number

frtw.parms    db    1       ; parameter list for FrontWindow
frtw.id       db    Ø       ; ID number of front window
```

SelectWindow

```
SelectWindow  equ   28      ; command number

selw.parms    db    1       ; parameter list for SelectWindow
selw.id       db    Ø       ; ID number of window
```

TrackGoAway

```
TrackGoAway    equ    29        ; command number

tga.parms      db     1         ; parameter list for TrackGoAway
tga.closeit    db     Ø         ; Go-Away status
```

DragWindow

```
DragWindow     equ    3Ø        ; command number

dg.parms       db     3         ; parameter list for DragWindow
dg.id          db     Ø         ; window ID number
dg.x           db     Ø         ; x mouse coord of cursor start
dg.y           db     Ø         ; y mouse coord of cursor start
```

GrowWindow

```
GrowWindow     equ    31        ; command number

grow.parms     db     1         ; parameter list for GrowWindow
grow.result    db     Ø         ; return status
```

WindowToScreen

```
WindowToScreen equ    32        ; command number

w2s.parms      db     5         ; parameter list for WindowToScreen
w2s.id         db     Ø         ; window ID number
w2s.wx         dw     Ø         ; X coordinate in window
w2s.wy         dw     Ø         ; Y coordinate in window
w2s.sx         dw     Ø         ; X screen coordinate
w2s.sy         dw     Ø         ; Y screen coordinate
```

ScreenToWindow

```
ScreenToWindow equ    33        ; command number

s2w.parms      db     5         ; parameter list for ScreenToWindow
s2w.id         db     Ø         ; window ID number
s2w.sx         dw     Ø         ; X screen coordinate
s2w.sy         dw     Ø         ; Y screen coordinate
s2w.wx         dw     Ø         ; X coordinate in window
s2w.wy         dw     Ø         ; Y coordinate in window
```

WinChar

```
WinChar       equ    34        ; command number

wch.parms     db     4         ; parameter list for WinChar
wch.id        db     Ø         ; window ID number
wch.wx        dw     Ø         ; X coordinate in window
wch.wy        dw     Ø         ; Y coordinate in window
wch.char      db     $ØØ       ; ASCII character to display
```

WinString

```
WinString     equ    35        ; command number

wstr.parms    db     5         ; parameter list for WinString
wstr.id       db     Ø         ; window ID number
wstr.wx       dw     Ø         ; X coordinate in window
wstr.wy       dw     Ø         ; Y coordinate in window
wstr.ptr      dw     Ø         ; pointer to string
wstr.res      db     Ø         ; reserved (for BASIC only)
```

WinText

```
WinText       equ    38        ; command number

wtxt.parms    db     5         ; parameter list for WinString
wtxt.id       db     Ø         ; window ID number
wtxt.wx       dw     Ø         ; X coordinate in window
wtxt.wy       dw     Ø         ; Y coordinate in window
wtxt.ptr      dw     Ø         ; pointer to first character
wtxt.len      db     Ø         ; number of characters
```

WinBlock

```
WinBlock      equ    36        ; command number

wblk.parms    db     6         ; parameter list for WinBlock
wblk.id       db     Ø         ; window ID number
wblk.ptr      dw     Ø         ; pointer to Dinfo Data Structure
wblk.x1       dw     Ø         ; X upper-left window coordinate
wblk.y1       dw     Ø         ; Y upper-left window coordinate
wblk.x2       dw     Ø         ; X lower-right window coordinate
wblk.y2       dw     Ø         ; Y lower-right window coordinate
```

WinOp

```
WinOp          equ    37          ; command number

wop.parms      db     4           ; parameter list for WinBlock
wop.id         db     Ø           ; window ID number
wop.wx         dw     Ø           ; X window coordinate
wop.wy         dw     Ø           ; Y window coordinate
wop.op         db     Ø           ; window operation
```

Control Region Commands

These commands deal with the control regions in the front window: the horizontal and vertical scrolls bars, including the Thumbs.

FindControl

```
FindControl    equ    39          ; command number

findc.parms    db     4           ; parameter list for FindControl
findc.wx       dw     Ø           ; X window coordinate of point
findc.wy       dw     Ø           ; Y window coordinate of point
findc.ctl      db     Ø           ; control region point is in
findc.part     db     Ø           ; part of region point is in
```

SetCtlMax

```
SetCtlMax      equ    4Ø          ; command number

setct.parms    db     2           ; parameter list for SetCtlMax
setct.ctl      db     Ø           ; control region affected
setct.newmax   db     Ø           ; new maximum value
```

TrackThumb

```
TrackThumb     equ    41          ; command number

tkthmb.parms   db     3           ; parameter list for TrackThumb
tkthmb.ctl     db     Ø           ; control region affected
tkthmb.pos     db     Ø           ; position Thumb moved to
tkthmb.moved   db     Ø           ; Thumb moved code
```

UpdateThumb

```
UpdateThumb    equ    42          ; command number

upt.parms      db     2           ; parameter list for UpdateThumb
```

```
upt.ctl       db    Ø        ; control region affected
upt.newpos    db    Ø        ; new position of Thumb
```

ActivateCtl

```
ActivateCtl   equ   43       ; command number

actl.parms    db    2        ; parameter list for ActivateCtl
actl.ctl      db    Ø        ; ctl region to change
actl.inact    db    Ø        ; inactivate code
```

Chapter 4

The Pascal Interface

The Pascal Interface for the MouseText Tool Kit is a Pascal intrinsic unit that provides the interface to the MouseText Tool Kit, Version 2.1. Each of the Tool Kit commands described in Chapter 2 is supported by one of the Pascal Interface procedures described in this chapter. In addition to the command procedures, there is a utility procedure for obtaining the address of a Pascal variable.

Installing the Pascal Interface

The Pascal Interface and the Tool Kit routines are supplied together in a linked object file named MTXKIT.CODE. To use the Tool Kit, you must install MTXKIT.CODE as a Unit in your System.library file. With the Tool Kit code in the system library, the application program can use the Tool Kit commands by including the statement "Uses MTXKIT;" after the heading.

Data Structures

This chapter presents the specifications of the data types and data structures used in the Pascal Tool Kit, including the Menu Data Structure, the Window Information Data Structure, and the Document Information Data Structure, as defined in Chapter 2.

Constants and Type Definitions

The following constants and data types are used in the Pascal Interface.

Constants

max_menus= 1Ø (A maximum of 1Ø menus is supported).
max_title_str= 2Ø (A maximum of 2Ø characters per menu title is supported).
max_item_str= 3Ø (A maximum of 3Ø characters per menu item name is supported).
max_num_items= 1Ø (A maximum of 1Ø menu items is supported).

The following event type values are provided as constants rather than as an enumerated type so that the user can define and handle his own events.

```
no_event = Ø
button_down = 1
button_up = 2
key_down = 3
drag = 4
apple_key = 5
```

A single byte value is defined as:

```
byte = Ø..255;
```

Event

An event is defined as:

```
type_event = packed record
      evt_kind : byte;
      char1 : byte;
      char2 : byte;
      reserve1 : byte;
      end;
```

where:

evt_kind is the event type value (see above under Constants).
char1 is event byte 1, X coordinate or key value.
char2 is event byte 2, Y coordinate or key modifier.
reserve1 is reserved for use by the Tool Kit.

Menu titles are defined as:

```
title_str = string[max_title_str];
```

Menu Item Names

Menu item names are defined as:

```
item_str = string[max_item_str];
```

Menu Item Blocks

A Menu item block is defined as:

```
menu_item = packed record

      open_apple : boolean;        {bit Ø}
      solid_apple : boolean;
      item_has_mark : boolean;
      reserve2 : boolean;
      reserve3 : boolean;
      item_is_checked : boolean;
      item_is_filler : boolean;
      disable_flag : boolean;      {bit 7}

      mark_char : byte;

      char1 : byte;
      char2 : byte;

      item_str_ptr : ^item_str;

      end;
```

where:

The first 8 fields in the record are the bits in the Item Option Byte:

open_apple is on when the modifier is OPEN-APPLE key;
solid_apple is on when the modifier is SOLID-APPLE key;
item_has_mark is on when the item has mark;
reserve2, reserve3 are reserved for use by the Tool Kit;
item_is_checked is on when the Item Is Checked;
item_is_filler is on when the Item Is Filler;
disable_flag is the Disable Flag;

mark_char is the mark character;

char1 is Character 1;
char2 is Character 2;

item_str_ptr is Pointer to Item String;

Menu Data Structures

The Data Structure for a Menu is defined as:

```
menu_data = packed record
      num_items : byte;
      reserve1 : byte;
      reserve2 : byte;
      reserve3 : byte;
      items : packed array [1..max_num_items] of menu_item;
      end;
```

where:

num_items is the Number of Items;
reserve1, reserve2, reserve3 are reserved for use by the Tool Kit;
items is the array of Menu Item Blocks;

Menu Title Blocks

A Menu Title Block is defined as:

```
menu_title = packed record
      menu_id : byte;
      disabled : byte;
      title_ptr : ^title_str;
      m_data_ptr : ^menu_data;
      reserved : packed array [1..4] of byte;
      end;
```

where:

menu_id is the Menu ID;
disabled is the Disable Flag (only bit 7 can be used);
title_ptr is the Pointer to Title String;
m_data_ptr is the Pointer to Menu Data Structure;
reserved is reserved by the Tool Kit;

Menu Bars

The menu bar is defined as:

```
menu_bar = packed record
      num_menus : byte;
      reserved : byte;
      menus : array [1..max_menus] of menu_title;
      end;
```

where:

num_menus is the Number of Menus;
reserved is reserved for use by the Tool Kit;
menus is the array of Menu Blocks;

Window Information Data Structures

A Window Information Data Structure (Winfo) is defined as:

```
winfo = packed record
      window_id: byte;

      dialog: boolean;                {bit Ø}
      goawaybox: boolean;
      growbox: boolean;
      reserve1: boolean;
      reserve2: boolean;
      reserve3: boolean;
      reserve4: boolean;
      dinfo_or_user: boolean;         {bit 7}

      title_ptr: ^title_str;

      windowx: integer;
      windowy: integer;

      contwidth: byte;
      contlength: byte;

      mincontwidth: byte;
      maxcontwidth: byte;
      mincontlength: byte;
      maxcontlength: byte;

      dinfo_ptr: ^dinfo;

      hactive: boolean;             {bit Ø}
      reserve6: boolean;
      reserve7: boolean;
      reserve8: boolean;
      reserve9: boolean;
      reserv1Ø: boolean;
      hthumb: boolean;
      hscrollbar: boolean;          {bit 7}

      vactive: boolean;             {bit Ø}
      reserv11: boolean;
      reserv12: boolean;
      reserv13: boolean;
```

```
      reserv14: boolean;
      reserv15: boolean;
      vthumb: boolean;
      vscrollbar: boolean;          {bit 7}

      hthumbmax: byte;
      hthumbpos: byte;
      vthumbmax: byte;
      vthumbpos: byte;

      reserv16: boolean;
      reserv17: boolean;
      reserv18: boolean;
      reserv19: boolean;
      reserv20: boolean;
      reserv21: boolean;
      reserv22: boolean;
      win_open: boolean;

      reserv23: byte;

      nextwinfo = ^winfo

      reserv24: byte;
      reserv25: byte;
      reserv26: byte;
      reserv27: byte;
      reserv28: byte;
      reserv29: byte;

      end;
```

where:

window_id is the Window ID#

dialog is dialog/alert window flag
goawaybox is on when Go-Away Box present
growbox is on when Grow Box present
reserve1, reserve2, reserve3, reserve4
      are all reserved by the Tool Kit

dinfo_or_user is user routine adr/dinfo ptr

title_ptr is Title Str ptr

windowx is Window Location X
windowy is Window Location Y

contwidth is Current Content Width
contlength is Current Content Length

mincontwidth is Min Content Width
maxcontwidth is Max Content Width
mincontlength is Min Content Length
maxcontlength is Max Content Length

dinfo_ptr is Dinfo Ptr

hactive is on when horizontal scrollbar active
reserve6, reserve7, reserve8, reserve9, reserv1Ø
are all reserved by the Tool Kit
hthumb is on when horizontal Thumb present
hscrollbar is on when horizontal scroll bar present

vactive is on when vertical scroll bar active
reserv11, reserv12, reserv13, reserv14, reserv15
are all reserved by the Tool Kit
vthumb is on when vertical Thumb present
vscrollbar is on when vertical scroll bar present

hthumbmax is horizontal scroll maximum
hthumbpos is current horizontal Thumb position
vthumbmax is vertical scroll maximum
vthumbpos is current vertical Thumb position

reserv16, reserv17, reserv18, reserv19, reserv2Ø,
reserv21, and reserv22
are all reserved by the Tool Kit

win_open is window open

reserv23 is reserved by the Tool Kit

nextwinfo is the pointer to the next winfo structure

reserv24, reserv25, reserv26, reserv27
reserv28, and reserv29
are all reserved by the Tool Kit

Document Information Data Structures

A Document Information Data Structure (Dinfo) is defined as:

```
dinfo = packed record

      doc_ptr: integer;

      reserved: byte;
      docwidth: byte;

      docx: integer;
      docy: integer;
```

```
      doclength: integer;
      reserve2: byte;
      reserve3: byte;

      end;
```

where:

doc_ptr is Document ptr

reserved is reserved by the Tool Kit
docwidth is Document Width

docx is Document X
docy is Document Y

doclength is Document Length
reserve2, reserve3 are reserved by the Tool Kit

Screen Region Types

The type of screen region is defined as:

```
type_area = (inDeskTop,
      inMenubar,
      inContent,
      inDrag,
      inGrow,
      inGoAway);
```

where each value is as returned by FindWindow:

inDeskTop is in desktop
inMenubar is in menu bar
inContent is in contentregion
inDrag is in drag region
inGrow is in Grow Box
inGoAway is in Go-Away Box

Control Region Types

The type of control region is defined as:

```
ctlarea = ( notctl,
      ver_scroll,
      hor_scroll,
      deadzone );
```

where each value is as returned by FindControl:

notctl is in content region
ver_scroll is in vertical scroll bar
hor_scroll is in horizontal scroll bar
deadzone is none of the above

Control Region Part Types

The type of a part of a control region is defined as:

```
ctlpart = ( ctlinactive,
      scrollupleft,
      scrolldownright,
      pageupleft,
      pagedownright,
      thumb );
```

where each value is as returned by FindControl:

ctlinactive is never returned
scrollupleft is up arrow of vertical scroll bar
or Left-Arrow of horizontal scroll bar
scrolldownright is Down-Arrow of vertical scroll bar
or Right-Arrow of horizontal scroll bar
pageupleft is "page up" region of vertical scroll bar
or "page left" region of horizontal scroll bar
pagedownright is "page down" region of vertical scroll bar
or "page right" region of horizontal scroll bar
thumb is Thumb of scroll bar

Pointers

A general purpose pointer is provided and defined as:

```
pointer: integer;
```

Error Codes

The Mouse Tool Kit error code is defined as:

```
TKError : integer;
```

Command Functions and Procedures

Here are the specifications of the procedure calls in the Pascal Tool Kit Interface.

Startup Commands

A program will normally call the appropriate startup commands once to set up the operating environment. The proper sequence of steps to start the mouse is:

(1) Call PascIntAdr to get the address of the Tool Kit's interrupt handler.

(2) Pass the interrupt address to the mouse firmware by calling SetMouse as described in Appendix B, "The Mouse Firmware Interface." Mouse Mode should be set to passive.

(3) Call StartDesktop with the UseInterrupts parameter set the way you want it for your program.

(4) (optional) Call SetUserHook to pass the addresses of your program's interrupt handlers, if any, to the Tool Kit.

StartDeskTop

```
Procedure StartDeskTop ( mach_id :  integer; sub_id:  integer;
      var slot_num :  integer; use_interrupts :  boolean;
      column_8Ø :  boolean );
```

mach_id is the machine ID number.
sub_id is the subsidiary ID number.
slot_num is the slot number of the mouse card.
use_interrupts is the interrupt usage parameter:
    false= Passive Mode only
    true= use interrupts
column_8Ø is the col (number of text columns) parameter:
    false= 4Ø columns
    true= 8Ø columns

StopDeskTop

```
Procedure StopDeskTop;
```

PascIntAdr

Procedure PascIntAdr (var IntAdr: integer);

IntAdr is the address of the interrupt routine

SetUserHook

Procedure SetUserHook (hook_id, hook_adr: integer);

hook_id is the ID number (Ø or 1) for the program's interrupt routine.
hook_adr is the address of the program's interrupt routine.

Version

Procedure Version (var ver_num, rev_num: integer);

ver_num is the version number.
rev_num is the revision number.

KeyboardMouse

This function is used with the MenuSelect, DragWindow, and GrowWindow commands only. Calling one of those commands immediately after calling KeyboardMouse causes the command to operate in keyboard mouse emulation mode, where the user can control the cursor motion by means of the keyboard. The KeyboardMouse function has no parameters.

Function KeyboardMouse;

Cursor Commands

These commands control the appearance of the cursor.

SetCursor

Procedure SetCursor (new_ch : integer);

new_ch is the character to use as cursor.

ShowCursor

Procedure ShowCursor;

HideCursor

Procedure HideCursor;

ObscureCursor

Procedure ObscureCursor;

Event Handling Commands

These commands deal with the event queue.

CheckEvents

Procedure CheckEvents;

GetEvent

Procedure GetEvent (var event : type_event);

event is returned with:

evt_kind set to the event type.
char1 set to event byte 1: X coordinate or key value.
char2 set to event byte 2: Y coordinate or key modifier.

PostEvent

Procedure PostEvent (var event : type_event);

event should be supplied with:

evt_kind set to the event type.
char1 set to event byte 1: X coordinate or key value.
char2 set to event byte 2: Y coordinate or key modifier.

FlushEvents

Procedure FlushEvents;

SetKeyEvent

Procedure SetKeyEvent (chk_keyboard : boolean);

chk_keyboard is the sk (set keyevent) parameter:

false= don't check keyboard
true= check the keyboard

PeekEvent

Procedure PeekEvent (var event : type_event);

event is returned with:

evt_kind set to the event type.
char1 set to the event byte 1: X coordinate or key value.
char2 set to the event byte 2: Y coordinate or key modifier.

Menu Commands

These commands handle menu selection and display.

InitMenu

Procedure InitMenu (save_buffer, buf_size : integer);

save_buffer is a pointer to the save area.
buf_size is the save area size.

SetMenu

Procedure SetMenu (var my_menu_bar : menu_bar);

my_menu_bar is the menu bar structure. The procedure obtains the pointer to the structure for you.

MenuSelect

Procedure MenuSelect (var menu_id, menu_choice : integer);

menu_id is the menu ID number.
menu_choice is the menu item number.

MenuKey

Procedure MenuKey (var menu_id, menu_choice : integer;
var key_event : type_event);

menu_id is the menu ID number.
menu_choice is the item number.
key_event is returned with:

```
char1 as the key value
char2 as the key modifier
```

HiliteMenu

```
Procedure HiliteMenu ( menu_id :  integer );
```

menu_id is the menu ID number.

DisableMenu

```
Procedure DisableMenu ( menu_id :  integer; disable :  boolean );
```

menu_id is the menu ID number.
disable is the dis (disable) parameter:
false= enable
true= disable

DisableItem

```
Procedure DisableItem ( menu_id, item_num :  integer;
      disable :  boolean );
```

menu_id is the menu ID number.
item_num is the item number.
disable is the dis (disable) parameter:
false= enable
true= disable

CheckItem

```
Procedure CheckItem ( menu_id, item_num :  integer;
      check :  boolean );
```

menu_id is the menu ID number.
item_num is the item number.
check is the ck (check) parameter:
false= turn checkmark off
true= turn checkmark on

SetMark

```
Procedure SetMark ( menu_id, item_num:  integer; mark_on:  boolean;
      mark_char:  char);
```

menu_id is the menu ID number.
item_num is the menu item number.

mark_on is the mark on parameter.
mark_char is the mark char parameter.

Window Commands

These commands deal with window selection and display.

InitWindowMgr

Procedure InitWindowMgr (drag_buffer, buf_size : integer);

drag_buffer is the pointer to the buffer.
buf_size is the buffer size.

OpenWindow

Procedure OpenWindow (var my_Winfo: winfo);

my_Winfo is the Winfo data structure.

CloseWindow

Procedure CloseWindow (window_id: integer);

window_id is the window ID number.

CloseAll

Procedure CloseAll;

GetWinPtr

Procedure GetWinPtr (window_id: integer; var winfo_ptr: integer);

window_id is the ID number of the window.
winfo_ptr is a pointer to the Winfo data structure.

FindWindow

Procedure FindWindow (pointx, pointy: integer; var area: type_area;
var window_id: integer);

pointx is the X coordinate of the point.
pointy is the Y coordinate of the point.

area is the type_area (region type) parameter.
window_id is the window ID number.

FrontWindow

Procedure FrontWindow (var window_id: integer);

window_id is the window ID number.

SelectWindow

Procedure SelectWindow (window_id: integer);

window_id is the window ID number.

TrackGoAway

Procedure TrackGoAway (var makeitgoaway: boolean);

makeitgoaway is the go away status:
    Ø = not in Go-Away Box
    1 = mouse was in Go-Away Box

DragWindow

Procedure DragWindow (window_id, mousex, mousey: integer);

window_id is the window ID number.
mousex is the mouse X coordinate.
mousey is the mouse Y coordinate.

GrowWindow

Procedure GrowWindow(var makeitgrow: boolean);

makeitgrow is the return status:
    Ø = window did not grow
    1 = window did grow

WindowToScreen

Procedure WindowToScreen (window_id, windowx, windowy: integer;
    var screenx, screeny: integer);

window_id is the window ID number.
windowx is the window X coordinate.

windowy is the window Y coordinate.
screenx is the screen X coordinate.
screeny is the screen Y coordinate.

ScreenToWindow

```
Procedure ScreenToWindow ( window_id, screenx, screeny:  integer; var
      windowx, windowy:  integer );
```

window_id is the window ID number.
screenx is the screen X coordinate.
screeny is the screen Y coordinate.
windowx is the window X coordinate.
windowy is the window Y coordinate.

WinChar

```
Procedure WinChar ( window_id, windowx, windowy:  integer;
      my_char:  char );
```

window_id is the window ID number.
windowx is the window X coordinate.
windowy is the window Y coordinate.
my_char is the character to display.

WinString

```
Procedure WinString ( window_id, windowx, windowy:  integer;
      my_string:  string );
```

window_id is the window ID number.
windowx is the window X coordinate.
windowy is the window Y coordinate.
my_string is the string to write.

WinText

```
Procedure WinText ( window_id, windowx, windowy, text_buffer,
      textlength:  integer );
```

window_id is the window ID number.
windowx is the X coordinate in the window.
windowy is the Y coordinate in the window.
text_buffer is the pointer to the first character of text.
textlength is the number of characters to display.

WinBlock

```
Procedure WinBlock ( window_id:  integer; var my_dinfo:  dinfo;
      startx, starty, stopx, stopy:  integer );
```

window_id is the window ID number.
my_dinfo is the document information structure.
startx is the X coordinate of the upper-left corner.
starty is the Y coordinate of the upper-left corner.
stopx is the X coordinate of the lower-right corner.
stopy is the Y coordinate of the lower-right corner.

WinOp

```
Procedure WinOp ( window_id, windowx, windowy:  integer;
      opcode:  byte);
```

window_id is the window ID number.
windowx is the window X coordinate.
windowy is the window X coordinate.
opcode is the code for the operation to perform.

Control Region Commands

These commands deal with the control regions in the front window: the horizontal and vertical scroll bars, including the Thumbs.

FindControl

```
Procedure FindControl ( windowx, windowy:  integer;
      var whichctl:  ctlarea; var whichpart:  ctlpart );
```

windowx is the window X coordinate.
windowy is the window Y coordinatei.
whichctl is the control region.
whichpart is the part of the control region.

SetCtlMax

```
Procedure SetCtlMax ( whichctl:  ctlarea; newmax:  integer );
```

whichctl is the control region.
newmax is the new maximum value.

TrackThumb

```
Procedure TrackThumb ( whichctl:  ctlarea; var thumbpos:  integer;
      var thumbmoved:  boolean );
```

whichctl is the control region.
thumbpos is the Thumb position.
thumbmoved is the return status:
    Ø = Thumb didn't move, thumbpos not valid
    1 = Thumb did move

UpdateThumb

```
Procedure UpdateThumb ( whichctl:  ctlarea; thumbpos:  integer );
```

whichctl is the control region.
thumbpos is the new Thumb position.

ActivateCtl

```
Procedure ActivateCtl ( whichctl:  ctlarea; makeactive:  boolean );
```

whichctl is the control region.
makeactive is the state to make the control region:
    Ø = inactive
    1 = active

Utility Functions

In addition to a call for each of the Tool Kit commands, there is a utility function for obtaining the address of a Pascal variable.

PointerTo

This function obtains the address of the specified variable and returns it as the function value.

```
Function PointerTo(var Variable) :  integer;
```

Chapter 5

The Applesoft Interface

The Applesoft Interface for the MouseText Tool Kit is a set of commands that are added to the standard Applesoft commands by means of the ampersand hook. So that you can use it with other ampersand packages, the Applesoft Interface saves the existing ampersand hook address and passes any unrecognized ampersand commands on. If there is another ampersand package, that package then gets control and can test the commands. If there is no other ampersand package, then Applesoft gets control and issues a SYNTAX ERROR message.

Installing the Applesoft Interface

You'll need the relocating loader from the ProDOS Assembler Tools to load both the MouseText Tool Kit routines and the Applesoft Interface that contains the ampersand commands. The procedure for loading the Mouse Tool Kit is as follows:

(1) Load the MouseText Tool Kit from file MTXKIT.OBJ using RBOOT.

(2) Load the Applesoft Interface from file MTXAMP.OBJ using RBOOT.

(3) Write the starting address of the Tool Kit in the first two bytes of the Applesoft Interface (--not the other way around, now!).

(4) Start the Applesoft Interface by a CALL to its address plus two.

One way to do this in Applesoft looks like this:

```
1Ø  PRINT  CHR$ (4);"BRUN RELEASE"
2Ø A1 = Ø:A2 = Ø
3Ø  PRINT  CHR$ (4);"BRUN RBOOT"
4Ø A1 = USR (Ø);"MTXKIT.OBJ"
5Ø A2 = USR (Ø),"MTXAMP.OBJ"
6Ø  IF A1 < Ø THEN A1 = A1 + 65536
7Ø  IF A2 < Ø THEN A2 = A2 + 65536
8Ø I =  INT (A1 / 256)
9Ø J = A1 - I * 256
1ØØ  POKE A2,J
11Ø  POKE A2 + 1,I
12Ø  CALL A2 + 2
13Ø  END
```

The MouseText Tool Kit routines are in the file named MTXKIT.OBJ; the Applesoft Tool Kit ampersand routines are in the file named MTXAMP.OBJ.

Using the Ampersand Commands

The Applesoft Tool Kit interface does not treat string variables the same as numeric variables. The interface routines copy numeric variables into internal buffers, so altering the values of those variables after an ampersand command will not change anything that the Tool Kit is doing. String variables, on the other hand, are not copied into buffers, so changing a BASIC string variable will cause changes in the display the next time the Tool Kit redisplays the menu or window with the changed string.

Note: All input parameters can be either variables or expressions, but output parameters must, of course, be variables.

The Ampersand Commands

The Applesoft Interface includes an ampersand call for each of the Tool Kit commands except for a few, such as PascIntAdr, which is used only with Pascal. There are also two utility commands, &STCNTNT (Set Content), and &DSKTPERR (Desktop Error).

The names of the ampersand commands are not the same as the command names listed in the other chapters. This is to avoid having Applesoft tokenize certain letter combinations, thereby altering a program and its listing. You must spell the command names exactly as shown or the Applesoft Interface won't recognize them and Applesoft will give you a SYNTAX ERROR message.

WARNING
A misspelled ampersand command can cause Applesoft to hang if it occurs after a RUN command. When Applesoft fails to recognize the misspelled ampersand command, it jumps back to the RUN statement, thereby getting itself into a loop condition.

By The Way: The variable names shown here are only suggestions; you may use any variable names you choose.

Startup Commands

A program normally calls these commands once to set up its operating environment.

StartDesktop

&STRTDSKTP(ID%,SID%,SN%,IU%,COL%)
ID% = machine ID
SID% = subsidiary ID
SN% = slot number (input and output). If SN% = Ø, StartDeskTop searches for a mouse card and returns the slot number in SN%.
IU% = interrupt usage:
Ø = Passive Mode
1 = Interrupt Mode

COL% = number of text columns:
Ø = 4Ø columns
1 = 8Ø columns

StopDeskTop

In addition to making the appropriate call to the Tool Kit, the Applesoft version of the StopDeskTop command disconnects the Applesoft Interface from the ampersand hook and restores the previous address.

Tool Kit ampersand commands will not work after a call to &STPDSKTP unless you reconnect the Tool Kit by means of the Applesoft command CALL A2+2, where A2 is the starting address of the Tool Kit Applesoft Interface. If you do stop and reconnect, you don't get back any used memory pages; for that, you have to run RELEASE and reload the Tool Kit and the Applesoft Interface.

To make sure that the ampersand hooks get properly restored when the program ends, there must be a call to StopDeskTop. Your program should include an ONERR call to &DSKTPERR, then a call to &STPDSKTP.

&STPDSKTP
(no parameters)

Version

&VRSN(V%,R%,AV%,AR%)
V% = version number
R% = revision number
AV% = Applesoft Interface version number
AR% = Applesoft Interface revision number

KeyboardMouse

The application calls this command immediately before MenuSelect, DragWindow, or GrowWindow to make those commands run in keyboard mouse emulation mode. Note that the KeyboardMouse command has no parameters.

&KYBRDMSE

Cursor Commands

These commands control the appearance of the cursor.

SetCursor

&STCRSR(CC%)
CC% = cursor character (ASCII code)

ShowCursor

&SHWCRSR
(no parameters)

HideCursor

&HDCRSR
(no parameters)

ObscureCursor

&OBCRSR
(no parameters)

Event-Handling Commands

These commands deal with the event queue.

CheckEvents

&CHCKEVNTS
(no parameters)

PostEvent

&PSTEVNT(ET%,E1%,E2%)
ET% = event type (input):
  Ø = no event
  1 = button down
  2 = button up
  3 = key pressed
  4 = drag event
  5 = Apple key down
E1% = X coordinate or key value (input)
E2% = Y coordinate or key modifier (input)

GetEvent

```
&GTEVNT(ET%,E1%,E2%)
ET% = event type:
      Ø = no event
      1 = button down
      2 = button up
      3 = key pressed
      4 = drag event
      5 = Apple key down
E1% = X coordinate or key value
E2% = Y coordinate or key modifier
```

FlushEvents

```
&FLSHEVNTS
(no parameters)
```

SetKeyEvent

```
&STKYEVNT(SK%)
SK% = Setkey flag:
      Ø = don't check keyboard
      1 = check keyboard
```

PeekEvent

```
&PKEVNT(ET%,E1%,E2%)
ET% = event type:
      Ø = no event
      1 = button down
      2 = button up
      3 = key pressed
      4 = drag event
      5 = Apple key down
E1% = x coordinate or key
E2% = y coordinate or key modifier
```

Menu Commands

These commands handle menu selection and display.

InitMenu

The InitMenu command sets aside memory space needed for the Menu Data Structure and for saving the part of the display obscured by menus.

You can determine the amount of memory space to reserve for menu displays by calculating the screen area of the largest menu in the program. The largest menu could have a large screen area because it has many items, or it could have only a few items, each of which is very long.

You calculate the screen area of a menu by taking the product of the number of items in the menu, plus one, times five bytes more than the length of the longest item string in that menu. If you are using keys to select items, each item string must include three bytes to display a space, an Apple key, and the key that selects the item. A page is 256 bytes, so to find the number of pages required, divide the size of the largest menu by 256 and round to the next highest integer.

To calculate the amount of memory to set aside for the Menu Data Structures, in bytes, add fourteen bytes for each menu plus six bytes for each item, plus two. Divide the result by 256 and round to the next highest integer to find the number of pages required. If you don't make this parameter large enough, you'll get garbage in the display when you open the menu.

```
&INITMNU(P1%,P2%)
P1% = number of pages to set aside for menu area buffer
P2% = number of pages to set aside for menu data structure
```

SetMenu

```
&STMNU(N%,M%,MI%,NA$,OB%,KC%)
N% = number of menus
M% = maximum number of items in any menu
MI% = menu information array, DIM MI%(1,N%),

where:
      MI%(Ø,n) = menu ID of nth menu
      MI%(1,n) = number of items in nth menu
NA$ = name array, DIM NA$(M%,N%),
where:
      NA$(Ø,n) = title of nth menu
      NA$(m,n) = name of mth item in nth menu
OB% = option byte array, DIM OB%(M%,N%),
where:
      OB%(Ø,n) = option byte of nth menu (see Table 2-2)
      OB%(m,n) = option byte of mth item in nth menu (see Table 2-4)
KC% = key character array, DIM KC%(M%,N%),
where:
      KC%(m,n) = both key characters for the mth item
      in the nth menu. Both key characters are stored in a single
      integer element of the form 256*(char1)+(char2).
```

Note: The key character array must be dimensioned even if you don't use it.

MenuSelect

&MNUSLCT(ID%,IN%)
ID% = menu ID number
IN% = item number

MenuKey

&MNUKY(K%,KM%,ID%,IN%)
K% = character typed (ASCII)
KM% = key modifier
ID% = menu ID number
IN% = item number

By the way: The parameters are not in the same order as in the machine-language call.

HiLiteMenu

&HILTMNU(ID%)
ID% = menu ID number: Ø = turn off highlighting

DisableMenu

&DSABLMNU(ID%,DIS%)
ID% = menu ID number
DIS% = disable flag:
1 = disable
Ø = enable

DisableItem

```
&DSABLITM(ID%,IN%,DIS%)
ID% = menu ID number
IN% = item number
DIS% = disable flag:
      1 = disable
      Ø = enable
```

CheckItem

```
&CHCKITM(ID%,IN%,CK%)
ID% = menu ID number
IN% = item number
CK% = check status:
      Ø = turn item off
      1 = turn item on
```

SetMark

```
&STMRK(ID%,IN%,MF%,MC%)
ID% = menu ID number
IN% = item number
MF% = mark flag:
      Ø = don't use mark
      1 = use mark
MC% = mark character (ASCII)
```

Window Commands

These commands deal with window selection and display.

InitWindowMgr

The InitWindowMgr command sets aside memory space needed for the Window Information Data Structure and for saving the part of the display obscured by the outline of the window.

You can determine the amount of memory space to reserve for the outline of the window by by calculating the perimeter of the largest possible window. The perimeter is the sum of twice the height plus twice the width. A page is 256 bytes, so to find the number of pages required, divide the perimeter of the largest window by 256 and round to the next highest integer.

To calculate the amount of memory to set aside for the Window Information Data Structures, in bytes, allow 42 bytes times the maximum number of windows that can be open at the same time. Divide

the result by 256 and round to the next highest integer to find the number of pages required.

&INITWM(P1%,P2%)
P1% = number of pages to set aside for window area buffer
P2% = number of pages to set aside for Winfo data structure

OpenWindow

Note: The dimensioned variable called WI% here can only be one dimensional.

&OPNWNDW(WI%,TS$,CS$)
WI% = window information array, DIM WI%(18),
where:

```
      WI%(Ø) is reserved
      WI%(1) = window ID number
      WI%(2) = window option byte (see Table 2-6)
      WI%(3) = window X coordinate
      WI%(4) = window Y coordinate
      WI%(5) = current content width
      WI%(6) = current content length
      WI%(7) = minimum content width
      WI%(8) = maximum content width (>Ø)
      WI%(9) = minimum content length
      WI%(1Ø) = maximum content length (>Ø)
      WI%(11) = horizontal ctl option (see Table 2-7)
      WI%(12) = vertical ctl option (see Table 2-7)
      WI%(13) = horizontal scroll maximum
      WI%(14) = horizontal Thumb pos
      WI%(15) = vertical scroll maximum
      WI%(16) = vertical Thumb pos
      WI%(17) = contents X offset
      WI%(18) = contents Y offset
```

TS$ = title string
CS$ = content string array: a one-dimensional string array, where each element is one row of the contents of the window

CloseWindow

&CLSWNDW(ID%)
ID% = ID of window to be closed

CloseAll

&CLSALL
(no parameters)

FindWindow

&FDWNDW(X%,Y%,T%,ID%)
X% = X coordinate (in mouse coordinates)
Y% = Y coordinate (in mouse coordinates)
T% = type of area point is in:
    Ø = desktop
    1 = menu bar
    2 = content region
    3 = drag bar
    4 = Grow Box
    5 = Go-Away Box
ID% = window ID if in window: Ø = not in a window

FrontWindow

&FRNTWNDW(ID%)
ID% = ID number of front window

SelectWindow

&SLCTWNDW(ID%)
ID = window ID number

TrackGoAway

&TRCKGA(GF%)
GF% = GoAway function:
    Ø = window should not close
    1 = window should close

DragWindow

&DRGWNDW(ID%,X%,Y%)
ID% = window ID
X% = X coordinate
Y% = Y coordinate

GrowWindow

Note: After you change the size of a window, you'll need to call &STCNTNT to redisplay its contents.

```
&GWNDW(ST%)
ST% = Status:
      Ø = size didn't change
      1 = size changed
```

WindowToScreen

```
&WN2SCR(ID%,WX%,WY%,SX%,SY%)
ID% = window ID
WX% = window X coordinate
WY% = window Y coordinate
SX% = screen X coordinate
SY% = screen Y coordinate
```

ScreenToWindow

```
&SCR2WN(ID%,SX%,SY%,WX%,WY%)
ID% = window ID
SX% = screen X coordinate
SY% = screen Y coordinate
WX% = window X coordinate
WY% = window Y coordinate
```

SetContent

This ampersand command changes the contents and content offsets in a window definition. It is typically used to redisplay the contents of a window after a call to GrowWindow. It can also be used for scrolling the contents of a window by making the content string the same as before and changing the X and Y offset.

You should not expect the window position with coordinates X%,Y% to contain the X%th character in the Y%th element of the content string array. Remember that there are offsets OX% and OY%, and that the window Y coordinate of the first column in a window has the value Ø, not 1. You will normally have to perform some arithmetic to figure out which character in the content string array corresponds to a given window position.

Note: The SetContent command subsumes the functions of WinString, WinChar, WinText, and WinBlock. Of those commands, the Applesoft Interface includes as separate commands only WinString and WinChar.

&STCNTNT(ID%,RE%,CS$,OX%,OY%)
ID% = window ID number
RE% = a reserved variable (should be set = Ø)
CS$ = new content string (see &OPNWNDW command)
OX% = new X offset into content (replaces value set by WI%(17))
OY% = new Y offset into content (replaces value set by WI%(18))

WinChar

Please refer to the notes under the WinOp command.
&WNCHR(ID%,X%,Y%,CH%)
ID% = window ID
X% = X coordinate of position in window
Y% = Y coordinate of position in window
CH% = ASCII code for character

WinString

Please refer to the notes under the WinOp command.
&WNSTR(ID%,X%,Y%,S$)
ID% = window ID
X% = X coordinate of position in window
Y% = Y coordinate of position in window
S$ = character string

Parameter Note: While all the WinString parameters are inputs, S$ cannot be an expression, although it can be either a simple variable or an array element.

WinOp

```
&WNOP(ID%,X%,Y%,OC%)
ID% = window ID
X%  = X coordinate of position in window
Y%  = Y coordinate of position in window
OC% = operation code:
      26 = clear from start of window to (but not including)
           position X,Y.
      27 = clear from start of line to (but not including)
           position X,Y
      28 = clear entire window
      29 = clear from position X,Y to end of window
      3Ø = clear line
      31 = clear from position X,Y to end of line
```

Important Note: The window commands &WNCHR, &WNSTR, and &WNOP change the contents of the window in the display, but they do not change the content string array (called CS$ in this manual) that is used to update the window after it has been moved, resized, re-exposed, or the like. It is up to the application to update the content string array to match the new content of the window. (It is not necessary to call &STCNTNT under these circumstances.)

You should not expect the window position with coordinates X%,Y% to contain the X%th character in the Y%th element of the content string array. Remember that there are offsets OX% and OY%, and that the window Y coordinate of the first column in a window has the value Ø, not 1. You will normally have to perform some arithmetic to figure out which character in the content string array corresponds to a given window position.

Control Region Commands

These commands deal with the control regions in the front window: the horizontal and vertical scroll bars, including the thumbs.

ActivateControl

```
&ACTVTCTL(CTL%,DIS%)
CTL% = which control region
      1 = vertical scroll bar
      2 = horizontal scroll bar
DIS% = disable flag
      Ø = disable
      1 = enable
```

FindControl

```
&FDCTL(WX%,WY%,CTL%,PC%)
WX% = window X coordinate
WY% = window Y coordinate
CTL% = control region point is in:
      Ø = content
      1 = vertical scroll bar
      2 = horizontal scroll bar
      3 = none of the above
PC% = part of the control point is in:
      Ø = inactive control
      1 = Up/Left-Arrow
      2 = Down/Right-Arrow
      3 = page up/left region
      4 = page down/right region
      5 = Thumb
```

SetCtlMax

```
&STCTLMX(CTL%,NM%)
CTL% = control to set new maximum for:
      1 = vertical scroll bar
      2 = horizontal scroll bar
NM% = new maximum for control range (must be > 1)
```

TrackThumb

```
&TRCKTHMB(CTL%,TP%,MF%)
CTL% = which control to update Thumb for:
      1 = vertical scroll bar
      2 = horizontal scroll bar
TP% = new Thumb position
```

MF% = move flag:
Ø = Thumb didn't move
1 = Thumb did move

UpdateThumb

&UPDTTHMB(CTL%,TP%)
CTL% = which control to update Thumb for:
1 = vertical scroll bar
2 = horizontal scroll bar
TP% = new Thumb position

Utility Commands

These commands provide utility functions not included in the standard Tool Kit commands.

Get Window Info

This ampersand command fills the WI% array with the current values of the array originally set in &OPNWNDW. An application program can use it to obtain values changed by the Tool Kit by user actions—for example, current width and length after a call to &GWNDW (GrowWindow). The WI% array need not be the same as the one in the original call to &OPNWNDW.

Important: The dimensioned variable called WI% here must be dimensioned to at least eighteen; otherwise, the call will return an error.

&GTWNFO(ID%,WI%)
ID% = window ID number
WI% = window information array, as dimensioned in &OPNWNDW.

DesktopError

Errors that are generated by the Tool Kit return an Applesoft error number 53, ILLEGAL QUANTITY; when this happens, a call to &DSKTPERR will return the Tool Kit command and error number. The Applesoft Interface itself can also generate other Applesoft errors such as SYNTAX ERROR and OUT OF MEMORY.

&DSKTPERR(CN%,EN%)
CN% = command number of the last call made to the Tool Kit
EN% = error code returned by that command: Ø = no errors

NextWindow

Starting with &FRNTWNDOW and using this call repeatedly until I2% = Ø, the program can select each window on the screen, in order of depth, testing or changing them as it goes.

&NXTWNDW(I1%,I2%)
I1% = input window ID number. I1% = Ø selects front window.
I25 = output ID of the next window. If none, I2% = Ø.

Set Interrupt Mask

This command sets the interrupt mask bit in the 65Ø2's status byte. If IB% = 1, the bit is set--that is, interrrupts are disabled. If IB% = Ø, the bit is cleared and interrupts are enabled.

WARNING
This is a dangerous command. It does not save or restore anything, nor does it update any status information in the Tool Kit. It is included for debugging, and may not be in the final version of the Tool Kit.

&STIMB(IB%)
IB% = value to set interrupt mask bit (Ø or 1)

Appendix A

The AppleMouse II Interface Card

To use the Apple mouse with an Apple II, Apple II Plus, or Apple IIe, you need the AppleMouse II Interface Card installed in one of the expansion slots (Apple recommends using slot 4). Like most Apple peripheral cards, it contains I/O firmware that is executed by the 65Ø2 central processor whenever you access the slot. The mouse interface card also contains its own microprocessor with firmware and a timer. The microprocessor on the card keeps track of the position of the mouse and the state of the button on the mouse. The microprocessor handles the transfer of mouse information and other communications between the card and the central processor.

Passive Versus Active Operation

Most positioning devices used with the Apple II, such as the joystick and the graphics tablet, are passive devices: they don't require any processing until an application program requests information from them. The mouse, on the other hand, is an active device, at least at the hardware level: movement of the mouse requires immediate attention to keep the system from losing track of its position and direction.

A computer normally handles this need for immediate response by means of interrupts. When the mouse is moved rapidly, it generates interrupts often enough to have a significant impact on the computer's operation. If the computer is engaged in other tasks that are dependent on precise timing, as the Apple II often is, the added burden of processing the interrupts from the mouse can be intolerable.

To reduce the interrupt burden on the Apple II's processor, the AppleMouse II uses an intelligent interface card. The card has an MC68Ø5 microprocessor that is dedicated to keeping track of the mouse, thus making it possible for the AppleMouse II to operate as either an active device or a passive device. In the Passive Mode, the MC68Ø5 determines the instantaneous movement and direction of the mouse and stores the information on the card until the processor in the Apple II requests the information. Thus, the AppleMouse II can act like a

passive device in applications that cannot tolerate interrupts, or, for applications where interrupts are appropriate, it can operate as an active device.

Mouse Interrupts

One reason to use the mouse in Interrupt Mode is to be able to move a cursor on the display screen without the flicker produced by updating the cursor during the wrong part of the display refresh cycle. In Interrupt Mode, the AppleMouse II generates interrupts that are synchronized with the vertical blanking interval.

The Apple IIe has a signal named VBL, but it isn't available as an interrupt. The VBL signal is not available at all on an Apple II or Apple II Plus, so the mouse card has a hardware timer that it uses to generate interrupts synchronized with the vertical blanking interval.

Because the AppleMouse II transmits an interrupt request only at the beginning of a vertical blanking interval, it cannot generate interrupts faster than 6Ø times per second. This limits the number of mouse interrupts and keeps the mouse from monopolizing the central processor.

The TimeData Firmware Call

There is a little-used call in the firmware on the AppleMouse II card. That call sets the interrupt rate to either 5Ø or 6Ø Hz. The default is 6Ø Hz., which keeps the VBL interrupts the card generates in step with the true VBL on a North American Apple II. For European machines, the VBL rate is 5Ø Hz.

The low byte of the TimeData entry-point address is $Cn1C. Input data is in the accumulator. With the accumulator set to Ø, TimeData sets the VBL rate to 6Ø Hz. With the accumulator set to 1, the call sets the VBL rate to 5Ø Hz. The only valid accumulator contents for this call are Ø and 1. On output, the carry bit is clear and the screen holes are unchanged.

You should call TimeData just before calling InitMouse. If you do not call TimeData first, the VBL rate will be set to 6Ø Hz when you call InitMouse.

Appendix B

The Mouse Firmware Interface

On the Apple IIc, the interface hardware and firmware for the AppleMouse II is built in. On the Apple IIe, the user must install a mouse interface card in order to use the AppleMouse II. The interface card for the AppleMouse II contains the firmware that communicates with and controls the mouse hardware.

The Apple II MouseText Tool Kit uses the mouse firmware in the Apple IIc or in the card in the Apple IIe to operate the mouse. This appendix describes the interface to the firmware.

Note: If you do all your mouse operations via Tool Kit commands, you do not need to communicate directly with the mouse firmware and so do not need to learn the material in this appendix.

Finding the Mouse Card

The AppleMouse II interface card can be installed in any peripheral slot except slot Ø; use of slot 4 is recommended but not required. The firmware on the card stores signature bytes in five of the memory locations assigned to the slot it is in. The addresses and values of the signature bytes are as follows:

Address	Value
$CnØ5	$38
$CnØ7	$18
$CnØB	$Ø1
$CnØC	$2Ø
$CnFB	$D6

The letter n in the addresses stands for the slot number. Your program can determine which slot the mouse card is in by reading the memory locations for each value of n from 1 to 7 and comparing the values with the values shown above.

Reading Mouse Data

The mouse firmware stores position and status information in the display buffer locations reserved for the slot the mouse card is in (the screen holes, also called mouse holes). When you call the ReadMouse routine or the ServeMouse routine (described later in this appendix), the firmware updates the information in the mouse holes. Your program can address these locations by using the slot number as an index, as indicated by the letter n in Table B-1.

By the way: Chapter 6 of the Apple IIe Reference Manual describes the way you address the reserved screen locations.

WARNING
If your program ever uses the auxiliary memory in the Apple IIe, be sure that you get all the switches set back to main memory before you use the Tool Kit. If you write data into the reserved screen locations in the auxiliary memory, not only will the mouse firmware not read them, but you may cause other firmware to malfunction (spelled c-r-a-s-h).

Table B-1. Screen Locations for Mouse Data

Address	Contents
$478 + n	Low byte of X position
$4F8 + n	Low byte of Y position
$578 + n	High byte of X position
$5F8 + n	High byte of Y position
$678 + n	(used by the firmware)
$6F8 + n	(used by the firmware)
$778 + n	Button and interrupt status
$7F8 + n	Current Operating Mode

In its normal operating position (oriented with its cable directed away from the user), the value of the X position coordinate increases as the mouse is moved to the right and the value of the Y position coordinate increases as the mouse is moved toward the user. The maximum values of X and Y are -32768 to +32767, but the firmware normally clamps them to the range Ø to +1Ø23 ($Ø to $3FF). You can change the clamping range by calling the ClampMouse routine, which is described later in this appendix.

The smallest mouse movement that the mouse hardware can detect is one count in either the X or Y direction; that is equivalent to about Ø.Ø1 inch (Ø.3 mm). The largest movement that the hardware can handle is 16 bits in either axis. A change of position from -32768 to +32767 corresponds to about 6Ø feet of mouse movement.

The bits in the button and interrupt status byte are assigned as shown in Table B-2, where a value of 1 means the function is true.

Table B-2. Button and Interrupt Status Byte

Bit #	Function
7	Button is down
6	Button was down at last reading
5	Mouse moved since last reading
4	(used by the firmware)
3	Video blanking interrupt
2	Button press interrupt
1	Mouse movement interrupt
Ø	(used by the firmware)

Operating Modes

When you turn on the power, the firmware comes up in the off condition with its X and Y position registers set to Ø. You activate the firmware by loading the accumulator with a mode byte and calling the SetMouse routine. The settings of the bits in the mode byte determine the mode of operation, as shown in Table B-3.

Table B-3. Bits in the Mode Byte

Bit #	Function
7-4	(used by the firmware)
3	Enable interrupt on video blanking (VBL)
2	Enable interrupt on next VBL after button pressed
1	Enable interrupt on next VBL after mouse movement
Ø	Turn on the mouse

You can enable any combination of interrupts by setting the appropriate bits in the mode byte. You can set mode combinations that don't make sense, such as $Ø2: Mouse Off plus Enable Interrupt On Mouse Movement, which acts just like $ØØ: Mouse Off.

Setting the low bit in the mode byte to Ø turns off certain functions of the mouse: the mouse position is not tracked, calls to ReadMouse don't update the status byte or the screen holes, and button and movement interrupts are not generated. Other mouse functions will work as usual: PosMouse and ClearMouse will change the mouse position data, ClampMouse will set new values, and so on. Turning the mouse on and off by changing the mode byte does not reset any mouse values.

WARNING
You must not set the high bits of the mode byte. Mode byte values greater than $ØF will cause the SetMode routine to return an illegal-mode error.

Passive Mode

Calling the SetMouse routine with a mode byte of $Ø1 puts the firmware into Passive Mode (no interrupts occur). Passive mode is the simplest way to use the mouse, and it is the only way to use it in systems with peripherals that cannot tolerate interrupts.

In Passive Mode, the interface card stores mouse information without affecting the operation of the CPU. When your program calls the ReadMouse routine, the firmware updates the mouse information in the screen locations, where your program can read it.

Interrupt Mode

If your program uses interrupts, it must include an interrupt handling routine that calls the ServeMouse routine. The ServeMouse routine determines whether the interrupt was caused by the mouse. If it was, the ServeMouse routine calls ReadMouse.

Depending on the setting of the mode byte, the firmware can interrupt the CPU on one or more of the following events:

- Mouse motion

- Mouse button pressed

- Display video blanking

You can set the mode byte to $Ø8—mouse off, VBL interrupt on—to generate interrupts on display video blanking (VBL) only. Regardless of the kind of event that causes the interrupt, the mouse hardware will interrupt the CPU only at the beginning of the video blanking interval, which occurs every 6Øth of a second. This enables your program to update the display between screen refresh cycles and avoid making the display flicker.

Unclaimed Interrupts

There is a bug in the AppleMouse II firmware that can effect the way ServeMouse works. If the application program takes more than one video blanking cycle (normally about 16 milliseconds) to respond to a mouse-generated interrupt, there is a chance that ServeMouse will not claim the interrupt. In a ProDOS or Pascal environment, this can be fatal. There are several possible ways to avoid this problem.

One approach, if you are not working under a system like ProDOS or Pascal, is to make sure that unclaimed interrupts aren't fatal to your system and just ignore them. Another solution is to make sure that you always service interrupts within one VBL cycle (one sixtieth of a second). If you have to turn off interrupts for that long or longer, you should first use SetMouse to set the mode to Ø and call ServeMouse to clear any existing interrupt.

If you are working under an established operating system, like ProDOS or Pascal, for which unclaimed interrupts are fatal, you can use one of the following suggestions to make sure that all interrupts are claimed.

If the mouse is the only interrupting device, write your interrupt handler so that it claims all interrupts.

If the mouse is not the only interrupting device, there are three ways of handling the problem. One is to write the mouse interrupt handler to claim all unclaimed interrupts and make sure that it is installed last. Another method is to write a spurious interrupt handler (sometimes called a demon), not associated with any device, that claims all unclaimed interrupts. This interrupt handler must be installed last. The third method is to include code in every interrupt handler to determine whether that interrupt handler is last. If it is, then that interrupt handler claims any unclaimed interrupts, even if not generated by its device.

Making Calls to Mouse Firmware

Your programs make calls to the mouse firmware by means of a table that conforms to Apple Firmware Protocol 1.1, described in the Apple IIe Design Guidelines as Pascal 1.1 Protocol. Table B-4 contains the low byte of the entry address of each of the firmware routines. (The high byte of each address is $Cn, where n is the number of the slot the mouse interface card is in.) The address bytes are stored in locations $Cn12 through $Cn19, arranged as shown in Table B-4.

Table B-4. Entry Point Address Bytes

Location	Contents
$Cn12	Low byte of SetMouse entry-point address
$Cn13	Low byte of ServeMouse entry-point address
$Cn14	Low byte of ReadMouse entry-point address
$Cn15	Low byte of ClearMouse entry-point address
$Cn16	Low byte of PosMouse entry-point address
$Cn17	Low byte of ClampMouse entry-point address
$Cn18	Low byte of HomeMouse entry-point address
$Cn19	Low byte of InitMouse entry-point address

Thus, for a mouse card installed in slot 4, you can calculate the entry address for the SetMouse routine by adding $C4ØØ to the contents of location $C412. Your program can use the values in the table to construct a jump table to use for calling the routines.

By the Way: You must disable interrupts before calling the mouse firmware.

Parameter Passing

Before calling any of the firmware routines, your program must load the X and Y index registers with the number of the slot the mouse card is in, as follows:

X index register: $Cn

Y index register: $nØ

Your program passes information to certain firmware routines via the accumulator and the screen locations, as noted in the descriptions of the routines.

When your program regains control, the contents of the accumulator and the index registers will be undefined, except as noted in the descriptions of the routines. The carry bit indicates the error status of the routine just ended:

Successful execution: C = Ø

Unsuccessful execution: C = 1

The Firmware Routines

This section describes the functions of the firmware routines whose entry-point addresses are given in the previous section.

SetMouse

SetMouse starts the mouse operating in the mode indicated by the contents of the accumulator, as defined in the "Operating Modes" section earlier in this appendix. If the mode byte is greater than $ØF, the routine will return with the carry bit set to one, indicating an error. This routine does not clear the screen locations used for storing mouse data.

ServeMouse

If the pending interrupt was caused by the mouse, ServeMouse sets the status byte at location $778 + n to show what event caused the interrupt. Upon return from this routine, the carry bit is set to Ø if the interrupt was caused by the mouse; otherwise, the carry bit is set to 1. This routine does not update the other mouse screen locations.

Note: This routine is an interrupt service routine; it does not require particular values in the accumulator or the index registers.

ReadMouse

Readmouse transfers the current values of the mouse X and Y position and button data into the appropriate screen locations and sets bits 1, 2, and 3 of the status byte at location $778 + n to Ø. On return, the carry bit is Ø.

ClearMouse

ClearMouse sets the mouse's X and Y position values to zero, both on the interface card and in the screen locations. It does not change the contents of the interrupt and button status byte. On return, the carry bit is Ø.

PosMouse

PosMouse sets the mouse X and Y position to the values in the screen locations. On return, the carry bit is Ø.

WARNING
Do not change the contents of any screen locations other than the X and Y position locations.

ClampMouse

ClampMouse sets the clamping bounds for either the X or Y position value. To clamp the X direction, load the accumulator with a Ø; to clamp the Y direction, load the accumulator with a 1. Store the new bounds in the slot Ø screen locations, as follows:

$478	low byte of lower clamping bound
$4F8	low byte of upper clamping bound
$578	high byte of lower clamping bound
$5F8	high byte of upper clamping bound

On return, the carry bit is Ø and the X and Y position screen locations are undefined. To get valid position data, you have to call the ReadMouse routine.

HomeMouse

HomeMouse sets the internal position values to the upper-left corner of the clamping window. On return, the carry bit is Ø and the X and Y screen locations are changed.

InitMouse

InitMouse sets internal mouse data to default values and synchronizes the interrupt timer on the card with the display vertical blanking. On return, the carry bit is zero and the screen locations are unchanged. To get valid position data, you have to call the ReadMouse routine.

WARNING
On the Apple II plus, the InitMouse routine clears the Hi-Res screen in order to synchronize its timer with the vertical blanking, so you should display Hi-Res graphics only after you have called InitMouse.

Appendix C

The Mouse Pascal Attach Driver

What's-It-All-About Department: The material in this appendix is not part of the MouseText Tool Kit. It is included here because it is new and is not described in any existing manuals.

Installing the Mouse Pascal Attach Driver

The Pascal disk that came with the Tool Kit contains two versions of the Pascal mouse I/O attach driver. It also contains the file SYSTEM.ATTACH, which performs the attach operation each time the user starts with it on the system disk. If the mouse driver is the only one you need to attach, all you have to do is copy the appropriate files onto your system disk. Table C-1 contains a list of the files on the Pascal disk that came with the Tool Kit.

Table C-1. Attach Files

File Name	Contents of File
SYSTEM.ATTACH	The system code file that performs the attach operation each time the system is initialized.
ATTACH.DRIVERS	Driver with its own interrupt manager.
ATTACH.DATA	Data for driver with own interrupt manager.
M.ATTACH.DRIVER	Add to existing drivers with interrupts.
M.ATTACH.DATA	Data to add to drivers with interrupts.

There are two versions of the ATTACH.DATA and ATTACH.DRIVERS files, one with interrupts and one without. If the mouse is the only source of interrupts in your system, use the files named ATTACH.DRIVERS and ATTACH.DATA . If you already have other attach drivers that include an interrupt handler, you can add just the mouse driver by using the files named M.ATTACH.DRIVER and M.ATTACH.DATA . To do this, you'll have to use the Library Program to make a new ATTACH.DRIVERS file with the mouse driver added to your other attach drivers. You'll also have to execute the ATTACHUD.CODE utility program to make a new ATTACH.DATA file. For a complete description of Pascal attach drivers and the procedures to follow in installing them, see Pascal Tech Note #11.

About Pascal Attach Drivers

Pascal 1.1 and Pascal 1.2 for the Apple II include a method for adding custom I/O drivers to the system. To add a driver using this method, you have to use the programs ATTACHUD.CODE and SYSTEM.ATTACH provided by Apple.

When the system is initialized, part of the program SYSTEM.PASCAL looks for the program SYSTEM.ATTACH on the main system disk. If program SYSTEM.ATTACH is present, the system executes it before executing SYSTEM.STARTUP. SYSTEM.ATTACH, in turn, uses files named ATTACH.DATA and ATTACH.DRIVERS, which must also be on the main system disk. ATTACH.DATA is the file you created using the ATTACHUD program, and ATTACH.DRIVERS is a library file that contains all of the drivers being attached.

SYSTEM.ATTACH installs the attach drivers in the Pascal heap space below

the point where ordinary programs access it. This reduces the stack and heap space available to the program by an amount equal to the size of the drivers.

The Pascal Interface

Table C-2 shows the Pascal I/O calls for each of the mouse firmware entry points. An outline of the functions of the direct I/O calls follows.

Table C-2. Pascal I/O Calls

Firmware entry point	Direct I/O call
PINIT	none
PREAD	none
PWRITE	none
PSTATUS	none
SETMOUSE	UNITSTATUS control code Ø
SERVEMOUSE	interrupt handler
READMOUSE	UNITREAD
CLEARMOUSE	UNITCLEAR
POSMOUSE	UNITSTATUS control code 1
CLAMPMOUSE	UNITSTATUS control code 2
HOMEMOUSE	UNITSTATUS control code 3
INITMOUSE	UNITCLEAR (first time only)

UNITCLEAR
reset mouse position to Ø, Ø.
reset user interrupt address to No-op.
reset clamping to default values [(Ø, 1Ø23), (Ø, 1Ø23)].

UNITREAD
read x, y button status
The read buffer should be defined as follows:

```
ReadBuffer: record
               X:  integer;
               Y:  integer;
               Button:  integer;
            end;
```

UNITWRITE
No-op.

UNITSTATUS
CONTROL CODE Ø: set Mouse Mode and user interrupt address
The control data buffer should be defined as follows:

```
Buffer: record
           MouseMode: integer;
           IntAddr: integer;
        end;
```

IntAddr should be obtained by a call to the user's interrupt handler. (See the GetIntAddr procedure in MouseInt.Text). If IntAddr is zero, the user interrupt address will be set to a No-op (RTS instruction).

CONTROL CODE 1: set mouse position
The control data buffer should be defined as follows:

```
Buffer: record
           X: integer;
           Y: integer;
        end;
```

CONTROL CODE 2: clamping
The control data buffer should be defined as follows:

```
Buffer: record
           MouseLeft: integer;
           MouseRight: integer;
           MouseTop: integer;
           MouseBottom: integer;
        end;
```

CONTROL CODE 3: Home mouse.

STATUS CODE Ø: return Mouse Mode and interrupt address
The status data buffer should be defined as follows:

```
Buffer: record
           MouseMode: integer;
           IntAddr: integer;
        end;
```

STATUS CODE 1: No-op

STATUS CODE 2: return clamping values
The status data buffer should be defined as follows:

```
Buffer: record
           MouseLeft: integer;
           MouseRight: integer;
           MouseTop: integer;
           MouseBottom: integer;
        end;
```

STATUS CODE 3: No-op.

Interrupts

Call SERVEMOUSE.
Call user interrupt handler. If no user interrupt, the call defaults to a No-op (RTS instruction).

Appendix D

Sample Program

This appendix contains a sample program showing how to use the mouse and the Tool Kit. The disks that contain the Tool Kit routines also contain three versions of a sample program, in Pascal, Applesoft, and assembly language. All three sample programs are functonally the same. The pseudocode listing that follows is similar to those sample programs, but it is not identical. To find out exactly what the sample programs look like, you should list them from the disk.

The pseudocode program is an example of the way the Tool Kit is intended to be used. The program includes the following functions.

- start the desktop
- set up menus
- set up a cursor
- track the mouse
- display a pull-down menu
- enable and disable an item in a menu
- open a window
- select a window
- drag a window
- grow a window
- scroll the contents of a window
- close a window

The user stops this program by selecting the "Quit" item in the menu.

Pseudocode Listing

Here is the pseudocode listing of the program.

```
call StartDeskTop                ; start up the Tool Kit
call InitMenu                    ; allocate screen save space
call SetMenu(DemoMenu)           ; set up our menus
call ShowCursor                  ; turn on cursor
```

```
call InitWindowMgr                    ; allocate screen save space for window
quitflag := false                     ; used to terminate program

while not quit flag do                ; main loop
; call CheckEvents                    * no longer needed in version 2 *
  call GetEvent                       ; get the next event in event queue
  case eventtype of                   ; base action on type of event returned
    button_up, no_event, drag_event, open_apple_drag_event :
      do nothing                      ; we are ignoring these
    keypress : call HandleKeys        ; handle keyboard input from user
    button_down : call HandleButton   ; handle button down on mouse
  end case
end while                             ; end of main loop
do any clean up
end program                           ; end of program

HandleKeys :                          ; character input is enter here
  if open_apple_key down do           ; check for commands
    call MenuKey                      ; translate into menu command
    call MenuCase                     ; and execute it
  end if
  return

HandleButton
  call FindWindow                     ; where did button go down ?
  case event_location of              ; base action on where it occurs
    in_desktop : do nothing
    in_menu : call HandleMenu         ; menu bar, menu operation
    in_content : call DoContent       ; content region, find out more
    in_drag_bar : call DragIt         ; drag bar, drag the window
    in_grow : call DoGrow             ; growth region, grow the window
    in_close : call CloseIt           ; close the top window
  end case
  return

HandleMenu
  call MenuSelect                     ; have toolkit perform selection
  call MenuCase                       ; execute selection
  return

MenuCase                              ; execute the menu selection
  if menu_id = Ø
    then do nothing                   ; nothing selected
    else do
      case menu_id & menu_item
        do corresponding operation
      end case
      call HiliMenu(Ø)                ; task is done, turn off highlight
  return

DoContent                             ; button down inside a window
  call FrontWindow                    ; find front window id
```

```
  if button_down does not occurs in front window
    then call SelectWindow          ; bring that window to front
    else do
      call ScreenToWindow           ; use local coordinate
      call FindControl              ; find if it occur in control
      case point_is_in
        in_content : depend on application, nothing here
        in_vertical_scroll_bar, in_horz_scroll_bar :
          call ScrollBar            ; perform scrolling
        in_dead_zone : do nohing
      end case
  return

ScrollBar
  case where_in_scroll_bar
    arrow, page :
      scroll 1 or n lines
      call UpdateThumb              ; udpate thumb position
    thumb :
      call TrackThumb               ; let toolkit track thumb movement
      if thumb_moved then scroll accordingly
  end case
  return

DragIt
  call SelectWindow                 ; bring window to front if it is in back
  call DragWindow                   ; let toolkit follow the drag
  return

DoGrow
  call GrowWindow                   ; let toolkit follow the growth
  if size_changed do                ; if size of windwo changed extra work
    call SetCtlMax                  ; thumb position etc may be changed
    call ActivateCtl                ; scroll bar may become active/inactive
    call WinBlock                   ; window is blank afterwards, update it
  return
```

Appendix E

MouseText Characters

The character generator ROM in the Apple IIc includes a set of text icons in the alternate character set. Apple is planning to make these icon characters available on the Apple IIe as well. The primary purpose of the new icon characters is for producing interactive displays using the Apple Mouse or other pointing devices.

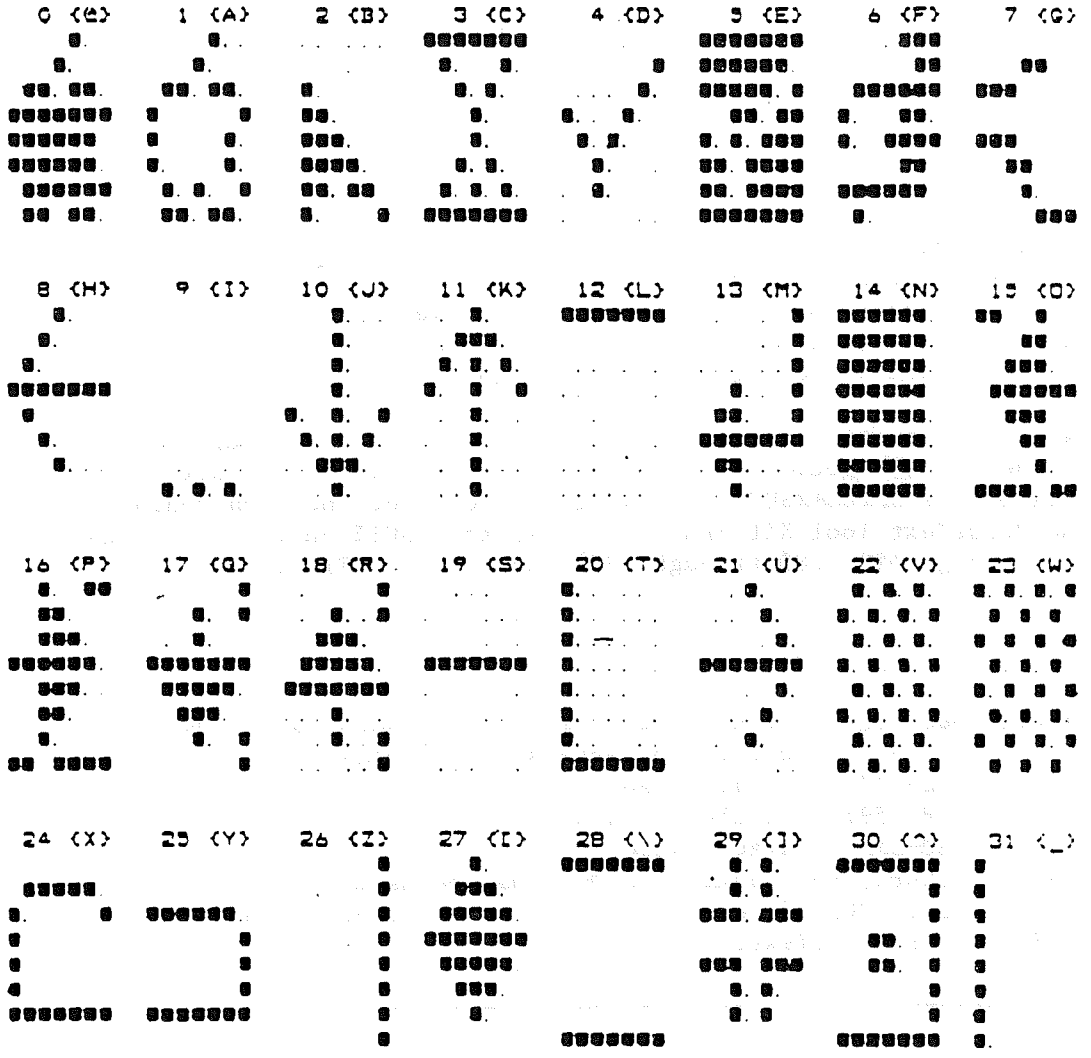
The new icon characters replace one of the sets of inverse uppercase letters (and a few special characters) in the alternate character set (selected by the ALTCHARSET soft switch). To print the icon characters with the MouseText Tool Kit installed, use the ASCII character values from 128 through 159 ($8Ø through $9F), as shown in Figure E-1.

ASCII Note: The Tool Kit interprets ASCII codes as follows:

Ø-31:	control characters or mousetext
32-127:	normal video
128-159:	MouseText characters
16Ø-255:	inverse video

In the future, the range from Ø-31 may be used as control codes only. Therefore, you should use the range from 128-159 for MouseText.

Figure E-1 The Mouse Text Icon Characters



Appendix F

Tool Kit Error Codes

Table F-1 is a cumulative list of the error codes returned in the 65Ø2's accumulator when a MouseText Tool Kit command encounters an error condition. The error codes returned by each command are listed with the commands in Chapter 2.

In addition to the error codes returned by individual commands, the first three listed here are generic error codes that can be returned by any command.

Table F-1. MouseText Tool Kit Error Codes

1 ($Ø1)	Illegal command number
2 ($Ø2)	Wrong number of parameters
3 ($Ø3)	StartDeskTop hasn't been called
4 ($Ø4)	Machine or operating system not supported
5 ($Ø5)	Invalid slot number (less than Ø or greater than 7)
6 ($Ø6)	Mouse Interface Card not found
7 ($Ø7)	Interrupt mode in use (Program specified interrupt mode in StartDeskTop, so it can't call CheckEvents.)
8 ($Ø8)	Menu ID was not found
9 ($Ø9)	Item Number is not valid
1Ø ($ØA)	Save area (from InitMenu) is too small
11 ($ØB)	Tool Kit could not install interrupt handler
12 ($ØC)	Window with same ID already open
13 ($ØD)	InitWindowMgr buffer too small for this window
14 ($ØE)	Bad Winfo -- tried to open window with ID = Ø, or conflicting max and min width or length
15 ($ØF)	Window ID number not found
16 ($1Ø)	There are no windows
17 ($11)	Error returned by user hook routine
18 ($12)	Bad control ID (not 1 or 2)
19 ($13)	Event queue full, event not posted
2Ø ($14)	Illegal event, event not posted
21 ($15)	Illegal UserHook ID number (not Ø or 1)
22 ($16)	Operation cannot be performed

APPLE II HUMAN INTERFACE TOOLS

A good human interface is vital to your software product's success. Top-selling applications packages are those that are simple to learn, easy to remember, consistent, and efficient, as well as reliable and accurate.

Apple Computer has developed guidelines and a number of human interface tools for applications software development, reflecting years of collective human/computer research. We offer the Apple II human interface tool kits summarized here for inclusion in your new or updated Apple II software products. Potential benefits include "friendly" user interaction, reduced development time/costs, improved product reliability, and easier Apple II/Macintosh software portability.

Apple II Human Interface Models

Because no single style of interface is appropriate for every application, we support two approaches to human interface design for the Apple II. These are the "Filecard" and "Desktop" paradigms.

Filecard Interface

The Filecard interface was first used by Apple II's Appleworks (see accompanying example). Overlaying filecards on the screen visually represent the different menu levels with the paths available to the user. Options or activities are arranged in a hierarchy. Functions are performed in a specific order. We recommend the Filecard interface for applications where the program must exercise restraint and guidance over the user's activities. It is also a good choice for upgrading your existing, menu-driven software: It offers the visibility of modern software design without requiring major restructuring.

Desktop Interface

The more flexible, relational Desktop interface follows Macintosh design principles and calling conventions. The Apple II screen represents a desktop, with multiple screen windows representing different "documents" on the desktop. Graphic components--the drag bar, close box, grow box, and scrollbar--control window operations.

Software using this human interface is driven by events from the mouse, keyboard, and the application. Using the mouse or cursor keys with pull-down menus from the desktop menu bar, the user has a broad range of activities available at once. Graphic images can be used with text output.

Apple Computer Inc
[illegible]
[illegible]
[illegible]
TLX [illegible]

Apple II Human Interface Software

Three sets of human interface software tools are now available for Apple II software developers: a Filecard Toolkit, a MouseText Toolkit, and a MouseGraphics Toolkit. Most toolkit modules are available for Pascal, Applesoft, and assembler, except as noted.

Text-Mode "Filecard"-Style Toolkit

Four software modules are provided in this set of tools: a User Input Routine, a Console Driver, a Pascal "ConsoleStuff" Unit, and a Pascal Filecard Menu Support Unit. The modules may be used independently, except that the Pascal units require the console driver.

User Input Routine. This software module implements the standard Apple keyboard input, specified in the Apple II Human Interface Guidelines. Editing commands are supported, as are interrupt and termination characters. An insert cursor, replace cursor, default string, and fill characters can be customized to suit your application. An "immediate" mode is available, should your application need special control during the input process--e.g., for "live" syntax checking. The User Input Routine can be used with assembler, Applesoft, and Pascal programs. (The Pascal version requires the Console Driver.)

Console Driver. The 80-column Console Driver offers fast, efficient text output and screen control. The utility processes a buffer of combined text characters and screen control commands in one call. Commands include viewport and cursor controls, initialization and clears, and character normal/inverse and normal/alternate controls. The driver also returns data describing the current screen environment. It can be called from assembler and Applesoft, and is available as an "Attach" driver for Pascal.

ConsoleStuff Unit. This Pascal unit offers text formatting aids, as well as utilities for overlaying message boxes and Help Screens. It provides a 1K buffer for the Console Driver and console buffer setup aids. (This unit is not required for Applesoft or assembly-language programs.)

Filecard Menu Support Unit. This Pascal unit provides utilities for implementing the Filecard interface. There are utilities for setting up the screen areas, displaying and removing "filecards", getting the user's menu selection, and generating error boxes. Other routines work with the Console Driver to display formatted text. The unit includes a facility for building your application's hierarchical menu structure. This module is available for Pascal only; however, the toolkit's documentation provides in-depth design details useful to Applesoft and assembly-language developers.

The Desktop-Style MouseText Toolkit

With the MouseText Toolkit and the MouseText characters available now for both the IIe and the IIc, you can implement the Desktop interface -- all within text mode. The package offers complete support for mouse- or keyboard- controlled multiple windows, pull-down menus, and event handling. The utilities, designed to parallel Macintosh's interface tools, provide for menu bars, 40- or 80-col window writing and dragging, scroll bar control regions, and cursor selection and control. Error-handling is included as well. The 12K MouseText Toolkit machine-code package can be used with assembler, Applesoft, and Pascal-based applications.

Graphics-Mode Desktop Tools

This new package contains two machine-code modules: an 8K independent set of Graphics Primitives and an 8K "MouseGraphics Toolkit", which uses the Graphics Primitives.

Graphics Primitives. This software module supports 40/80-column text and Double HiRes (560x192) MousePaint-like graphics on the Apple IIe and IIc. Its utilities can be used to paint lines and polygons of different widths. Drawings, defined relative to the application's coordinate system, can be directed to either the screen or a nonscreen bitmap. Polygons can be filled with patterns or colors. The changeable drawing environment includes current pattern, pen location, pen size, pen mode (eight options), and font. The Graphics Primitives, callable from assembly language, Applesoft, and Pascal, are based on Macintosh's QuickDraw graphics package to facilitate applications software portability.

MouseGraphics Toolkit. This toolkit is similar in function and structure to the MouseText Toolkit described above. Procedures in this package support pull-down menus, windows, cursors, and event-handling. The difference is that it calls on the Graphics Primitives, allowing your application to present its information in graphic form.

Matching Apple II Human Interface Tools With Application

The Apple II human interface toolkit best suited for your software product depends on factors such as application, user, Apple II system, space, speed, and language. The MouseGraphics Toolkit, for example, provides for a more flexible, icon-oriented desktop, but it uses more memory and is slower than the MouseText Toolkit. If you are upgrading an existing application, you might prefer the hierarchical filecard-style toolkit over either of the event-driven desktop- style toolkits. Data in the accompanying chart can help you with your human interface tool selection.

Apple II Human Interface Tools - Comparison Chart

	Filecard:				Desktop:		
	Input	CDrvr	CStff	FilCd	Text	Primitive	Graphic
IIc	yes	yes	yes	yes	yes	yes	yes
128K enhncd IIe	yes	yes	yes	yes	yes	yes	yes
128K IIe	yes	yes	no	no	no	yes	yes
64K enhnced IIe	yes	yes	yes	yes	yes	no	no
64K 80col IIe	yes	yes	no	no	no	no	no
64K II+	yes	yes	no	no	no	no	no
Display Mode	Text	Text	Text	Text	Text	Graphic	Graphic
I/O Buffer	1K	1K			2K	16K	
Module Size	1.5K	3.5K	3K	3.5K	12K	8K	8K
Mouse Support	no	no	no	no	yes	n/a	yes
Mouse Req'd	no	no	no	no	no	no	no
Pascal Intfc	yes	yes	yes	yes	yes	yes	yes
ProDOS/BASIC	yes	yes	n/a	no	yes	yes	no
ProDOS/Assy	yes	yes	n/a	no	yes	yes	yes

Apple II Human Interface Tools Availability

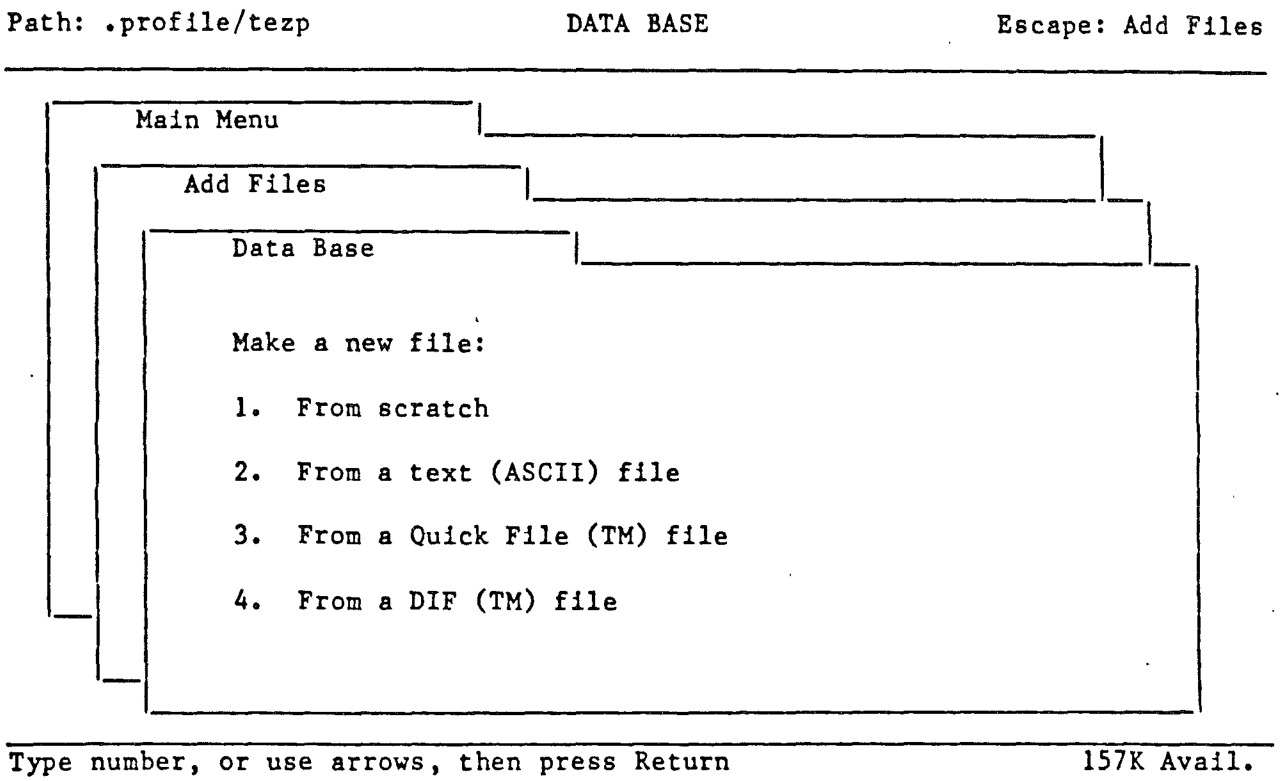
Apple IIe owners can obtain the Enhanced IIe Upgrade Kit from dealers. The enhancement is now being integrated into Apple IIe's in production. The kit is available to certified third-party Apple developers via Apple's Developer Relations Group (408-996-1010).

Revised Apple II Human Interface Guidelines have been published, replacing Apple product number A2F2116 11/82.

You can obtain preliminary versions of the Filecard Toolkit, MouseText Toolkit, and MouseGraphics Toolkit from the Apple II Developer Support Group. The final software should be available directly from Apple's Licensing Group by the end of April. Licensing fees will be nominal: $50-$100 per year.

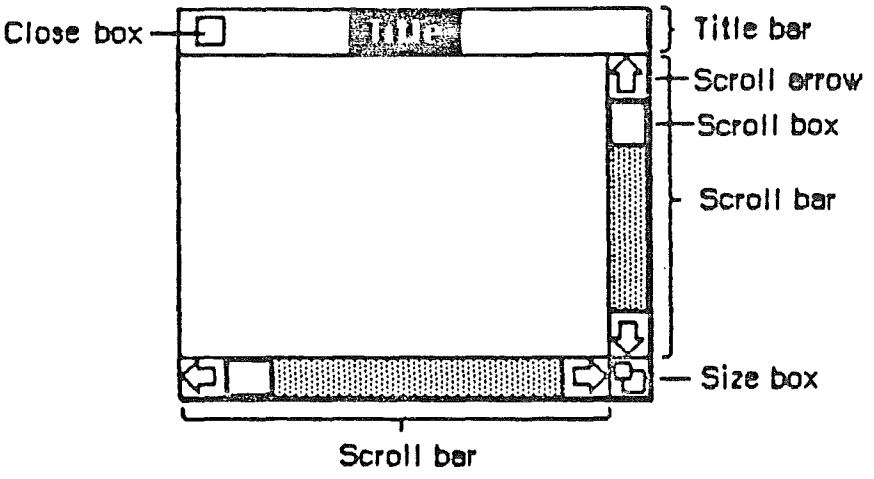
4/17/85--rjr

Apple II FileCard Sample Screen

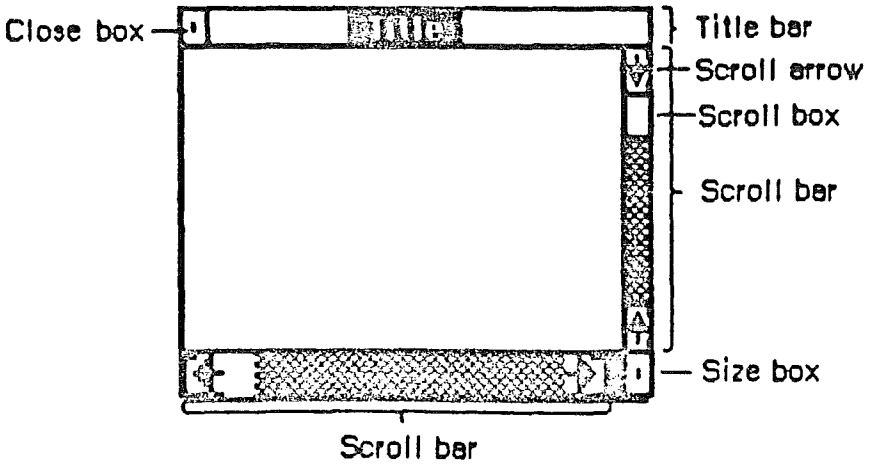


```
Path: .profile/tezp              DATA BASE              Escape: Add Files

  Main Menu
    Add Files
      Data Base

        Make a new file:

        1.  From scratch

        2.  From a text (ASCII) file

        3.  From a Quick File (TM) file

        4.  From a DIF (TM) file

Type number, or use arrows, then press Return              157K Avail.
```

Apple II MouseText Toolkit and MouseGraphics Toolkit Screens

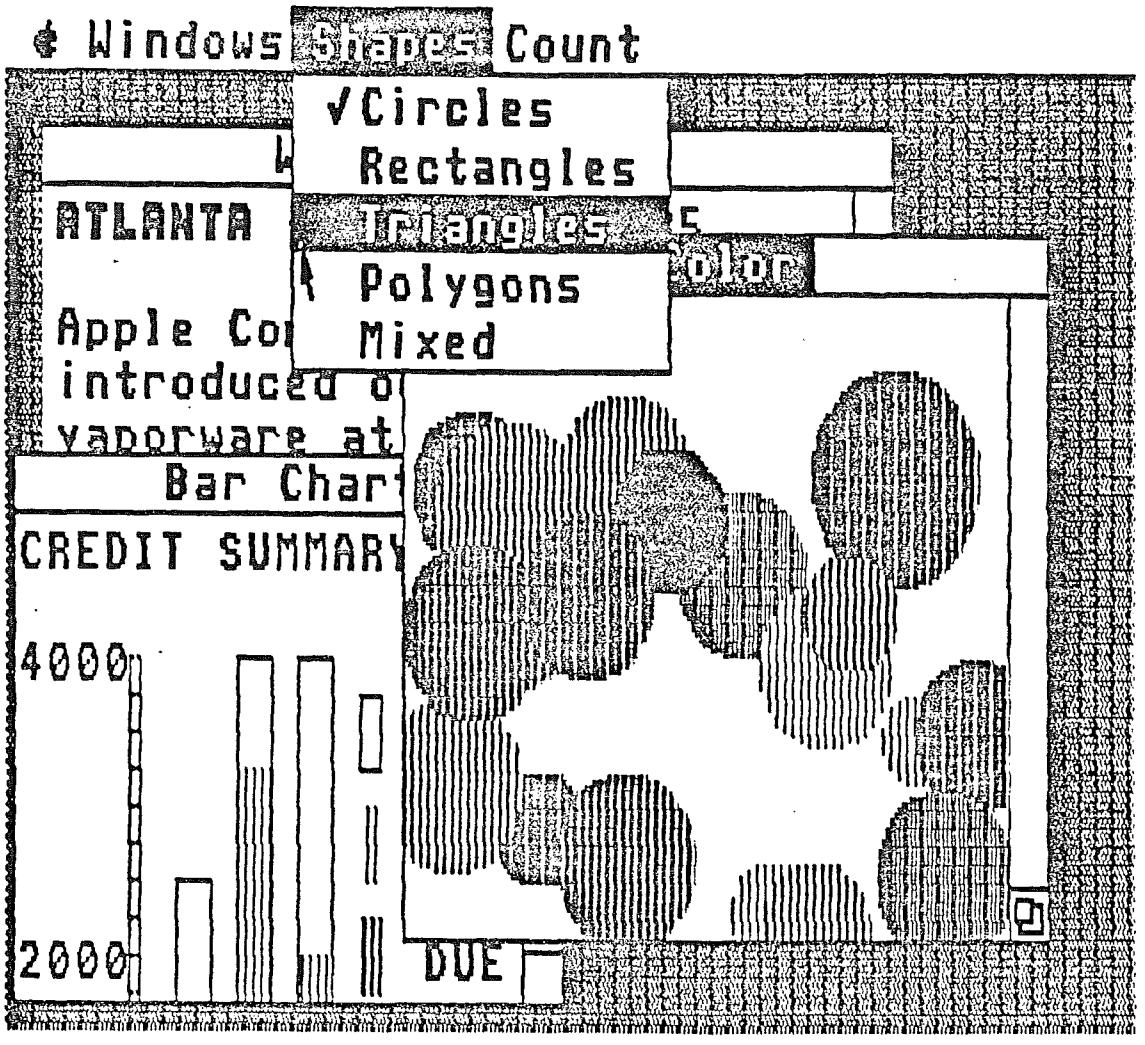


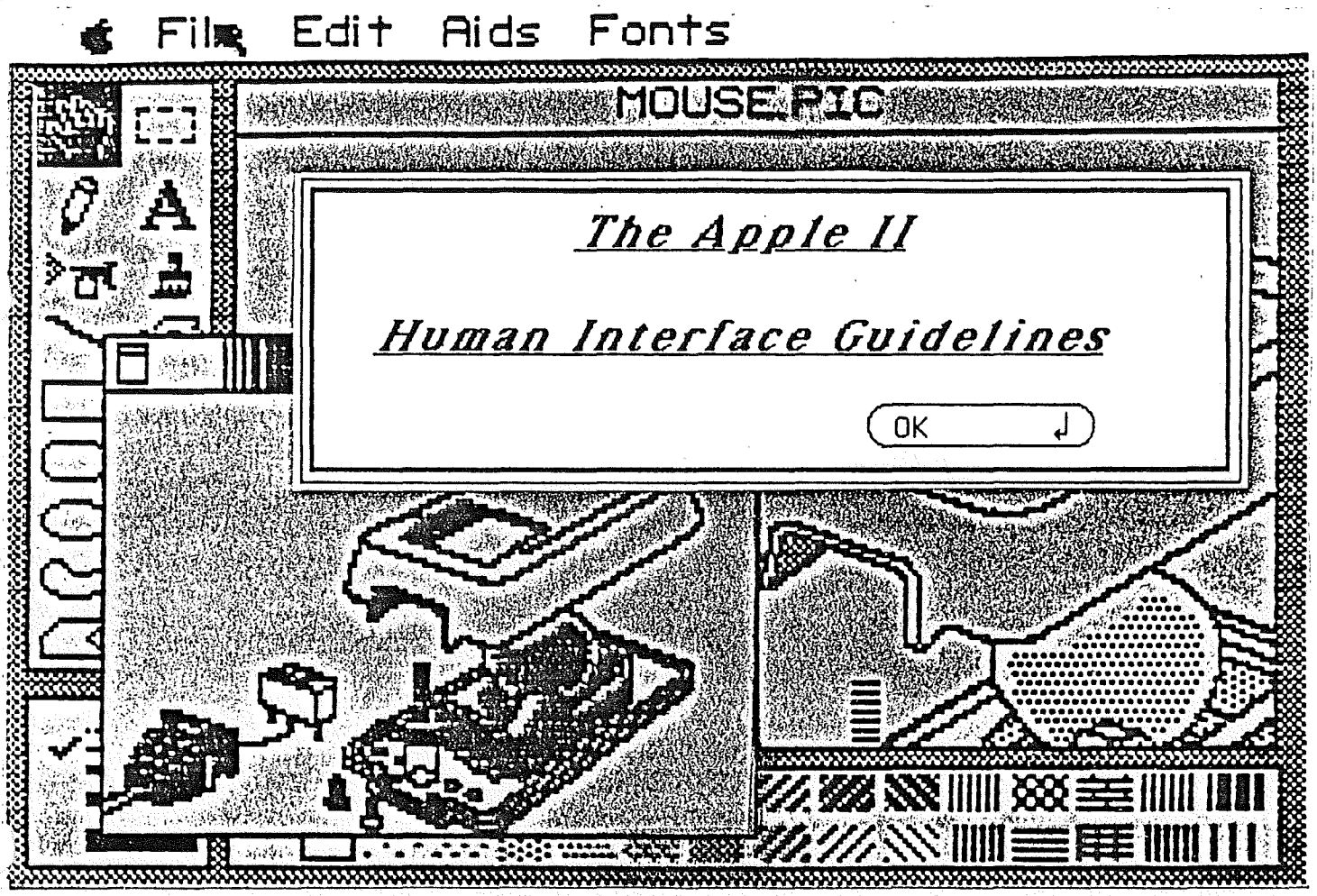
MouseGraphics



MouseText

Apple II DeskTop Sample Screen




File Edit Aids Fonts
MOUSEPIC
The Apple II
Human Interface Guidelines
OK

Apple II Human Interface Guidelines

Modification History:

1st Draft	Bruce Tognazzini	9/15/78
2nd Draft	Bruce Tognazzini	3/12/79
3rd Draft	Bruce Tognazzini	6/18/80
4th Draft	Steve Smith	2/15/81
1st Draft, Macintosh	Joanna Hoffman	3/17/82
2nd Draft, Macintosh	Chris Espinosa	10/11/82
1st Release, II	Bruce Tognazzini	1/19/83
Addendum	Bruce Tognazzini	8/ 5/83
1st Draft, II Mouse	Bruce Tognazzini	11/10/83
3rd Draft, Macintosh	Andy Averill	7/31/84
4th Draft, Macintosh	Andy Averill	11/30/84
2nd Draft, II Mouse	Bruce Tognazzini	1/15/85
2nd Release, II, Alpha	Bruce Tognazzini	3/21/85

Abstract:

This is a rough-cut of the Apple II Human Interface Guidelines that will be officially released later this year. Because of the time importance of this information and because the standards themselves are now quite stable, we have made this pre-release available to you.

These guidelines describe the most basic common features of an Apple II application. Included are two different metaphors: the Macintosh-inspired desktop, conforming to the Macintosh guidelines and including support for the user with no mouse, and the Filecard metaphor, popularized in the first AppleWorks, offering an easy upgrade path for existing, menu-based applications.

TABLE OF CONTENTS

Part I:

Introduction To Human Interface Design

Introduction
Good Human Interfaces: So Often Elusive

The human interface of a program is as vital to its success in the marketplace as is its accuracy in performing its task. An otherwise well designed, powerful piece of software or hardware is nearly useless if it is poorly human engineered. As Dr. Frank Gilbreth, the father of time and motion study said: "It is cheaper and more productive to design machines to fit men rather than to force men to fit machines."

Human interface design should come into play from the very beginning. A good design is no mean task: expect to expend a great deal of design and programming effort toward a smooth interface. For most programs with a good human interface, the total human interface effort consumes more design time, is more prone to bugs, and is harder to test than any other part. By offering you this book, along with the many human interface tools available from Apple Computer, Inc., we hope to vastly reduce your share of the human interface effort, leaving you more time to devote to the power areas of your program.

There are two primary functions of a good human interface design: make the product easy to learn, and make it easy to use, both without seriously compromising power and performance. This guide will lead you toward making the kind of hard decisions that will result in a successful, marketable product.

When the Apple II computer first came on the market, software developers experimented with a wide variety of interface designs. Some were good; some were bad. All were somewhat hard to learn, because all were unique. As time went on, though, the natural personalities of the keyboards, displays, and the computer itself led to a remarkable similarity of approach to certain basic problems of ease-of-use.

These guidelines represent a careful blending of these tacit decisions of the develoment community and the knowledge gleaned from the vast research and development project that resulted in the Macintosh computer.

Our systems and peripherals now come with training disks and materials that prepare your customers to use programs written with these guidelines: Use this book and our tools, and your customers will feel comfortable the first time they see your program at their dealer--while they are still making their buying decision.

Elements of Style: Designing an Interface that Works!

Goals

There are five goals to a program design:

1. Ease-of-Learning. 2. Ease-of-Use. 3. Satisfaction of human needs 4. Saleability 5. Power and expandability

Most of the science of human interface deals with increasing ease-of-learning and ease-of-use without seriously affecting power and expandability. Most of the art aims toward satisfying humans' need to feel warm, comfortable, and protected. All these, plus spark and flair, add up to products that sell and can be sold. We assume your competence as a designer or programmer in creating power and expandability in your program; the rest of this book will address how to fulfill the first four goals.

The Underlying Concepts

It is one thing to have the above goals in mind; it is another to be able to actively and effectively address them. The following basic concepts form an structure on which you can base both large-scale and day-to-day design decisions.

1. Familiarity

Familiarity is the single greatest factor in reducing the learning burden without affecting power and expandability. People feel comfortable with things they already know. You promote familiarity by using guidelines such as this, by conforming the flow of your product as closely as practical to the way your users did things before they "computerized," and by choosing familiar metaphors, such as desktops and file-folders, around which to build your programs.

2. Intuition

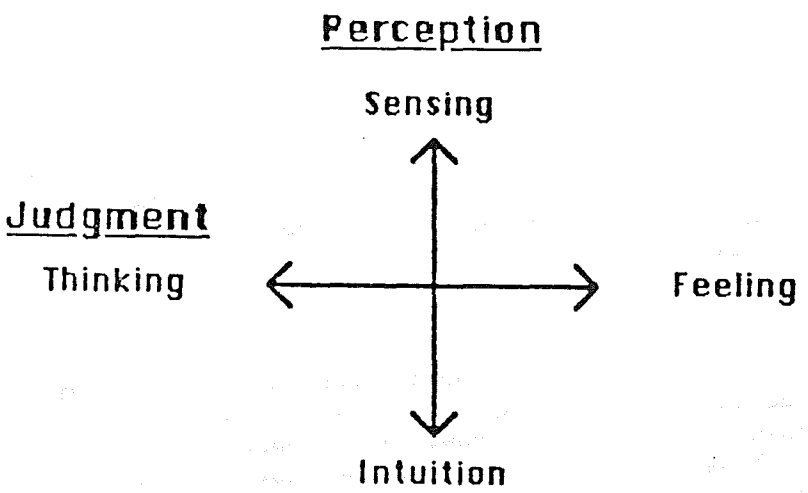
A few years ago, an engineer at Apple described the ideal interface as being "familiar and intuitive". During the preparation of this book, we asked the same engineer again, and he stated that while it was important for an interface to be familiar, it was no longer necessary for it to be intuitive. It turned out there was a good reason for this: computer scientists don't know what "intuitive" means, so we don't know how to deal with it.

Intuition is real, it has been researched, it has been defined. It is a powerful mental capability of both the designer and the user, and as such deserves practical understanding.

The Nature of Intuition

Stated perhaps over-simply, intuition is the ability to discern

patterns among often seemingly unrelated information. Jung classified intuition as a perception skill, alongside the taking in of external sensory information. He classified logical thinking as a judgement skill, alongside feeling. (He used feeling more in the sense of ethical consideration than emotional response.)



Simplified Jungian Personality Classification

In the Jungian world, logic and intuition operate at right-angles to each other. In practice (as described below), they operate sequentially: intuition, then logic. Seemingly, there should be little conflict between them, but there is evidence of unending conflict.

A prevelent theory states that logic resides primarily in the left lobe of the brain, the unquestioned location of the speech center, and that intuition resides primarily in the right lobe. It is also known that the lobes of the brain are connected in a criss-cross fashion to the body: the left lobe controls the right side, the right lobe controls the left side. Therefore, the right-hand side of the body is connected to the seat of logic, the left hand, to the seat of intuition.

Western society has a traditional culteral bias against the body's left side, left hand, and therefore right lobe of the brain. French for left is gauche, defined as lacking social grace, awkward. Latin for left is sinister. Right is held as truth, justice, morality. For centuries, left-handed people--who often test higher in intuition than right-handed people--have been discriminated against.

People with life-threatening epilepsy sometimes undergo an operation to cut the connecting links (called the corpus calloseum) between the right and left lobes of the brain. After the first such operations, scientists were puzzled by the apparent lack of side-effects. However, further examination began to reveal fascinating new insights into the relationship of the sides of the brain.

The first, simple test consisted in effect of blindfolding a subject. They then placed a three-dimentional number 5 in his right hand and asked that he hold up the number of fingers equal to it. He held up 5 fingers. When asked to verbalize the number, he spoke the word, "five". But when the test was repeated with the number 3 in the left hand, although he quickly held up three fingers, he was unable to speak the word, "three". This was due to the speach center being located on the opposite side of the brain. With the connecting links severed, that lobe had no way of knowing what the left hand held.

The psychologists next tried to find the seats of logic and intuition, by offering puzzles to each hand, puzzles that were oriented toward either logical or intuitive solutions. The right hand showed great skill with logic puzzles, less with intuitive. But the most interesting thing happened during the left hand's effort to assemble a logic puzzle. The left hand, although taking several times as long as

the right hand, was nevertheless growing close to a solution when the right hand unexpectedly came over and, rather than helping out, scattered the pieces of the puzzle all over the table. This agressive act was witnessed repeatedly with other split-brain patients.

There was a war for supremacy going on in these people's heads, a war that seemed to reach a truce several months after the operation, with the left, logic-oriented lobe achieving dominance. When one couples this strange phenomenon with the empiricle evidence of Western society's bias against left-handedness, one begins to understand why we know so little about this profoundly powerful skill--our logical, concious mind doesn't want to know.

So what is intuition, what is it good for, and how do we use it? Intuition deals with patterns: pictures, chains of events, clustering of seemingly irrelevant information. Intuition is a non-verbal skill, and words cannot effectively describe it. We have to get to it through metaphor, example, and shared experience.

Intuition operates in leaps: it churns away with no concious thought and little concious control and then suddenly springs forth with what is often described as an "Aha!" experience. An example of this takes place when you learn a new board game: you keep hearing and attempting to memorize the rules of the game and suggested strategies, but it is all very compartmentalized and difficult to keep ahold of. Then, all at once, you "get it": the entire underlying strategy and purpose of the game is instantly and permanently obvious.

Those of you who are programmers have undoubtedly experienced working until 2:ØØ in the morning on a seemingly insoluable bug problem, finally giving up, and going home to sleep. Then, at 6:3Ø AM, you wake up knowing exactly what is wrong and kick yourself for not realizing it earlier--after all, it is perfectly obvious.

Reaching a useful conclusion through intuition is a three-stage process:

1. Gathering information 2. Allowing time for "incubation" 3. Judging the results

We recognize two kinds of intuition. The first is "women's intuition," a remarkable ability to understand human relationships and interactions. Men also use intuition, but they call it "hunches". While these two kinds of intuition are different, they arn't as gender-specific as our culture has declared them; in Eastern culture both men and women develop powerful abilities to "look through" people.

1. Gathering information. This data may be new sensory information, coupled with old memories. Because intuition works on drawing together what is often logically unrelatable data, you should make a concious effort to be non-judgemental during this stage. Take in everything.

Intuition is an ancient skill, both in evolution and personal development. Dogs are no intellectual giants when compared to man, but

they are supremely adept at detecting and reflecting the mood of their masters from very subtle hints, often more so than the many psychiatrists and psychologists who have abandoned intuition in favor of pure intellect. Children are also highly intuitive, often reflecting the state of their parents' relationship that the parents are unaware of. As we grow older, we develop logic and slowly push intuition away as an inferior, "childish" skill.

Most of the great scientists and inventors either failed to grow up "properly" and loose this childhood skill or re-learned how to be a kid again. Intuition works best when you gather information with child-like wonder and rapt attention. Archimedes had given up on his logical pursuit of how to measure the purity of the gold in the king's crown without melting it down. Then, one day, he stepped into his over-filled bathtub, absorbed the fact that he was splashing water out, and realised he was displacing exactly as much water as his own volume. Coupled with prior knowledge of the relationship between volume and weight, he had his solution.

As soon as you begin to pre-judge the data streaming in, things begin to get lost. Eric Berne, the noted psychiatrist, carried out an experimentation in intuition on 4Ø,ØØØ soldiers being separated from the Army. He attempted to guess their pre-army careers based on gazing at them with rapt attention for approximately 1Ø seconds. His success rate was quite high--in the case of soldiers who had previously been farmers, he was able to guess their prior occupation 74% of the time. He then analyzed logically what he was doing to come up with these conclusions, studying facial muscle configuration, body position, eye movement. When he could analyze no more, he applied these rules to additional soldiers, while blocking the child-like attention that he had used before. His success rate immediately fell by half.

The conclusion? For intuition to have the kind and quantity of disparate information it needs to function properly, you must turn off your concious, logical mind's attempt to filter and pre-judge everything coming in. Relearn the art of being a child, of offering childlike, rapt attention to people and to problems.

2. Incubation--that time between 2 and 6:3ØAM when you thought you were just sleeping, but you were also casting about for a pattern that would solve your bug problem. Incubation can take anywhere from seconds to days. It stubbornly resists rushing, very much the way memory will perversly hold back someone's name if you really need it. Incubation time can be shortened through relaxation, practice, and encouragement.

It is likely you could have worked on your bug until noon the next day with no results. The intuitive leap came after you finally relaxed your concious mind enough for intuition to be able to function. For most of us, intuition is a rusty, repressed skill. As you begin to listen to your intuition, the number of intuitive leaps will increase and the time for incubation will decrease.

The discouraging thing about intuitive leaps is their resulting

obviousness. It is childishly obvious, for example, that what goes up must come down (unless it happens to be exceeding 18,ØØØ MPH at the time). But when Newton intuited the law of gravity, it was a real breakthrough. When you do finally find that software bug, the tendency is to berate yourself for not seeing it earlier. Intuition will be strengthened and incubation time reduced if you instead congratulate yourself for being clever enough to find it at all.

3. Judgement

Intuition usually delivers its results to the concious mind in a convoluted, metaphoric fashion. It is very easy to ignore or misread the results. Several years ago, a data-base program was under developement for a computer with large mass-storage. No effort was spared in making every section of the program as "friendly" as possible. When a particular task proved somewhat difficult to learn or use, the task was reduced by picking up bits and pieces of it within other tasks. The program slowly drifted toward being consistently somewhat difficult to learn and use.

A consultant was brought in to "simplify the interface," a task he found difficult: the original designers had done a really good job in making each section as simple as possible. It seemed simply to be a super-powerful program that had to be difficult to use because of its expanded capabilities and features. He struggled with the program for several days to no avail.

During the last four of those days, he kept remembering an administrative assistant he had once worked with. The administrative assistant used to tell all who would listen that he, the administrative assistant, had to do all the work around there, that he held the office together. The consultant kept pushing this memory away, but it kept coming back. One evening, he decided to listen to it, and he began to realize that he had seen such a person in almost every office he had ever spent time in.

Then he remembered the tiny detail that had been trying to push its way into his conciousness for days, the detail that led to the re-design and ultimate success of the product: whenever the administrative assistant complained about his terrible responsibility and crushing work-load, he always wore a smile. In fact, this administrative assistant, like so many others, are quite proud of the difficult job they do; they tend to brag about it in a negative way because they receive so little appreciation from those around them, and, of course, it is not polite to brag.

No one had never considered who their audience was beyond their being "office workers," so the consultant sat down and did a user-profile study (see: Planning) of what kinds of people would be users of the system. He discovered three groups of people, the last one being that administrative assistant his intuition was trying to remind him of:

1. The data-entry persons. These folk would be proficient typists who initially would be expected to enter a great deal of pre-existing

information. They might be temporary help, or they might be people who normally performed a different job. Their needs were for an quick to learn, easy to use interface.

2. The decision-makers. These people would be expected to draw information from the system, both by calling up data on the display and generating reports. They could be expected to be habitual users of the system: they could handle a long but gentle learning-curve that would give them progressively more power.

3. The Key Operator. These people are the ones who, in real life, read the manuals. They can be expected to spend some time with the system initially and can be expected to learn how to perform the more technical operation and maintenance tasks of the system.

Once the users of the system were identified, once their individual needs were identified, the designers were able to "unbalance" their equally-difficult interface, so that each user had a level of difficulty consistent with their abilities and the amount of time they could spend learning the system.

A smiling, complaining administrative assistant is a rather obscure hint to a program design problem, but it is typical of the way the intuitive mind communicates its results. Keep in mind that the poor, "primitive" intuition is quite incapable of speech and logic. On the other hand, it knew the answer to the problem four days before the logical mind. The finest intuitive leaps are utterly useless if you fail to listen: learn to relax when things look the darkest; you may already know the answer.

The Jungian judgement skills are Logic and Feeling. Of the eight Jungian classifiers (of which four have been presented here), only one has shown any gender bias: approximately 60% of men depend primarily on logical conclusions, while 60% of women depend primarily on ethical considerations. (This explains a lot of insoluable domestic arguments.) So-called "women's intuition" is intuition with Feeling judgement, while traditional men's hunches are intuition with Logic judgement.

Intuition and the Programmer

Programmers and designers can make use of intuition on design and debugging problems (linked with Logic) and during user-testing (linked with Feeling). Programmers in a recent test were found to depend on Intuition as the primary perception skill twice as much as the general population. By following the above steps, you can increase the power and effectiveness of your own skill.

In the section on Testing, we will discuss how intuition can be used to detect problem areas of programs during user-testing, with a method far less expensive and more effective than computer-analysis of elaborate questionaires.

Intuition and the User

Now that we've laid the foundation for an understanding of intuition, we can explore the "intuitive interface". 75% of your users depend primarily on sensory perception. They are the ones who are most helped out by familiarity and a what-you-see-is-what-you-get approach. The 25% of the population that depends on intuition are looking for simple, distinct patterns.

The intuitive interface is restrained, consistent, simple, and predictable. Techniques that work in one place work in all places. The intuitive user attemts to internalize a rational model of the program. Once this is done, he or she will use that model to predict the behavior of areas of the program not yet explored. Any inconsistencies discovered require the user to either expand the model or abandon it altogether. People who depend almost entirely on intuition (a small but significant proportion of the population), when faced with an erratic program, either must memorize each area or abandon the program.

How do you write for these people? Spend time on your conceptual model (see below) and stick to it: if you must redesign a section of your interface, go back and reconsider the impact of that change on every other part of the program. Build designs that will allow future expansion without turning the original, simple model upside down. We have all been exposed to language and operating systems that started out simple and have now ended up with such a topsy-turvy mapping that they seem more like an Adventure game than a serious environment.

Use conceptual models that heighten intuitive grasp. Such a model is the Macintosh windowing illusion, now available for the Apple II family. Beyond the leverage of familiarity this interface offers you, it is highly consistent and therefore intuitive. Modes (where necessary) are distinct, and the resulting behavior changes are predictable. It is also expandable in such a way that the original fabric is not torn apart: an entirely new power, such as a spelling-checker, can be added to a word-processor simply by adding an item on a pull-down menu. While you, as a programmer, may go completely nuts attempting to integrate it into the old code (some things haven't changed), your user will have no problem adding the new power to his or her old model.

Aiding Memorability

Our market research has consistently shown that people taper off their software buying. One of the chief reasons for this is that people tire of having to learn new software; often their last piece of software was so difficult for them to learn and memorize that they lost interest.

Programmers often have a superior ability to remember abstractions, such as numbers, and disconnected details, such as lists of keywords. If you have a superior memory, you should be particularly sensitive to the needs of more average people.

The greatest aid to memorization is familiarity: if the person already knows how something functions, they don't have to memorize anything. By using a standard human interface, you save your user from having to remember

anything about your interface--he or she learned to use it already. The second greatest help is a good conceptual model. The simpler the model, the easier it is to grasp and remember.

Current theory of memorization holds that people remember not the event but a simplified set of rules which allow the event to be reconstructed. One of the aids to this process is tying elements to people or events that have already been learned. This phenomenon can be seen with crime witnesses, all of whom have different recollections of the same crime. One witness will remember that the criminal had a bald head, "just like my Uncle Harry". A teenager may not have remembered anything about the criminal, except he was old, but will be prepared to discuss in detail the carburator on the criminal's 1957 Ford.

While details of such reconstructions will be selective and sometimes conflicting, the primary event will usually be uniform: the man came into the bank, he robbed it, he left. By making building your program on one simple conceptual model, generatable by a few powerful rules and concepts, your user can form his or her own internal model and memorize the few details necessary to reconstruct it.

Increasing Receptivity

The most easy-to-learn program in the world will not be learned if your user resists it. You want your program to be sellable and useable, so your program and documentation must be able to overcome fear and anxiety, boredom, and frustration. It must be capable of creating trust and confidence in the user.

You must remember that you are dealing with a human being and tailor your interface to deal gently with the kind of fears and anxieties that the very existence of your program may provoke. If you are designing a data-base program for small businesses, for example, you must consider and plan for the fact that the employees may fear that the computer is there to eliminate their job. With any program you ever expect to sell in a retail store, you must be sensitive to the salesman's fear you will embarrass him or her, and the customer's preconception that this program is going to be far too difficult to learn. You overcome these kinds of fears by making programs familiar, intuitive, and memorable, and by being sensitive to the psychological needs of people.

In addition to fear and anxiety, people who use your program constantly can begin to suffer from boredom, thereby lowering productivity. There are a number of ways to reduce boredom, segmenting tasks, rewarding achievement with positive feedback. We will not go into any exhaustive list here (least you become bored); just keep boredom in mind when doing testing. If you notice production drop with time instead of rise, you probably need to explore ways to brighten people up.

Frustration means something has seriously failed in the human interface. It usually occurs because of a lack of user-testing (see: Testing). When you get alerts (error messages) that tell you what is wrong, but offer no hint of what you should do to correct it, that's frustration. When you

get to a part of a tutorial that tells you to just press Open-Apple-P to print out your document, but fails to mention that you should have already spent two hours configuring the ports, that's frustration. It happens because the designer, programmer, and documentor have become so familiar with the programmer that they have forgotten their own learning problems. It is always overcomable with user-testing.

Trust is a most fragile commodity. For a user to trust your program, you must be consistent and absolutely honest. The program with the right answer 99 times out of 1ØØ is useless. The program which uses Open-Apple-E for Edit under all circumstances except that one undocumented one where it stands for Eradicate All Files will never be trusted again. If you say a document has been loaded, then the document should have been loaded. Your user will find you less than honest if he or she then removes the disk, only to be told that your program can't seem to find the document. You should try to make your program as safe an environment as possible (without frustrating the constant user), and you should partition off and clearly mark those operations that are of danger.

Next to a competent design, the most important attribute a program can display is a caring, polite, respectful attitude. The Apple II computer has a definite personality, as embodied in the tutorial disks and manuals supplied with the computer. Use these materials until you have grasped (intuitively) their flavor, and follow their lead. It is what your user feels comfortable with.

If you, in your personal and professional life, do not have to skills to be caring, polite, and respectful with people around you--and many great programmers do not--you probably should not be designing human interfaces. Team up with someone who is a good communicator and spend your time with the nuts-and-bolts issues on which you excel.

Putting the Concepts to Work

The last section covered theoretical aspects of some important underlying concepts. This section applies those concepts to cover practical design issues, such as selection of a metaphor (conceptual model), lowering the learning curve, and increasing productivity and salability.

The Metaphor

The human interface is an illusion: the pattern of light and darkness that your user perceives as "real" on the display is a careful contrivance of you, its creator. The quality of the human interface can often be measured by nothing more than the effectiveness of the chosen illusion.

Visicalc (TM) was the first serious microcomputer program that depended on a metaphorical illusion. The user was operating on a giant, classical business spreadsheet, seen through the limiting viewport of the monitor. The user could move the viewport around to see parts of the spreadsheet that were currently hidden. This illusion was particularly effective: it was familiar

to business users who had learned the paper version, it was perceivable by those who were sensory-oriented, and it was a simple model for the intuitive to grasp.

These guidelines are presenting specifications for two metaphors: the Macintosh windowing "desktop", and the AppleWorks hierchical "filecard" system.

Windowing software makes use of a more powerful, business-oriented metaphor: the desktop. The Apple II desktop metaphor offers the familiarity and perceptability of Visicalc (TM), but goes one step further in reducing the memory burden with pull-down menus instead of keywords. As more and more developers begin to publish software based on it, it will become the metaphor of choice for productivity tools.

The AppleWorks filecard system is a visually-perceptible version of a standard heirarchical menu structure. Tree-structured programs have historically depended on the user's forming an internal "map" of the program. Because of programmers' disproportionate ability to intuit such maps, we had no problem mapping things out, and considered those who did to be just a little slow. The fact is that three-quarters of the general population puts little faith in their intuitive abilities; they require direct, visual evidence of their location in a program. The visual presence of the filecards and the feeling of movement among them provide the external sense of program structure these people need.

Adults feel more comfortable with metaphors that do not require the user to travel around the program. The windowing metaphor, which brings program elements to the user, provides a much more secure, comfortable The windowing metaphor, which brings program elements to the user, provides a much more secure, comfortable environment than the filecard system. Our reason for supporting the filecard system is a purely practical one: many developers have existing software either on the Apple II or a competing system. It is an easy task to convert existing hierarchical software to the filecard system. The windowing metaphor generally requires an entirely fresh approach to problems and is better reserved for new projects.

Software for entertainment, education, home control, and other non-personal-productivity types of applications will often be more effective with another type of metaphor. Your creative skill in choosing appropriate metaphors will often dictate the success or failure of the entire project. Look for metaphors with real-life counterparts that are already familiar to your target audience. Once selected, carry over not only its static form, but its dynamic behavior.

Consider a home-control system. The obvious metaphor is the home. The user locates various lights and appliances on a top-view diagram of a home, then assigns on and off times to them, as desired. Now consider the most successful selling feature of a home control system: security. By turning lights on and off in some sort of sequence, you are supposed to create a "lived-in" look when you are away.

Typically, people try to program such a lived-in look by having the system turn lights on and off at random intervals. Anyone can spend $5Ø to $1ØØ on timers and duplicate random on/off timing; the real advantage to central control is the ability to simulate an awake, active person traveling from room to room. To effectively program this kind of static system, the user must first visualize a person moving from room to room, turning lights on and off as they go. Then, to make the computer actually simulate the result, they must painstakingly program in each and every on and off time.

Consider a more dynamic metaphor which has a little person living in the house, one who comes out when you leave home. You instruct the computer not what lights and appliances to turn on, but which ones you want left unaffected. You would also explain the general properties of each device: radios you play for hours, but never when the TV is on, and so forth. Then, when you leave home, the little person might wander from the kitchen where he ran the blender for a few seconds, turning out the lights as he went, into the living room, where he dims the lights and watches TV for five or ten minutes. Tiring of that, he might wander down the hall and into the master bedroom, dousing the lights after a few minutes and listening to some quiet music on the stereo. Every so often, of course, he would rise and go into the bathroom. Suffering from compulsive eating, he would also retrace his steps periodically and drop by the kitchen.

This dynamic home metaphor, then, provides a way that the average user can easily simulate an actively lived-in look that cannot be achieved by simple timers. The results will appear more "real" to the potential intruder than any amount of programming done with the static metaphor, because you will have far more dynamic randomness, and yet it will be tempered by the laws of nature and rules of human behavior patterns.

Metaphors are a product of perception, not judgement. We cannot set down a series of logic rules that will enable you to generate sound, creative ones. A good metaphor is a product of awareness and imagination. When you are beginning a project, spend time not only with the particular hardware, but with the people who will use it, in the environment where it will be located. Only then can you begin to coalesce all you see around you into a intuited pattern that will form an effective conceptual model.

The smooth, sleek model

Whatever metaphor you end up with, remember that it must be recognized, understood, and remembered. Programs that are evolved rather than designed with broad strokes at the beginning tend to end up with a myriad of cul-de-sacs and alleys, all places where the user can get lost. Spend the time in front in selecting or creating a model that can grow later on. Again, we urge that you use one of out two models specified in these guidelines unless there are compelling reasons not to: they have been designed to allow free expansion without major overhauls.

Whatever model you adopt, should you need to add features along the

way, you should step back and review the kind of impact it will have on the rest of the program. Will the user suddenly be confronted with an entirely different way to do a similar thing? Will every part of the program be menu-driven except this feature, where documentation wants you to use an undocumented command word because then they don't have to re-do the manual? Does someone else want to put a new feature in an inappropriate part of the program because they don't have to do as much re-coding that way?

It is a difficult but creative task to come up with a good, sleek, effective metaphor; it is a difficult and thankless task to try to defend it against creeping disarray. Spend the time in front making a structure that can be added to. Educate your writers in the political art of never giving a straight answer: a lead-in to a picture of the main menu should read, "it will look something like this...". Then, be prepared to be honest about the difficulty of changing a design for one of marketing's last-minute whims. And when something must be changed, review its effects on everything else.

Ease-Of-Learning and Ease-Of-Use

Often there is a trade-off between ease-of-learning and ease-of-use. Carefully balance your decisions: if the program is too difficult to learn, salesmen will not learn it and, thus, not sell it. If endless instructions and voluminous menus make it slow and cumbersome to use, people will get frustrated and tell their friends not to buy it.

There are several techniques and attributes that help keep the learning curve low and ease-of-use high, without seriously affecting power and performance:

Leveraged Learning

Make use of that which is already familiar to the user: when you design a program for an Apple computer, use the computer, run through the supplied tutorials, try out the most popular software. Then build an interface that is consistent with the personality of the machine. The single largest advantage of using either our filecard or windowing metaphor is that our users are already familiar with them: you can carry the user to a much higher level of sophistication with the subject matter of your program because your training material does not have to assume that the user doesn't know what the Open-Apple key does. Attempt to impose some foreign kind of implementation, such as function keys (Escape 1-9 or some other aberrant scheme) and you will spend most of your tutorial time trying to get people to learn everything they ever knew about their friendly little computer.

Make the flow of the subject matter follow the flow of the same operation when it was done manually. The spreadsheet was such an instantly successful metaphor because it simulated a system familiar to its target audience. It went beyond the manual system, in exhibiting continuously updated results, and so forth, but it made no effort to clash with the old knowledge of those who were expected to use it. In fact, it made every

effort to be as comfortable as possible. (Armed with hindsight, one can argue that the target audience went far beyond the halls of the business schools, and that spreadsheets are not familiar to the typical user of these programs. But this unexpected success should not detract from the fact that the program conformed to the old knowledge and expectations of its original target audience.)

Staged Learning

People can and do master some remarkably complex computer programs, such as AppleWorks (TM) and Apple Writer (TM). Other, far less powerful programs leave people so bewildered they often abandon even trying to learn them. With AppleWorks and Apple Writer, you learn a little bit and can then begin to do useful work; with these other programs, you must first learn virtually everything before you can do anything at all.

Design your programs and manuals so that a person can learn to do something useful within 30 minutes or less. Stage your learning out so that one can pick up tricks and shortcuts along the way, but needn't stumble over them at the beginning. Remember that you want a salesperson in a computer store to learn enough to be able to demo your program: with the thousands of programs on the market, he or she doesn't have more than half an hour to pick it up. And people won't demo something that is going to make them look like a fool.

It is all right to be redundant: in a word-processor, have a menu selection which turns on bold-facing, but also allow the experienced touch-typist to press CONTROL-B and get the same effect. In our blinking-bar input routine, the user quickly sees he can forward-delete by moving forward and then pressing Delete, but we have Control-F there for when he or she is ready to learn the "magic" forward-delete shortcut.

By sticking to your own and Apple's guidelines, by letting beginners do the most important things in the simplest, if not most efficient, way at the beginning, by considering the plight of the salesman who will promote this product for you, you are going to significantly increase the learnability and success of your product.

Novice/Expert modes

The first time you use a program you have quite different needs from the tenth time you use it: In the beginning, you need as much information presented as possible so that you can use the program with a minimum of learning. Later on, with a program you use habitually, you want speed and simplicity. You want only information pertinent to the specific task you are carrying out, not a lot of instructions on how to delete an incorrect response.

Most large programs now have some sort of utility/configuration section. The configuration sections often enable the user to select date and time formats, color vs. B&W, and select whether or not to have sound. In that section, you can also enable the user to select a skill level. The rest of the program can then use the resulting flag, when set to expert, to simplify verbiage and perhaps enable more flexible branching within the

program -- branching that would serve to get the novice into trouble but gives the expert the added flexibility she needs.

The skill level selection could be more sophisticated, perhaps with more than two levels, perhaps based on the type of user. For example, the same tax planner program might better bridge the gap between accountant and Apple owner if the accountant could select, "Expert at taxes, Novice at Apple" and the Apple owner could select "Novice at taxes, Expert at Apple". The possible combinations and permutations are truly boggling.

Simplicity

The contemporary microcomputer user still may have no previous experience with a program. Therefore, you must dedicate a significant fraction of the programming effort to the creation of an intuitively natural human interface. The program must, in the simplest way possible, anticipate the user's questions and needs and be prepared to answer and fill them the moment they arise. Once the user has become basically familiar with the human interface, if she guesses at an unknown response, she should be correct 95% of the time.

* Keep the external appearance of the program is as simple as possible. The user should not get lost within a maze of branches. (You may safely assume that the first-time user has not read the manual.)

* Keep the number of screens and menus to a minimum. One of each is best, as in the Apple II windowing software. The user cannot "get lost" because there is only one place.

* If you choose to make the user move, make that movement easy and fluid. Maintain a structure simple enough to allow the user to move from place to place without becoming confused.

* Keep displays clean and simple. People need redundancy and reinforcement, so don't create displays so starkly bare that people question their own understanding of what is going on. But do strive to make everything count: layout and graphic design should be tied into and supportive of the task being accomplished. Pose questions that are clear and free of ambiguity.

* Provide the user with the tools necessary to work with the program. For example, in a personal finance program, an input requesting annual rent should allow an answer such as 435.ØØ * 12 or 435.ØØ X 12, and not expect the user to work out the answer in his or her head. (Alternatively, you can provide a "desk accessory" calculator.) If a file name must be selected from the disk, display the valid names.

* Match the program to the skill level of the user. If you are doing a pricing program for a shopkeeper, do not ask her what her historic elasticity of demand has been without letting her know what it is and giving her the tools to estimate it. (Also, the question may be unnecessary: the fact that you asked it in a similar program you wrote for a Fortune 5ØØ company is no reason to ask it of a shopkeeper.)

* Lower memorization: Programs that are not used literally every single day will be forgotten: users will not remember command words, the names of their files, nor the fact that you are accepting data not with RETURN, but with CTRL-V (Violet was the name of your very first computer science teacher.) Computers are notoriously good at remembering the above type of information. Share it with your users: make sure the information needed is available where and when needed.

The flaw in the original Visicalc design was its dependence on the user recalling all the command mnemonics. Occasional users essentially had to re-learn the program every time they used it. While they had achieved high productivity and excellent ease-of-use, they were not careful to maintain ease-of-learning. Be aware that the average programmer and designer have above-average memories. Also, people engaged in the developement of a program spend an inordinate amount of time with it. You should continually find novice user for your program so you can track any increase in memorization burden, and you should have "occasional" users, so you can see if your structure and flow is visible enough that such users need not relearn.

* Honesty: Do not lie to your users. Do not say, "File loaded" when the file is not loaded, the name of the file has simply been selected.

Consistency

All programs written for a given computer should have as great a commonality as is practical. The purpose of these guidelines and standards is to achieve a level of consistency across all products designed to run on the Apple, a level that will make learning your product easy, while not stifling your ability to create the specific human interface best suited to your particular application.

All programs produced by a given software house should perform the same function in the same way. The same key sequence must not do the opposite thing in different products (Open-Apple-E = edit, Open-Apple-E = eradicate). Many software houses have their own guidelines, guidelines from which we drew in preparing this document. These individual guidelines tend to outline in far greater detail the program "personality" that the software house wants to project. If you have not yet put together such a document, may we suggest you do so. It is a very effective way to eliminate those interface battles that tend to occur about three days before release to production -- or three days after.

All software should be self-consistent: menu formats should be identical. If Control-F is enabled to forward-delete characters in one part of the program, it should forward-delete characters in all parts of the program. If you are working on a large project, be sure to spend enough time in team meetings being sure that everyone is on the same track -- all too often the three or four sections of a program end up with an entirely different "feel". At the same time, avoid rigidity: human interfaces must be tested on real people. The agreed-upon interface at the beginning will undoubtedly need changing, once you try it out on real people.

Speed

The user should be able to perform the desired task in as little time as possible, with the minimum complexity. Even such an obvious maxim as this becomes complicated: there are two very different kinds of time.

Objective time is the actual time it takes to accomplish an activity. Subjective time is the user's internal sense of that time. Basically, the more intellectual involvement, the less bored the user and the faster time seems to pass.

Several years ago, a skyscraper in Manhattan was built with too few elevators for all the people who worked in it. People complained bitterly about the long lines and long waits, but there seemed no solution. Consuting engineers found that the elevators could not be sped up, there was no way to push through a new shaft, there was no way to increase capacity. Finally, a designer was brought in, who looked over the problem, measured the wall spaces around the elevators, and showed up a few days later with huge floor-to-ceiling mirrors. The problem was solved! Instead of reducing objective time, the designer reduced subjective time. People still had to wait around, but now they had something to do--look at themselves and covertly look at each other.

Another example of this difference arose in tests with the mouse. Subjects were given a test where they repeatedly moved the mouse pointer to randomly chosen areas of the display. They consistently found that they could move the pointer faster with the cursor keys than with the mouse. However, when the videotape with its accompanying time-track was played back, it showed that the mouse was actually significantly faster. The difference lay in the far higher level of intellectual involvement the user had with the cursor keys.

One could conclude that cursor keys are better than mice because the user feels they are faster, even if they are not. But tests with a "real" word-processor and a real writing task revealed a countervaling rule: the higher the level of intellectual involvement the more the user is distracted from his or her intellectual task. Finding the mouse and moving it requires the use of only very primitive nervous centers. Making decisions on which of four keys to press and carefully watching for over-shoot ties down a much more sophisticated area of the brain that is thereby distracted from the task at hand. The flow of creativity becomes punctuated, and this constant distraction can seriously affect the quality and quantity of the task.

This degradation becomes even more serious as higher and higher levels of concious thought are required. Editors that allow more rapid movement through the user estimating the distances and typing in numerical "repeat-factors" are encouraging users to completely halt their train of thought while they carry out abstract computations. Systems that use voice, which requires one of the highest level of intellectual involvement, can distract people to the extent that they forget the task entirely.

So, in designing software, try to reduce subjective time to a minimum, but be careful that you are not doing it to the detriment of the user's ability

to perform the task. To reduce subjective time:

* Reduce objective time. Once a program is up and running, identify those parts that are perceivably slow and then do a design and code review. Before carrying out any of the following subjective time hints, look at what you can actually speed up through recoding or simplification.

* Speed up those parts of the program that are most obvious to the user. In particular, screen displays should be fast. If you have one letter to change, don't erase the whole display and write it over again. If you cannot work out a way to avoid updating the whole display, don't erase the old one: pad the new one out with blanks. At least the user won't be faced with the constant flashing.

* Break long operations into sections. During a long boot operation, put up a title page as soon as possible, so the user has something to do during the remainder of the boot.

* React to users' input immediately. A user will interpret any delay of more than a few tenths of a second after pressing Return or otherwise accepting to mean that either the program or the user has made an error. If you need to make a computation, first acknowledge that you have accepted the input.

  In training or educational software, it is doubly important to react immediately to test questions. The greatest retention of knowledge occurs when response occurs either within one second or not until the end of the entire test. Apparently, waiting five to ten seconds for a correct/not correct judgement is so frustrating that people lose involvement with what is going on.

* Carry out housekeeping functions during "dead" time, e.g., between keystrokes

* Tell the user how long you will be away if you are going for a while, so he or she can spend the waiting time doing something else

* Get all information needed before you go away, so the user needn't sit around to enter information during the process.

* Animate the display during long disk or printer operations. The simplest way is to display a growing line of periods. A better way is to display information that is more intellectually engaging, such as track and sector counts. The user needn't understand it or even look at it, but if he or she is bored, it provides something to do. A countdown clock is also a nice touch.

* Provide a beep when you come back, so the user needn't stare at the display to avoid loosing time in returning to the task.

We work in an industry where programmers often spend a significant portion of their days reading commic books during interminable compilations. Because we get so used to this kind of enforced bordom, we often visit it

upon our users. Increase your sensitivity to this important factor.

A Planning and Testing Methodology

Planning and the User Profile

In order to properly address the needs of the users, you must first know who they are and what their needs are. Software design should begin with a user-profile study. This study should cover the following three phases:

1. Select the target audience. Begin your human interface design by identifying your target audience. Are you writing for businesspeople or children? Will your audience consist of people relaxing at home or accountants under severe time-constraints? Are there several different types of people who will use your program? If so, you need to identify each.

2. Ascertain the level and limitations of their pre-existing knowledge. You should have an understanding of how much the target users know about:

   A. using the Apple II computer
   B. the general subject matter your program deals with.

3. Identify their needs. Once you have an understanding of the knowledge and limitations of the users, you can then figure out what types of information and level of support the the program will have to supply.

The following are mythical examples of two possible user-profiles for a program which fills the exact same function: a tax planner. Even though the task performed, the formulas used, the raw data required are identical, the programs that would result from the two user-profiles might bear little external resemblance:

Professional Tax Planner User Profile Study

User: CPA or Public Accountant

Anticipated knowledge of Apple computers: none. (The accountant may well have purchased the system just because of your program.)

Assumed knowledge of subject matter: Expert

Needs:

1. Staged learning curve. Must feel comfortable in a minimum time. Extended features can be picked up later.

2. Facility. Must be able to create and edit scenarios quickly. The windowing system should be considered first, as it enables the most freedom of movement (and looks the flashiest to the client).

3. Clear instructions and error messages. User may have never touched a computer before. Help should be aimed toward problems in the use of the system, rather than explanations of the difference between Short- and Long-term capital gains.

4. Professional appearance. Accountants will be using this package not only to help their clients, but to impress them. The vocabulary used on the display and in printed reports should be serious and professional. It may contain accounting jargon in areas that will not cause confusion to clients. The accountant must be protected against embarrassing errors (and alert messages); he may have a client sitting beside him.

5. Supplementary Features: accountants surveyed currently add or subtract amounts from the "accurate" figures produced by tax planners. Such items as a rough guesstimate of state tax liability may need to be figured into reports. Provide this facility.

6. Accountants are habitual users of adding machines: they may be expected to do all intermediate calculations on their own adder. No calculator need be provided.

Personal Tax Planner User Profile Study

User: John Q. Middle- to Upper-income Public

Anticipated knowledge of Apple computers: owner with some experience. (Research indicates that tax planning programs do not stimulate an initial computer purchase: people who already own the computer are buying the packages.)

Assumed knowledge of subject matter: None

Needs:

1. The prompting and documentation needs to be tutorial: the user must be guided into finding the necessary information to enter into the program, carrying out the kind of explorations with the program that will be most beneficial, and then suggest where the user should go from here.

2. Clear content verification and alert messages. "Unlikely" data should be confirmed by user. Help should be aimed toward problems in understanding the subject of taxes.

3. Appearance and use of accounting jargon. Non-professionals will be using this package. The vocabulary used on the display and in printed reports should be non-intimidating and not filled with accounting jargon.

4. User will probably only use the program a few times per year. There must be a minimum learning curve, even at the expense of reduced power and facility. A menu-driven format should be considered.

5. The user has to be asked for a lot of pre-computed figures: use an expression-evaluator input to allow them to add, subtract, multiply, and divide during input.

The "research" quoted in the above examples is ficticious -- do not start writing a tax-planner based on it. (The rest of the examples in this book are real.)

Carrying out an early investigation such as the ones above requires a minimum of time and can save you man-months of work later on. The reports need not be works of art; it is only important that every member of the design team has a clear picture of who the audience for this product will be. The user profile should be included in your Market Requirments Document, along with more prosiac information on market-shares, product penetration, competitive analysis.

If you have a marketing department, these reports should be their responsibility. They should carry out surveys, conduct focus groups, and otherwise collect good, solid information to answer the above questions. Then and only then will you be in a position to create a design responsive to the needs of the market.

Once the abstract report is done, you can develop a good mental image of the target audience by creating characters with names, occupations, family-lives, and dreams who collectively embody the breadth and depth of the audience. You may "make up" people who don't exist, or you can build composites out of people you know. This exercise gives people who have trouble holding on to abstract mental images a concrete, "living" representation of the users. It ensures that everyone on the design team is clear about who the users are, what they need, and what their expectations will be. In short, it gives everyone a stable, consistent focus. Even if you are a sole designer/programmer, you will find this to be a useful dicipline in forcing you to think all the way through the abilities and needs of the user. As the project goes along, you will replace your imaginary users with real ones, the subjects of your testing program.

Specifying the Human Interface

Once the target market is defined, you should design an appropriate human interface. The kind and sequence of steps to be followed are:

1. Identify and explore companion packages and competing packages. If 80% of the target audience owns and uses a given piece of software, one your proposed package is complementary to, it only makes sense to conform your package to the user interface of the other, already-familiar package. Competing packages give you a good grasp of what users already expect: unless your strength lies in marketing, your package should go beyond those already out. After all, they are probably preparing their own next generation.

2. Ferret out standards and guidelines. Get to know the computer on which you are developing. Read books like this. Look at the most popular packages. Make your design follow the philosophy you discover.

3. Select or design a metaphor. Make it familiar and make it intuitive. If you are doing personal-productivity software, we urge you to use the windowing interface. Not only have we spent millions of dollars researching, building, and testing it, but your users are familiar with it. If you are doing a different type of program, such as educational, select a metaphor these users will feel comfortable with and that is supportive of the task at hand.

Always keep in mind that this "illusion" need not be connected in any way with the hardware or operating-system requirements of the computer: the fact that loading and saving are companion calls to ProDOS does not mean they should appear together within a program. In terms of work-flow, they are usually at opposite ends. Avoid copying the interface of your favorite language or utility: there are a great many primitive interfaces that we become so used to we think they are good. They are not.

4. Writing the External Requirements Specification

This specification should represent everything the user will see. The

programmer's job will then be to translate this static report into the final, dynamic program. Cover every display, every help message, every alert. Eliminate, consolidate: when you find that two displays are almost identical, make them identical. At the beginning, ignore the difficulty of implementing wild new features. If you try to save coding time and space at first, you will lose sight of the illusion you are trying to create and get bogged down in the illusion your operating system is presenting. Later on, you can go back and be practical.

Even though you are writing a static document, you are designing something dynamic: maintain a mental model of the whole, and "run" the program, exploring the dynamic pathways. Essentially, a lot of this stage of design is done with intuition (see: Intuition). It explains why programmers depend on intuition so much more than the general population. What you are seeing in your mind's eye will only come to fruition perhaps months from now.

Once specified, build effective prototypes of new design features and test them for efficiency and acceptance. Do not wait until you have invested several man-years of development to discover that what seemed like such a hot idea just doesn't work in real life.

Do not overprotect: developers are, in some cases, making their software too friendly. Apple II users have to learn to work with some less-than-friendly concepts and constructs, such as ProDOS file names, Control keys, and technical words such as disk and memory. While you can, within your single program, shield your users from our less-friendly constructs and concepts, they are going to have to deal with them eventually. Such attempts merely result in their having even more learning to do: now they not only have to learn our methodology and jargon, but yours, too. We are committed to raising the general friendliness of our system, and the Apple II will become progressively easier to use, but we have to do everything in concert. If you are finding some part of our underlying interface to be particularly troublesome to your users, please let us know. Then we can work together to do something about it.

Have only one area of the screen active at a time: avoid prompting the user at the top of the display, echoing input characters in the middle, and displaying alert and help messages at the bottom, all at the same time. The user is going to be confused. An exception to this rule is in a point-and-choose scheme where options are being pointed to--for example, a cell in a spreadsheet--and actual entry occurs at separate, standard point on the display, such as line 24. 6.76.7 Keep Them Informed When You Are Away

When the computer will be either carrying out computations or accessing the disk for an extended period of time , a message should be left on the screen, instructing the user that the computer will return shortly (this is not the suggested message). Some periodic change in the screen, as a lengthening line of periods (hence periodic) should occur so the user knows the computer has not simply gone into an endless loop somewhere. When the computation period is over, clearly signal it: simply showing up with a blinking cursor over in the corner won't do

after a brief, 20-minute pause.

Part of the specification process is to estimate document requirements. Not only does this enable you to plan early for writing needs, but it keeps you conciously aware of the documentation costs of new features. If you figure it will take twenty pages to document some neat little shortcut, you are more likely to drop it early.

A final part is estimating scheduling. It is not within the scope of this book to cover scheduling, except to comment that invariably marketing, sales, and management want the project completed yesterday. A good rule of thumb for the real time that a project will take is to figure out how long it should take, double it, and go to the next higher time unit. In other words, if you can program this thing in three weeks (by programming 18 hours a day), then figure six months. This covers the five weeks your mother-in-law will visit, the total redesign after the coding is done, the publisher loosing the manuscript, and sales arguing they can't possibly sell it in that silly green package.

Always keep in mind that the last 10% of the program will require 50% of the time. Then walk the tightrope between the true time (above) and the time that everyone else will let you get away with. And best of luck.

The final function of a good External Reference Specification is to sell the design. Do not be afraid to explain why you have designed something the way you have. Remember that everyone everywhere is a self-proclaimed user-interface expert. The best way to keep from having to defend yourself and your design is to have dazzled everyone with your brilliant insights within the ERS.

Testing

Once the users have been profiled and a prototype built, it is time to begin testing.

Human interfaces are not made; they are evolved. Software designers are simply too close to their product, their computer, and have put up with the most abysmal interfaces themselves to be able to outguess the naive user. Products must be repeatedly tested on "real people". ("Real people" means the target audience: as soon as you find yourself sitting in a meeting with other computerists, all announcing what users will or will not feel/think/do, you are in trouble -- build the prototype and find out.)

The job of the designer is to do his or her best to predict the response of the user; the job of the user is to do just the opposite.

Human interface testing is quite different from the kind of exhaustive "boundary condition" testing used to uncover bugs. You should begin testing as early as possible, using drafted friends, relatives, and new employees, to uncover the really big holes in your design. As you get closer to a finished product, try it out on larger groups drawn from

the target population.

It is imperitive that the designers actually watch people use the program. Do not just send off copies of the program and expect written responses. Get the users and the designers in a quiet room together.

Our testing method is as follows: We set up a room with five to six computer systems. We invite groups of five to six users at a time to try out the systems (often without their knowing that it is the software rather than the system that we are testing). We have two of the designers in the room. Any less, and they miss a lot of what is going on. Any more and the users feel as though there is always someone breathing down their necks.

The initial ground rules are that no questions will be answered, as by the time the formal testing begins, we can supply a draft of the manual. (Usually by the second group, some glaring defects in the interface have shown up, and we have to give them help getting past the stumbling blocks.)

95% of the stumbling blocks found are found by watching the body language of the users. Watch for squinting eyes, hunched shoulders, shaking heads, and deep, heart-felt sighs. When a user hits a snag, he will assume it is his fault: he will not report it; he will hide it. Make notes of each problem and where it occured. Question the users at the end of the secession to explore why the problems occured. (You will often be surprised at what the user thought the program was doing at the time he got lost.)

We have found that prepared questionaires handed out at the end of a secession are of little value: you will seldom predict the problem areas before testing, and users will lie to spare everyone's feelings. (If you had figured out the problem areas, you would have already fixed them.)

Generally, two or three groups on one occasion is more than sufficient: patterns will emerge almost immediately. You should have at least one more bank of testing after any major revision; as the next example shows, one often jumps out of the frying pan, into the fire.

Herein follows a true anecdote which illustrates how difficult the most simple human interface issue can be, and why thorough testing on real people is so important.

As we tune in, the authors of the software, both of whom pride themselves on clever interface design, have anguished for hours over difficult passages in their program. It was to turn out their guesses were quite accurate in said difficult passages. It was the simplest question of all that caused all the problems...

Problem: in Apple Presents... The Apple IIe , the training program for teaching fundamentals of using the new Apple IIe computer, find out if the user is working with a color monitor.

User profile: new owner, customer in a computer store, or member of a class learning to use Apple computers.

Test user profile: customers in a computer store, non-computerists in a classroom environment, friends, and relatives.

First design: A color graphic would be displayed.
Prompt: "Are you using a color TV on the Apple?"
Anticipated problem: Those who were using a monochrome monitor in a classroom or computer store situation wouldn't know whether the monitor was black-and-white or was color with the color turned off.

First attempt: A color graphic was displayed.
Prompt: "Is the picture above in color?"
Failure rate: 25%

Reason: As anticipated, but incorrectly overcome, those seeing black and white thought their color might be turned down. They didn't answer the question wrong; they turned around and asked one of the authors whether the monitor in question was color or not. A decision was made that the authors could not be suppied with the disk:

Second attempt: A smaller graphic with large-letter words in their own vivid colors was substituted: GREEN BLUE ORANGE MAGENTA
Prompt: "Are the words above in color?"
Failure rate: color TV users: none
black and white monitor users: none
green-screen monitor users: 1ØØ%

Third attempt: the graphic remained the same.
Prompt: "Are the words above in more than one color?"
Failure rate: color TV users: none
black and white monitor users: 16%
green-screen monitor users: 5Ø%
Reasons: the black and white monitor users who answered incorrectly admitted that they did so on purpose. (Our methods for wringing their confessions shall remain proprietary.) 5Ø% of the green-screen folk considered that they were looking at both black and green -- two colors -- and answered the question accordingly.

Fourth attempt: Same display of graphic and colored text
Prompt: "Are the words above in several different colors?"
Failure rate: color TV users: none
black and white monitor users: 20%
green-screen monitor users: 23%
Reasons: By this time, the authors were prepared to supply everyone who bought an Apple with a free color monitor, just so we would not have to ask the question. It turns out that around 20% of the people were not really reading the question. They were responding to:
"Are the words above, several different colors?"

Fifth attempt: Same display of graphic and colored text
Prompt: "Do the words above appear in several different colors?"
Failure rate: none.

In case it appears the authors were simply dull fellows, be it known that this was a fully-interactive training program in excess of 100K, and this was the only interface issue that required more than one correction. It clearly exemplifies how even the most careful designers can totally miss when guessing at how users are going to respond.

Had the designers not tested the program, it is probable that dealers would not have used the program in their showrooms, as they would have wearied of telling potential customers that they were/were not using a color TV and that the Apple Presents... Apple program was being very stupid to ask the question like that. (Potential customers, of course, wouldn't fall for such an explanation: they know it was the computer that asked the question and that one should always buy the computer that asks good questions.)

It is vital that programs be tested early and often with users from the target audience; this testing should be an integral part of any testing plan. This testing seems like a lot of extra effort. In practice, it really isn't, beyond the mechanical difficulties of getting your equipment and test group together. (Computer stores, colleges, and shoppping centers are often good random-testing locations.) The above testing cycles took only four days: the first two days were on-site, using new Apple employees. Only two days of testing required any set-up work at all, and the over-all improvement to the product was clearly worth the effort.

Even if the interface had not changed at all, it would have been worth it just to be able to ward off all the self-proclaimed experts with their (day-after-going-to-production) comments of "Boy, I sure wouldn't have done that this way. A lot of people out there are gonna have trouble." What joy to turn to such people and announce with a clear conscience, "Well, we tried it out on 109 people, and they all sailed through with flying colors."

High-budget Testing

You can hire market research firms to gather test subjects and conduct

focus groups for your program. Such efforts are usually beneficial, always fun, and invariably expensive. If you are going to use outside services, pick the most beneficial times: at the beginning and end of a project. At the beginning, focus groups can help you plan for the needs of the user. At the end, you can judge the results and "fine-tune" a design appropriately. Where you should not use expensive research techniques is in the middle of the project:

The first time you test a program on real people, it is almost certain to be an immediate disaster. Trying to gather feedback on how a user feels about the fine points of your design is a little difficult when he or she has just accidentally destroyed the contents of the disk. It is like asking test subjects driving your new car at freeway speeds to comment on ambient noise conditions when they have already noticed that you forgot to build in a braking system: somehow they always seem distracted. You can test a program for basic "survivability" with almost anyone; save expensive in-depth research for the point when people can use your product effectively.

Part II:

The Apple II

Generic Human Interface

Apple II Human Interface Guidelines

330

There are two primary hardware environments on the Apple II: the original Apple II and Apple II +, and the newer Apple IIe, IIc, and so forth. The new computers have user interface support in the form of a more complete keyboard and a special character set referred to as MouseText. (This character set was introduced after the first Apple IIe's; at the time of this writing, it was expected that the vast majority of IIe owners would avail themselves of the enhancement kit that includes it.)

We offer two primary software environments: the windowing (Macintosh-like) and the filecard heirarchical menu system. Unlike the Macintosh system, Apple II windowing software need not require a mouse.

These guidelines are based on the minimum configuration available on all Apple II computers beginning with the Apple IIe. Descriptions of keyboards, standard key definitions, and special display characters refer to these later computers. If you are working in a specialised market segment that needs to cater to the older machines, you will have to pick and choose those guidelines that are appropriate to the older equipment. In general, you can adapt the overall philosophy of these guidelines, and then directly use such standards as the standard input routine.

The hardware

The following sections introduce the Apple II keyboard, mouse, and display.

THE KEYBOARD

The Apple II keyboard, unlike the Macintosh keyboard, is used for entering both text and commands.

The keys on the keyboard are arranged in familiar typewriter fashion. The U.S. keyboard is shown in Figure 3.

Figure 3. The Apple II U.S. Keyboard

There are two kinds of keys: character keys and modifier keys. A character key sends characters to the computer; a modifier key alters the meaning of a character key if it's held down while the character key is pressed.

Character Keys

Character keys include keys for letters, numbers, and symbols, as well as the Space bar. If the user presses one of these keys while entering text, the corresponding character is added to the text. Other keys, such as the Tab, Return, Delete, and Escape keys, are also considered character keys. However, the result of pressing one of these keys depends on the application and the context.

The Tab key is a signal to proceed: It signals movement to the next item in a sequence. Tab often implies an Enter operation before the Tab motion is performed.

The Return key tells the application that the user is through entering information in a particular area of the document, such as a field in an array. Most applications add information to a document as soon as the user types or draws it. However, the application may need to wait until a whole collection of information is available before processing it. In this case, the user presses the Return key to signal that the information is complete. Return must do the job of both Enter and Return in the Apple II world. On the Macintosh, Enter accepts the information but causes no movement, whereas Return accepts the information, then moves down and to the left. On the Apple II, you

must consider what your users will expect. In two-dimentional arrays, have Return enter the information and move directly down one row. This gives the user the Tab key to enter and move right and the Return key to enter and move down. In text processing, have Return accept the information and move down and to the left in the traditional way.

Return and Space dismiss dialog and alert boxes (see "Dialogs and Alerts").

Delete is used to delete text or graphics. The exact use of Delete in text is described in the section on text editing.

Modifier Keys: Shift, Caps Lock, Control, Open-Apple, and Solid-Apple

There are five keys on the keyboard that change the interpretation of keystrokes: two labeled Shift, one labeled Control, one labeled Caps Lock, one labeled with the "Open-Apple" symbol, and one labeled with the "Solid-Apple" symbol. These keys change the interpretation of keystrokes and sometimes mouse actions. When one of these keys is held down, the effect of the other keys (or the mouse button) may change.

The Shift key chooses among the characters on each character key. Shift gives the upper character on two-character keys, or the uppercase letter on alphabetic keys.

Caps Lock latches in the down position when pressed, and releases when pressed again. When down it gives the uppercase letter on alphabetic keys. The operation of Caps Lock on alphabetic keys is parallel to that of the Shift key, but the Caps Lock key has no effect whatsoever on any of the other keys.

Pressing a character key while holding down the Control or either Apple key usually tells the application to interpret the key as a command, not as a character (see "Commands").

Control keys are reserved for functions that the user must do repeatedly with little or no concious thought. You will find standard definitions for most of them in the "Commands" section. Because these definitions remain standard throughout applications, the user has only an initial learning burden, and, since all these definitions are either short-cuts or very advanced features, the user can pick them up at their leasure.

The Open-Apple key has only a few reserved functions (see "commands"). As a general rule, it is available for your special mnemonic commands for your specific application. It is also used in conjunction with the mouse for extending a selection (as the Shift key is on a Macintosh); see "Selecting".

The Solid-Apple key generally mimics the action of the Open-Apple key. Users have been found able to learn one mnemonic per letter, for example, E for Edit. Defining Open-Apple-E to mean Edit and

Solid-Apple-E to mean something else, such as E for "Eradicate This Document" invariably leads to problems. The user must have some powerful rule which logically separates all Open-Apple combinations from all Solid-Apple combinations, if you are to use them separately.

If you want to enable your user to define keyboard macros, tie them to the Solid-Apple key: the powerful rule here is that the program owns the Open-Apple key and the user owns the Solid-Apple key.

Typeahead and Auto-Repeat

If the user types when a windowing application is unable to process the keystrokes immediately, or types more quickly than a toolkit can handle, the extra keystrokes are queued, to be processed later. This queuing is called typeahead. There's a limit in each toolkit to the number of keystrokes that can be queued, but the limit is usually not a problem unless the user types while the application is performing a lengthy operation.

The toolkits can be operated in two modes: with interupts or passive. When interrupts are on, queuing is done "automatically"; in passive-mode, queuing is done periodically, as described in the toolkit manuals. Remember in testing that the toolkits will shift to passive-mode automatically on an Apple IIe with no mouse card installed (as the interrupts are generated by the mouse card). So be sure to test your software for mouseless operation on a IIe without a card plugged in!

When the user holds down a character key for a certain amount of time, it starts repeating automatically. An application cannot tell whether a series of n keystrokes was generated by auto-repeat or by pressing the same key n times. Therefore, be sensitive to the discovery of a vast number of identical keystrokes—your user may have erred in holding the key down too long.

Holding down a modifier key has the same effect as pressing it once. However, if the user holds down a modifier key and a character key at the same time, the effect is the same as if the user held down the modifier key while pressing the character key repeatedly.

Versions of the Keyboard

There is only one current physical versions of the Apple IIe and Apple IIc keyboard. The standard layout on the European version is designed to conform to the ISO (Internation Standards Organization) standard; the U.S. key layout mimics that of common American office typewriters. European keyboards have different labels on the keys in different countries, but the overall layout is the same.

Reserved Key Combinations

Some characters are reserved for special purposes.

One Open-Apple keyboard command is reserved:

Character	Command
?	Help

Other Open-Apple keyboard equivalents are conditionally reserved. If an application enables these commands, it shouldn't use these characters for any other purpose, but if it doesn't, it can use them however it likes:

Open-Apple combinations:

Character	Command
P	Print
Q	Quit
S	Save

Windowing applications reserve these additional commands (see: Commands)

Character	Command
Z	Undo
X	Cut
C	Copy
V	Paste

(Note that these keys are the first four on the bottom row on the standard U.S. keyboard. If you translate a program to a keyboard with a different layout, you should change the actual characters typed so that they remain the first four keys on the bottom row.)

D	Drag or move the currently active window
G	Grow or shrink (size) the currently active window
M	Mark a selection

Control combinations--all environments:

Character	Command
B	Bold
C	Copy
D	Delete
E	Edit (toggle insert/overstrike)
F	Forward Delete
* H	Left Arrow
* I	Tab
* J	Down Arrow
* K	Up Arrow
L	Begin or End Underline
* M	Carriage Return
P	Print the contents of the screen
S	Save
* U	Right Arrow
V	Paste
X	Cut
Z	Undo
* [	Escape

* These are the control equivalents of the various Apple special keys. Current unmodified Apple II keyboards cannot differentiate between a Control-character sequence and its equivalent special key, for example, Control-M and Return.

Keys to Ease Foreign Translation

If you are designing your software for ease of translation into foreign language, please keep in mind that the following key characters are different in the "local" text character sets and keyboard layouts of international versions of Apples:

Different now: # @ [,] ` { | }

(Could change in the future: $ ^)

To allow foreign-language entries, you need to provide a number of "dead-keys" so users can properly punctuate. We have defined a standard set of keys:

OPEN-APPLE-^
OPEN-APPLE-'
OPEN-APPLE-`
OPEN-APPLE-

Typing any of these combinations will place the accent, followed by a backspace character, in your file, so that the next character typed will be accented. Display the accent and the character linked by the solid-dash MouseText character ("S"). As the user cannot type a solid dash, there is no ambiguity.

Your program should accept only valid accented characters, throwing away the dead-key character if, for example, a person types OPEN-APPLE-^ followed by an X.

THE MOUSE

The mouse is a small device the size of a deck of playing cards, connected to the computer by a long, flexible cable. There's a button on the top of the mouse. The user holds the mouse and rolls it on a flat, smooth surface. A pointer on the screen follows the motion of the mouse.

The mouse is reserved for windowing and entertainment applications only.
It should not be installed into a menu program for purposes of advertising "mouseability".

A complete description of the actions and activities of the mouse may be found the description of the mouse within the Mouse Guidelines section.

Software Standards

Input

The standard Apple II input routine is common across all Apple II series computers. There are added capabilities with the mouse within the windowing metaphor, but keyboard consistency is still maintained. All professional programs should be using this input. It is available as a tool to all registered developers, with a BASIC, Pascal, and Assembly Language front end.

The user should be able to tell the rules of the input from the kind of cursor being displayed. Users are confused when the computer speaks to them in a different way in each program, but they are confounded when the computer "understands" them differently in each program.

Of all the standards and guidelines presented in this book, this is the most important: use the standard input in exactly the standard way. If you need to use an entirely different kind of input scheme, select a different cursor character, and train your users to recognize it as yet another entity. If you wish to add to its capabilities, do so, but never twist the pre-existing definition. We have trained all new users with the tutorial material shipped with each computer. They know what the input looks like and expect it to always work the same.

Input Routine Standard Keys:

Keystroke	Editing Operation
Necessary:	
Left-Arrow	moves cursor left within input
Right-Arrow	moves cursor right within input line.
Control-D	deletes character to the left of the cursor position
Delete	deletes character to the left of the cursor position
Control-E	toggle between insert and replace (discussed below)
Control-F	deletes character forward (to the right) of the cursor position
Return	accepts entire response, regardless of current cursor position.
Control-X	deletes all characters on the input line (or all characters marked with mouse).
Control-Y	deletes all chars from present cursor position to end-of-line.
Control-Z	recalls display of default response. If no default, then it acts the same as Control-X.

Notes:

Typing any printing character will automatically insert that character into the input line at the current cursor position.

Pressing Return with the cursor anywhere within the input line will accept the entire input.

Default responses are displayed with the cursor at the beginning or end of the response. Pressing Delete (or Control-D) as the first character will delete the entire response. Once any other key has been pressed, Delete and Control-D revert to their standard definitions.

The blinking-bar cursor is, theoretically, a vertical bar that lies between two characters, representing an insertion-point. Because of hardware limitations in the text mode of the Apple II, text-based applications use a blinking underscore that alternates with the character that is to the right of the theoretical position. Thus, what in graphics environments looks like this:

A turnbuckle
A turnbuckle

in text-mode looks like this:

A t_rnbuckle
A turnbuckle

The Overstrike Cursor

You may also provide an overstrike capability, through Control-E. When pressed, it changes the appearance of the cursor from a blinking bar to an inverse-color square over the character to the right of the insertion point. As characters are entered, the inverse box moves to the right, replacing the original character with the new character, neither shrinking nor expanding the

size of the line. With this single exception, all keys and features work the same.

Using the standard input:

More specific guidelines for the windowing interface will be found in that section. While the windowing guidelines do not clash with the following information, they do go beyond it in power and performance. If you are working with the windowing software, refer to the appropriate sections for more specific information.

The program input statement asks the user for information by displaying a verbal prompt. Prompts should terminate in a colon (:) or greater-than sign (>) if a statement, a question-mark (?) if a question. The prompt is followed by 2 spaces on an 8Ø-column display, 1 space on a 4Ø-column display.

A default answer may be displayed, with the cursor following, in which no field length is denoted. If there is no default response offered, or the default is rejected by the user, the program can display a finite input field with a series of "ghost" underlines (MouseText character "I"). This character is a shortened underline with every other dot turned off. Since the user cannot type it, there can be no abiguity.

Leading and trailing spaces should be routinely stripped from input lines, unless they are specifically needed.

Keystroke errors are best trapped immediately: if you are accepting a number, do not accept a letter such as "A" or "B".

An example of the input:

What is a "drift"?

> A whole lot of cattle_

(Consider the underline to be blinking -- the printer was not able to quite capture the effect.) The user wants to change the answer to read:

> A herd of cattle

To edit the response, the user first moves back to the end of the word "lot," using the Left-Arrow. It looks like this:

> A whole lot of cattle_

The user now moves the cursor to the left by pressing the Left-Arrow.

> A whole lot of cattle
> A whole lot of cattl_

Because the cursor alternates with the character to the right of the theoretical insertion point, that character is invisible half the time. In the rest of the sequence, we shall assume that we are looking during the time that the character is invisible and the cursor is visible.

```
> A whole lot of catt_e

> A whole lot of cat_le

> A whole lot of ca_tle

> A whole lot of c_ttle

> A whole lot of _attle

> A whole lot of_cattle

> A whole lot o_ cattle

> A whole lot _f cattle

> A whole lot_of cattle
```

The user then presses the Delete key several times, until the words "whole lot" have been deleted:

```
> A _of cattle
```

Next, the user types the word "herd":

```
> A h_of cattle

> A he_of cattle

> A her_of cattle

> A herd_of cattle
```

Finally, the user presses Return to accept the entire response:

```
> A herd of cattle
```

Additional detailed specs for the blinking-bar cursor:

```
Blink-rate:                 8Ø cycles per minute
During 1 blink:
   Time showing bar:                1/3
   Time showing normal character:   2/3
```

For the overstrike cursor:

```
Blink-rate:                 8Ø cycles per minute
During 1 blink:
   Time showing normal character:  1/3
   Time showing inverse character: 2/3
```

Whenever the cursor is moved, start the blink cycle over again, first showing the bar.

The input buffer: You should maintain an input buffer larger than the field length of the input, with a pointer showing how much of it you should allow to be visible. Let's say you have a field length of 25 and the user has typed in:

```
What do you use if you want to turn left?  A right-hand turn signal_
                                           A right-hand turn signal.
```

(In order to show the two phases of the blinking cursor, each example shows the user-response, both while the bar is showing and the character "under it" is showing.)

The user suddenly realizes the error of his answer and backtracks:

```
What do you use if you want to turn left?  A _ight-hand turn signal.
                                           A right-hand turn signal.
```

So far, the contents of the input buffer have not been changed. Now the user types in the correct answer:

```
What do you use if you want to turn left?  A left_ight-hand turn sig
                                           A leftright-hand turn sig
```

Instead of either not allowing the user to enter any more characters, or shoving the "nal" part of "signal" off into oblivion, move all the characters ahead in your 25Ø+ character input buffer. So, internally, you are now carrying the answer:

```
A leftright-hand turn signal
```

with a pointer that tells you only to display to the "g" in "signal". Now, when the user uses the right-delete key (CONTROL-F) to delete the word, "right", you can again show the characters that had been hidden:

```
What do you use if you want to turn left?  A _eft-hand turn signal..
                                           A left-hand turn signal..
```

When the user presses RETURN, accept only those characters that are visible: this buffer is just there to make changes easier. You need not maintain a full 25Ø+ character buffer if you only have short input fields. Try to have an input buffer at least twice as long as the longest field, and dump characters off the right end if the user keeps backing up and inserting: don't ever have the input simply lock up. A CONTROL-Y or CONTROL-X should clear to the end of the actual input buffer, not just the visible portion.

Cursor Movement with no action taken

Sometimes programs such as word processors require pure cursor movement with no action taken. The standard keys in such cases are as follows:

Keys for up, right, down, and left motion:

Apple IIe and newer computers: the four arrow keys

Apple II and Apple II+:

```
          I=up
J=left    K=right
     M=down
```

These keys are often prefixed with an ESCape.

Keys for vertical, horizointal, and diagonal motion:

All Apple II computers:

```
U=up,left   I=up   O=up,right
      J=left    K=right
N=down,left  M=down   ,=down,right
```

These keys are often prefixed with an ESCape.

Alerts

Every user of every application is liable to do something that the application won't understand. From simple typographical errors to slips of the mouse to trying to write on a protected disk, users will do things an application can't cope with in a normal manner. Alerts give applications a way to respond to errors not only in a consistent manner, but in stages according to the severity of the error, the user's level of expertise, and the particular history of the error. The two kinds of alerts are beeps and alert boxes.

Beeps are used for errors that are both minor and immediately obvious. For example, if the user tries to DELETE past the left boundary of a text field, the application could choose to beep instead of putting up an alert box. The beep should not be a standard, Apple II bell, but a more gentle tone, as found in ProDOS and AppleWorks--the whole room doesn't need to know the user has yet again made a fool of him or herself. A beep can also be part of a staged alert, as described below.

An alert box looks like a modal dialog box, except that it's somewhat narrower and appears lower on the screen. An alert box is primarily a one-way communication from the system to the user; the only way the user can respond is by clicking buttons. Therefore alert boxes might contain dials and buttons, but usually not text fields, radio buttons, or check boxes. Figure 27 shows a typical alert box.

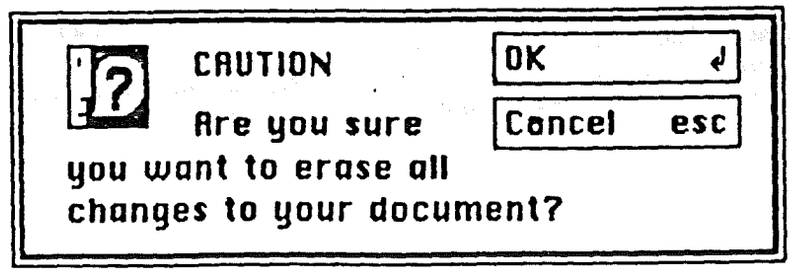


Figure 27. An Alert Box

There are three types of alert boxes:

- Note: A minor mistake that wouldn't have any disastrous consequences if left as is.

- Caution: An operation that may or may not have undesirable results if it is allowed to continue. The user is given the choice whether or not to continue.

- Stop: A situation that requires remedial action by the user. The situation could be either a serious problem, or something as simple as a request by the application to the user to change diskettes.

An application can define several stages for an alert, so that if the user persists in the same mistake, the application can issue increasingly more helpful (or sterner) messages. A typical sequence is for the first two occurrences of the mistake to result in a beep, and for subsequent occurrences to result in an alert box. This type of sequence is especially appropriate when the mistake is one that has a high probability of being accidental. An example is when the user chooses Cut when the selection is an insertion point.

Under no circumstances should an alert message refer the user to external documentation for further clarification. It should provide an adequate description of the information needed by the user to take appropriate action. Avoid at all costs such messages as:

> Application error #1463

Error messages should not only provide information (in the user's native tongue -- not computerese) as to what the error was, but should offer solutions as to what the user can do to correct the situation. A better message might be:

> The program here requires the name of the file you want to work from. You have not yet selected a file. Please type the name of one of the above files first.

So, generally, it's better to be polite than abrupt, even if it means lengthening the message. The role of the alert box is to be helpful, make constructive suggestions, and to help the user solve the problem, not to give an interesting but academic description of the problem itself.

Error-trapping

In most situations, user inputs must be checked for validity. Account numbers, employee numbers, and dates are just a few examples of items that should be checked to see if the data requested is on file or plausible. Numeric inputs should be screened for values too small or too large, if extreme values are invalid or potentially damaging to the program.

Many types of errors can be circumvented through software design: If, in testing, you find users repeatedly making the same kind of errors, change the software.

Be careful of the details, both during design and boundary-testing: for example, make your program insensitive to upper/lower case when no distinction is necessary, and test your program to make sure it is making no such distinction anywhere. This is a good example of design

"smoothing": if the user finds any input anywhere that is case-sensitive and you have not taken pains to make sure they know that this is an exception, they must assume that any input may fail at any time if they are not careful about case. If you have two or three "tail-ends" like this going on at once, the user will become very frustrated.

Enable only those keys you have informed the user you are enabling. Do not prompt: "Press Escape for Main Menu, Return to continue:" and fail to announce that Space bar will eliminate this afternoon's files. The classic counter-example of this was in an early Apple text editor with a verified-replace option. According to the manual (no instructions were displayed), R meant replace this occurrence, Space bar signified do not replace this occurrence. The actual code was such that any character with an ASCII value of 82 (R) or above caused a replacement, and any character with an ASCII value less than 82 caused a skip. Therefore, "[" would replace, "8" would not, "^" would replace, "," would not.

The best way to make an alert message understandable is to think carefully through the error condition itself. Can the application handle this without an error?

When you find errors occurring during user-testing, think through the problem: is there some way you can either lower the incidence of the error or eliminate it altogether? Many programs seem to suffer from the it's-easier-to-flag-an-error-than-correct-it error:

Syntax error: no comma after Aardvark

Is the error so specific that the program can handle it transparently? If not, is the error specific enough so that the user can fix the situation? What are the recommended solutions? Can the exact item causing the error be displayed in the alert message?

Many error problems can be eliminated quite readily by fairly simple design-changes, and you will usually end up saving memory by eliminating all the text necessary for the alert message. If you can't eliminate an error, then think through what can be done to lower the incidence: every new user should not be experiencing the same error. If they are doing so, you need to reconsider the design.

Specific graphic designs of filecard alert boxes and Windowing alert boxes can be found within those two sections.

Part III:

The Filecard Menu Interface

The Filecard Menu Interface

Introduction

The primary interface for future software development on the Apple II series is the windowing (mouse) interface, as adapted from Macintosh. Because not all software lends itself to this metaphor, we also support a hierachical menu structure, using a filecard metaphor. Adapt menu interface if:

1. you are transporting an existing menu-based application onto the Apple II.

2. you have an application in an area such as education that does not lend
   itself to the desktop metaphor.

Do not reject the windowing system because you do not want to require people to buy a mouse: the Apple II windowing system requires no mouse. Our design goal was a windowing system that, without a mouse, would be as functional as the filecard system. This goal was achieved in the Spring of 1985 with the introduction of this specification and the toolkits to support it.

[Illustration of filecard metaphor]

People have often become lost in menu systems because of the lack of visual feedback. The filecard system enables the user to see very clearly exactly where she or he is at any point. It allows a depth of four menu levels, a depth consistent with Apple II owners' navigational abilities, and discourages disconnected series of sub-menus, a practice that thoroughly confuses users.

The actual filecards are most easily created on Apple IIe and later computers. If you are writing for the Apple II/II Plus market, use the structure and program flow, but not the visual design. If you are writing for 4Ø-column, you may find you are lacking width for your menu items because of the space taken up by the filecards themselves. If so, either shorten the item names or abandon the filecards, again maintaining all but this visual element.

Menus

The filecard menu scheme uses the general Macintosh paradigm of point-and-choose, rather than typing. It still allows the standard method of responding to a menu, by having the user type in the number (or letter) of choice. But it also allows the user to use the arrow keys to move from choice to choice. Selecting an up or down arrow key and leaning on it until the right selection appears requires a lower level of intellectual involvement on the part of the user. The result is that the user need not disengage as much from the task at hand to use the menu tree.

The following illustration shows a typical menu tree with the user at the second level. Note the use of MouseText icons: the filecard indicates that selecting this option will take the user to another filecard one level deeper. The running man indicates an immediate activity.

```
Retail Store Manager                                   Escape: Main Menu
____________________________________________________________________________
 ___________________________
|Main Menu                  |
|  ________________________ |_________________________
|  |Inventory Control      |                          |
|  |                       |                          |
|  |  1.  Enter new product category                  |
|  |                                                  |
|  |  2.  <UPDATE INVENTORY FOR AN EXISTING PRODUCT>  |
|  |                                                  |
|  |  3.  Work up explanation of last month's shrinkage
|  |                                                  |
|  |  4.  Calculate current value                     |
|  |                                                  |
|  |__________________________________________________|
|___|
____________________________________________________________________________
Type number or use arrows, then press Return

[end of display]
```

Upon the user's moving to a lower level, the display would look as follows. Note that item or document names are indicated by a lack of icons. Ddisabled menu items are bracketed with the delete checkerboard character (#127) so the user knows what is currently valid. Jump over such items when the user moves up and down with the arrow keys.

```
Retail Store Manager                                  Escape: Inventory Control
____________________________________________________________________________

 Main Menu
   Inventory Control
     Update Inventory

       1.     Leisure Suits
       2.     Nehru Jackets
     ▓ 3.     Daggers and Automatic Weapons                                ▓
       4.     Quail Eggs
       5.     The little paper poofs you put on chicken feet
       6.    <MISCELLANEOUS OIL TANKERS>
       7.     Cocktail Glass Parasols

____________________________________________________________________________
Type number or use arrows, then press Return
```

[end of display]

This scheme allows a menu structure four levels deep, with no ambiguity as to where one is, how one got there, and how one gets back.

The current selection (starting with a default) is highlighted, but its number appears in the input field only if typed by the user. When highlighting such a menu, highlight only the option itself, not the number, letter, or any icons preceding it.

In 80-column mode, where a properly-working monitor is expected, the highlighting is done by displaying the selection in inverse-mode. Print the

Print the actual selection (not the associated number or letter) with a leading and trailing inverse space. These spaces make for easier readability of the first and last characters.

In the case of a 4Ø-column display, if you assume a TV may be in use, surround the current selection with < > and convert all lowercase characters to uppercase. (We did considerable experimentation with various schemes to highlight without inversing, including double-brackets, a slow, software flash, changing the indentation of the current selection, pointing arrows, etc. This scheme, in testing, proved to be the most readable and intuitive of the lot, so we have chosen it as a standard for all this type of 4Ø-column point-and-choosing.)

Either typing a new number or pressing the Up- or Down-Arrow changes the highlighted selection immediately. Moving by arrow causes the input field to be cleared of any number that may currently appear. If a number is currently in the input prompt and the user wants to type in a new number, she need not press the Delete key (or Control-D) before typing a different response. The new number will replace the old. Delete will, however, delete the number in the input field in the normal fashion, leaving the current selection still highlighted, and it can still be selected. (We found you must leave it highlighted so that the user knows what the "anchor" selection is when pressing the Up- or Down-Arrow key, and since it is highlighted, it must be selectable.)

All sub-menus enable the user to move to the next higher menu by pressing the Escape key. The main menu has no Escape-key option, so that the user can feel confident about leaning on the Escape key to get all the way back to the top level, without worrying about then being bounced completely out of the program. The Main Menu's last option enables the user to "quit": end the program.

Programs should always have an definite, clearly-labeled way out. Those of you who have dealt with new users of micro-computers who are running endless programs have probably seen their extreme discomfort at not being able to find their way out. Survivors of time-share days will often panic as they search in vain for a way to "log off" while their mental clocks tick away the dollars.

Even if your program is on a copy-protected disk and there really is no way out, give the user an End option and then tell him that he may now insert another disk and press RETURN, or whatever. Users feel positively trapped by programs with seemingly no end; they forget that the power switch solves all.

The Filecard Metaphor Without Filecards

The following example was taken from the original Apple IIc Utilities program. It was designed to work in either 4Ø- or 8Ø-column mode and is shown here in 4Ø-column operation.

[Substitute with actual screen-shot for final]

```
System Utilities                    Main Menu
____________________________________________

Work on Individual Files:

  1.  Copy Files
  2.  Delete Files
  3. <RENAME FILES>
  4.  Lock/unlock Files

Work on an Entire Disk:

  5.  Duplicate a Disk
  6.  Format a Disk
  7.  Identify and Catalog a Disk
  8.  Advanced operations

  9.  Quit: Exit System Utilities

Type a number or press | or | to select an option.  Then press Return.
____________________________________________ Help: Press Open-Apple-?
```

[end of display]

Using the help facility

In the following example, the user points to item 6 on the above menu and then presses Open-Apple-?. S/he is then presented with the following help:

[Screen-shot of IIc utilities with item 6, Format a Disk help displayed.]

When you put the help in a dialog box, make sure that you show only one set of prompts: the current one. Do not have some random, "Press Return to Format all your disks" left over in one part of the display, while exhorting the user to press Return to go back to the menu in your inset box. You will end up with a frightened user.

If you want to mark menu items previously selected by the user, do so with an asterisk. We use this system in our in-box tutorial materials, so your users will have been exposed to it before.

For more information about help dialog boxes, see: Help below.

Menus: numbers vs. letters

We have shown the menu items sequentially numbered. Numbering has the advantage of not requiring the user to be typewriter-literate, an important consideration when writing for young children. You may want to work out a mnemonic lettering scheme, but if you do so, use first letters only and do not repeat any letter more than once anywhere in the program. You should also consider future growth: most of the truly horrifying mnemonic systems started out small, but as the software evolved, they got out of hand.

Sequentially numbered menus should display the number followed by a period and two spaces. If you are using icons, follow the icon by two spaces:

1. Eat
2. Drink
3. <BE MERRY>

The highlighting scheme does not work well on numbered menus with more than 9 items: as it turns out, neither do people. If you have this many items, you should separate them into two or more categories and create more menus.

Sequentially lettered menus are usually quite difficult for non-touch-typists to handle, but should you use them, use the same format as for numbered menus:

A. Do this
B. <DO THAT>
C. Do the other

Mnemonically lettered menus display hyphens instead of periods and look like this:

C - Create layout
P - <PRINT-OUT LAYOUT>
B - Bill the customer

Note that there is an extra space before the hyphen, but there are still two spaces after it, to allow room for the < or inverse space. Again, only the option itself is highlighted.

How to write a menu entry.

Menu entries should be written so a novice can understand them, but an expert need read only a few keywords at the beginning. The examples below are a bit wordy, but illustrate the point: (the underlining shows the keywords an expert will use—it would not actually appear on the screen.)

1. Load a file from disk into memory.
2. Edit the file currently in memory.

3. Print the current document on the current printer.
4. Change printers: select a different printer to print your document.
5. Save the current file on disk.

The most fundamental design element for the filecard metaphor is the three regions defined by the two solid-horizontal lines. These three regions should appear on every display in the program, on any Apple II computer in any mode. As simple as such an element is, it gives the user a visual anchor-point.

The exact number of lines devoted to the three regions is not cast in stone: the real standard being striven for is that there be three regions with solid lines separating them, that these be devoted to titles, choices presented, and instructions. (The Apple II and Apple II+ can not produce a solid line in text mode; use either their hyphens or their short-underline characters.

The title region can have up to three titles (usually two in forty-column mode). The middle title (or left title, if only 2) should be the name of the display, and if it is a menu, it should contain the word, "menu". The other displays you will use, such as data-entry and information, will have a similar format, so make sure your user is clearly aware of what he is being asked to do: use a properly descriptive title and do not use the word menu. Similarly, on such information screens, do not number itemized lists; bullet them: otherwise, about 25% of your users will try to type in a "selection". All displays except the main menu should have the words:

Escape: [name of display]

in the top, left-hand corner, so the user knows where she or he will go by pressing it.

You may use or not use other titles as you see fit, but they should have a consistent meaning throughout a given application.

Choosing an Option

Use for confirmation questions and choosing among three alternatives or fewer. This is the horizontal version of the scheme used on menus.

```
Is the above information correct? <YES> no
Do you want to delete the old file? <NO> yes
Select fill-pattern for printed graph: Cross-hatch <DOTS>
Solid
```

In forty column mode (shown), the standard (default) selection is bracketed and uppercased. In 8Ø-column mode, it is displayed in inverse.

Pressing either Y or N on a confirmation question moves the pointer (highlighting) to that word, Yes or No. Pressing Return then accepts the selection. In any selection set where each word starts with a unique letter, allow the user to type that letter. Do not allow wrapping: if the user is on the left-most answer, require her to use the Right-Arrow key to get to the right-most answer. The reason for this seemingly unfriendly rule will be made clear below.

How to Ask Confirmation Questions Safely

One of the problems with confirmation questions is that the user's response eventually becomes entirely automatic. The danger in this is that when you really need confirmation of a dangerous situation, the user idly selects Y Return, just as always. The following guidelines will help overcome that problem:

1. Do not ask for confirmation when it is not needed--most important.
2. If destruction is involved, default to the least-destrutive option.
3. Do not ring the bell for confirmation questions asked every time: save the bell for unusual cirmstances. 4. Place the default answer first in the list, unless an error in the user's choice can result in catastrophic damage.

The user's pattern of use will thus be that accepting the default means a simple Return, and rejecting the default means Right-Arrow Return. This automatic pattern will always make the computer work, except in one case:

> This disk has active files. Reformatting will destroy them.
> Do you want to re-format and destroy all files? Destroy <CANCEL>

In this case, the user must break normal pattern and can only destroy what may be one month's work by pressing Left-Arrow Return, or typing a letter other than Y or N. (This is the reason for not allowing wrap-around, which would let the user press the Right-Arrow key--the "habit key".

There is one cardinal rule that must be followed to make this sheme work: Do not harrass the user. If there is an activity which must be habitually handled, you must allow the user to fail. Excessive prompting leads people to totally ignore the meaning of every prompt in their efforts to escape from your clutches. Then, when something really important arises, they will bang their way through it without even looking at the words, destroying exactly what you were trying to keep them from destroying.

These kinds of difficulties arise out of the most altruistic of motives; they will show up when you begin to do long-term testing with people from your target audience. Get your program into test sites as early as possible, and listen to user-feedback on just these sorts of issues.

Marking Groups of Selections:

Quite often, particularly in file-related functions and options, such as printer option screens, you have a group of names or options which the user needs to select or deselect, turn on or turn off.

[show with MouseText check-marks in place of -->'s]

Fred's Utilities Copy a File Escape: Exit to Main Menu

.d2/ FRED MODIFIED:

	Name:	TYPE:	SIZE:	DATE
	ZILLA.TEXT	TEXT	1 BLOCK	3-AUG-85
	SHAWN.TEXT	TEXT	2 BLOCKS	4-AUG-85
	HOUSE.FOTO	BINARY	5 BLOCKS	29-JUL-82
	LAZARUS	SYS	1 BLOCK	17-APR-85
	SHERI.FORMS	TEXT	2 BLOCKS	6-AUG-85
-->	RUPERTS.LIST	TEXT	4 BLOCKS	3-JUL-85
	RODS.NOTES	TEXT	1 BLOCK	12-JUN-85
	JDS.MISC.	TEXT	14 BLOCKS	6-AUG-85
	AMY.MEMO	TEXT	4 BLOCKS	14-JUL-86
	THAD.MEMO	TEXT	23 BLOCKS	18-JUL-85
-->	PETER.MEMO	TEXT	2 BLOCKS	3-AUG-85
	LEE.MEMO.3	TEXT	2 BLOCKS	3-AUG-85
	LEE.MEMO.4	TEXT	1 BLOCK	4-AUG-85

To Move: Press arrow keys To Mark/Unmark Documents: Press Solid-Apple
To Accept Marked Documents: Press RETURN.

Help: OPEN-APPLE-?

[end of display]

If there is not enough room on the display for all names, scroll the display when the pointer (highlighting) reaches the bottom. When there are hidden file names, display a note at the bottom (or top, when files are hidden above) that says:

(Additional file names)

Because of the lack of special keys on the Apple II and Apple II Plus, there has never been a standard way of doing selections such as these. Use your imagination, and make the design you come up with conform to the rest of your program.

"Press Return to continue"

The user controls the movement from one display to the next by pressing

the Return key (or, optionally but consistently, Space bar). He is informed by a message such as, "Press the Return key to go on to the menu." on the bottom line of the screen. (Delay loops are difficult to judge as to the proper duration, and become somewhat insulting to the intelligence of the user.) The actual prompt message should give some indication as to what will happen next, rather than simply saying "Press Return to continue."

The educational software community has pretty much selected Space bar instead of Return to control movement: children were found to occasionally press Reset by accident on the older Apple II's and Apple II+'s. Please be consistent in your choice of Return key or Space bar, not only within a given program, but across your complete product line.

Do not tell the user to, "Press any key". On the Apple II series computers, you cannot currently read every key by itself: Reset, Shift, Control. We have also found in testing that new users panic when asked to press any key. Over 80% of them will turn around and say, "but what key should I press?" In questioning them about this response, we discovered that they are quite convinced that even though the prompt implied all keys were O.K. to press, some could be dangerous. Of course, they were usually quite right.

While you should not tell them to press any key, you may, in this specific case, accept more than the key specified. Both Return and Space bar can be accepted, even though only one is prompted for: users grow used to using one or the other. The exception to this lies in alert messages: use Space bar for dangerous, unusual alerts rather than Return. The habitual user will attempt to clear most alerts with scarcely a glance, but when Return fails to clear it, she or he will be forced to look further.

Never accept Escape instead of Return or Space bar, unless the latter two keys will result in the same thing the user would expect of Escape: moving up one level.

Arrays and the Filecard Metaphor

Displays with several input statements:

* Movement from input to input is sequential: the user may move back and forth but not randomly skip around. (The exception is the spreadsheet sort of array, where the user can use the four arrow keys.)

* Pressing the Tab key automatically positions the user at the next input statement.

* Pressing Open-Apple-Tab automatically positions the user at the previous input statement. The prior response to the previous input will be displayed as that input's default.

* The last input on the display will normally ask if the user has completed all responses to her or his satisfaction.

* No input will be accepted without the user explicitly terminating it, usually with Return, Tab or Open-Apple-Tab. The fact that the user has used up all the spaces available in the field should not automatically move the user to the next question.

Alerts

An alert box is a narrow rectangle that appears low on the screen. An alert box is primarily a one-way communication from the system to the user; the only way the user can respond is with Return, Space, Escape, or perhaps Y or N. Figure 27 shows a typical alert box.

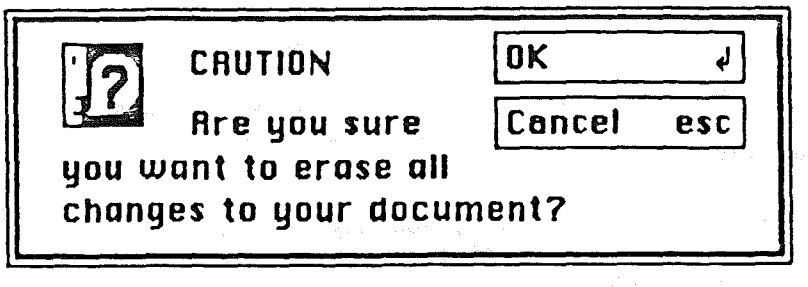


Figure 27. An Alert Box

The prompt in an alert box depends on the nature of the box. If the box presents the user with a situation in which no alternative actions are available, the box has a single prompt that says either "Press Return" or "Press Space bar". In this context, pressing the proper key means "I have read the alert."

Space bar is more apt to cause the user to read the contents of the alert; users press Return so automatically, they sometimes forget to look at what they are acknowledging. If your program has numerous displays that use Return to continue, you may want to use Space for each and every alert. Otherwise, reserve Space only for those "Stop" occasions (see: Alert Messages in the Generic Interface section) when the user is in danger of doing something large, permanent, and probably unexpected. If the user is given alternatives, then typically the alert is phrased as a question that can be answered "yes" or "no", although some variation such as Save and Don't Save is also acceptable.

For further information on beeps, the types of alert messages, how and when to write one, read Alert Messages in the Generic Interface section.

Help

The user should not be faced with page after page of instructions: experience has proven that people simply will not read them. Rather, supply help as it is needed. One way of doing that is described above in the section on menus.

When you try your program out on new users, be sensitive to the times they need fundamental help in using the features of the programs. For example, while you may have a program portion with detailed explanations on why ellipsoid analysis is so effective in figuring hog belly futures, your user may never get there: you may not have provided necessary help in how to enter preliminary data.

The standard help key on the Apple IIe and later model Apple II computers is OPEN-APPLE-? and, optionally, SOLID-APPLE-? (The SHIFT should not be required: therefore, also accept OPEN-APPLE-/ and SOLID-APPLE-/.)

The standard help key on the Apple II and Apple II+ is a question mark or slash, with no modifier key.

A help dialog box is a rectangle that appears higher on the screen than an alert box. A help box is also primarily a one-way communication from the system to the user; the only way the user can respond is with Return or Escape. Figure 27 shows a typical help box.

Figure 27. A Help Box

The prompt in a help box will normally either say "Press Return for more information" or "Press Return to continue," depending on whether there are more pages of information. You may also enable Open-Apple-Return to back up through the pages, and you should enable Escape in all cases to cancel the help function. In an education program in which you are consistently using Space bar instead of Return for "page-turning," you can use "Press Space bar".

Vocabulary

Jargon

Avoid computer jargon. A great deal of it has an unrelated emotional charge. (Abort, for example.) The appendix to *How to Write a Manual* has a comprehensive list of standard terms.

Abbreviations

Use abbreviations only where absolutely necessary or where an abbreviation is better understood than what it stands for, e.g., 8 PM.

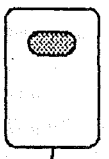
Defaults

Please do not ever use the word default in a program designed for humans. Default is something the morgage went into right before the evil banker stole the Widow Parson's house. There is an exhaustive list of substitutes (previous, automatic, standard, etc.) in the Appendix to *How to Write a Manual*.

Defaults should be declared, not assumed. Undeclared (not displayed) defaults such as pressing RETURN for Yes (or for No?) will cause confusion and anger. You need not declare ESCAPE every time you enable it: ESCAPE always gets you out of where you are, to where you came from, without causing damage or confusion. As long as that benign definition is adhered to, you may feel free to slip in ESCAPE anywhere.

Part V

The Desktop Interface



ABOUT THIS MANUAL

This section describes the Apple II windowing user interface.

The windowing user interface consists of those features that are generally applicable to a variety of applications. Not all of the features are found in every application. In fact, some features are hypothetical and may not be found in any current applications.

The best time to familiarize yourself with the windowing user interface is before beginning to design an application. Good application design happens when a developer has absorbed the spirit as well as the details of the user interface.

Before launching into your own design, you should have read this manual and have some experience using one or more applications, preferably one each of a word processor, spreadsheet or data base, and graphics application. If you are beginning early enough that such applications are not redily available on the Apple II, then use such applications on the MacIntosh.

INTRODUCTION

Apple II windowing software is designed to appeal to an audience of nonprogrammers, including people who have previously feared and distrusted computers. To achieve this goal, Apple II windowing applications should be easy to learn and to use. To help people feel more comfortable with the applications, the applications should build on skills that people already have, not force them to learn new ones. The user should feel in control of the computer, not the other way around. This is achieved in applications that embody three qualities: responsiveness, permissiveness, and consistency, leading to the user's having a sense of autonomy.

Responsiveness means that the user's actions tend to have direct results. The user should be able to accomplish what needs to be done spontaneously and intuitively, rather than having to think: "Let's see; to do C, first I have to do A and B and then...". For example, with pull-down menus, the user can choose the desired command directly and instantaneously. This is a typical operation: The user moves the pointer to a location on the screen and presses the mouse button.

Permissiveness means that the application tends to allow the user to do anything reasonable. The user, not the system, decides what to do next. Also, error messages tend to come up infrequently. If the user is constantly subjected to a barrage of error messages, something is wrong somewhere.

The most important way in which an application is permissive is in avoiding modes. This idea is so important that it's dealt with in a separate section, "Avoiding Modes", below.

The third and most important principle is consistency. Since users usually divide their time among several applications, they have historically felt confusion and irritation as they faced learning a completely new interface for each application. The main purpose of this manual is to describe the shared interface ideas of windowing applications, so that developers of new applications can gain leverage from the time spent developing and testing existing applications both for Macintosh/Lisa and the Apple II.

With the MouseText and MouseGraphics windowing toolkits available from Apple, consistency has become an achievable goal. However, you should be aware that implementing the user interface guidelines in their full glory often requires writing additional code that isn't supplied.

Of course, you shouldn't feel that you're restricted to using existing features. The Macintosh/Apple II world is a growing system, and new ideas are essential. But the bread-and-butter features, the kind that every application has, should certainly work the same way so that the user can move easily back and forth between applications. The best rule to follow is that if your application has a feature that's described in these guidelines, you should implement the feature exactly as the guidelines describe it. It's better to do something completely

different than to half-agree with the guidelines.

Illustrations of most of the features described in this manual can be found in various already-released Macintosh and Apple II applications. However, there is probably no one application that illustrates these guidelines in every particular. Although it's useful and important for you to get the feeling of the user interface by looking at existing Macintosh and Apple II applications, the guidelines in this manual are the ultimate authority. Wherever an existing application disagrees with the guidelines, follow the guidelines.

Avoiding Modes

> "But, gentlemen, you overdo the mode."
>
> -- John Dryden, The Assignation, or Love in a Nunnery, 1672

A mode is a part of an application that the user has to formally enter and leave, and that restricts the operations that can be performed while it's in effect. Since people don't usually operate modally in real life, having to deal with modes in computer software reinforces the idea that computers are unnatural and unfriendly.

Modes are most confusing when you're in the wrong one. Unfortunately, this is the most common case. Being in a mode is confusing because it makes future actions contingent upon past ones; it changes the behavior of familiar objects and commands; and it makes habitual actions cause unexpected results.

It's tempting to use modes in a windowing application, since most existing software leans on them heavily. If you yield to the temptation too frequently, however, users will consider spending time with your application a chore rather than a satisfying experience.

This is not to say that modes are never used in windowing applications. Sometimes a mode is the best way out of a particular problem. Most of these modes fall into one of the following categories:

- Long-term modes with a procedural basis, such as doing word processing as opposed to graphics editing. Each application program is a mode in this sense.

- Short-term "spring-loaded" modes, in which the user is constantly doing something to perpetuate the mode. Holding down the mouse button or a key is the most common example of this kind of mode.

- Alert modes, where the user must rectify an unusual situation before proceeding. These modes should be kept to a minimum.

Other modes are acceptable if they meet one of the following requirements:

- They emulate a familiar real-life model that is itself modal, like picking up different-sized paintbrushes in a graphics editor. MousePaint and other palette-based applications are examples of this use of modes.

- They change only the attributes of something, and not its behavior, like the boldface and underline modes of text entry.

- They block most other normal operations of the system to emphasize the modality, as in error conditions incurable through software ("There's no disk in the disk drive", for example).

If an application uses modes, there must be a clear visual indication of the current mode, and the indication should be near the object being most affected by the mode. It should also be very easy to get into or out of the mode (such as by clicking on a palette symbol).

Several features of the keyboard (mouseless) interface are modal. For example, the cursor keys are usually redefined, along with Escape and Return being used as mode-terminators. However, every effort has been made to limit both the extent of this modality to only these keys, and to be consistent in the kind of behavior changes that the user can expect.

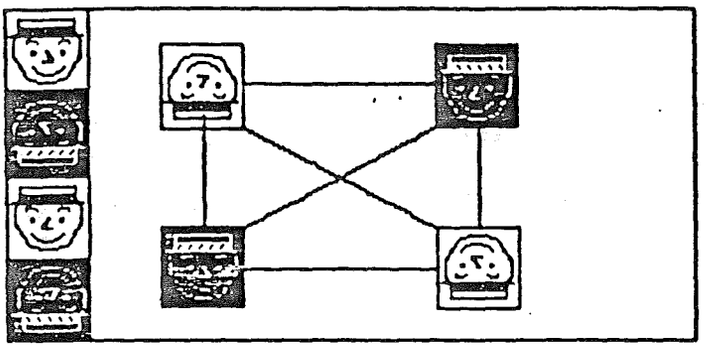
TYPES OF APPLICATIONS

It's useful to make a distinction among three types of objects that an application deals with: text, graphics, and arrays. Examples of each of these are shown in Figure 1.

The rest to some faint meaning make pretence
But Shadwell never deviates into sense.
Some beams of wit on other souls may fall,
Strike through and make a lucid interval;
But Shadwell's genuine night admits no ray,
His rising fogs prevail upon the day.

MacFlecknoe Page 1

Text



Graphics

Advertising	132.9	
Manufacturing	121.3	
R & D	18.7	
Interest	12.2	
Total	285.1	

Array

Figure 1. Ways of Structuring Information

Text can be arranged in a variety of ways on the screen. Some applications, such as word processors, might consist of nothing but text, while others, such as graphics-oriented applications, use text almost incidentally. It's useful to consider all the text appearing together in a particular context as a block of text. The size of the block can range from a single field, as in a dialog box, to the whole document, as in a word processor. Regardless of its size or arrangement, the application sees each block as a one-dimensional string of characters. Text is edited the same way regardless of where

it appears.

Graphics are pictures, drawn either by the user or by the application. Graphics in a document tend to consist of discrete objects, which can be selected individually. Graphics are discussed further below, under "Using Graphics".

Arrays are one- or two-dimensional arrangements of fields. If the array is one-dimensional, it's called a form; if it's two-dimensional it's called a table. Each field, in turn, contains a collection of information, usually text, but conceivably graphics. A table can be readily identified on the screen, since it consists of rows and columns of fields (often called cells), separated in graphics environments by horizontal and vertical lines. A form is something you fill out, like a credit-card application. A form may not be as obvious to the user as a table, since the fields can be arranged in any appropriate way. Nevertheless, the application regards the fields as in a definite linear order.

Each of these three ways of presenting information retains its integrity, regardless of the context in which it appears. For example, a field in an array can contain text. When the user is manipulating the field as a whole, the field is treated as part of the array. When the user wants to change the contents of the field, the contents are edited in the same way as any other text.

Another case is text that appears in a graphics application. Depending on the circumstances, the text can be treated as text or as graphics. In MousePaint, for example, the way text is treated depends on which palette symbol is in effect. If the text symbol is in effect, text can be edited in the usual way, but cannot be moved around on the screen. If the selecting arrow is in effect, a block of text can be moved around, but it cannot be edited.

USING GRAPHICS

The MouseGraphics toolkit gives full access to the Apple II high-resolution graphics screen. To use this screen to its best advantage, MouseGraphics applications use graphics copiously, even in places where other applications use text. As much as possible, all commands, features, and parameters of an application, and all the user's data, appear as graphic objects on the screen. Figure 2 shows some of the ways in which applications can use graphics to communicate with the user.

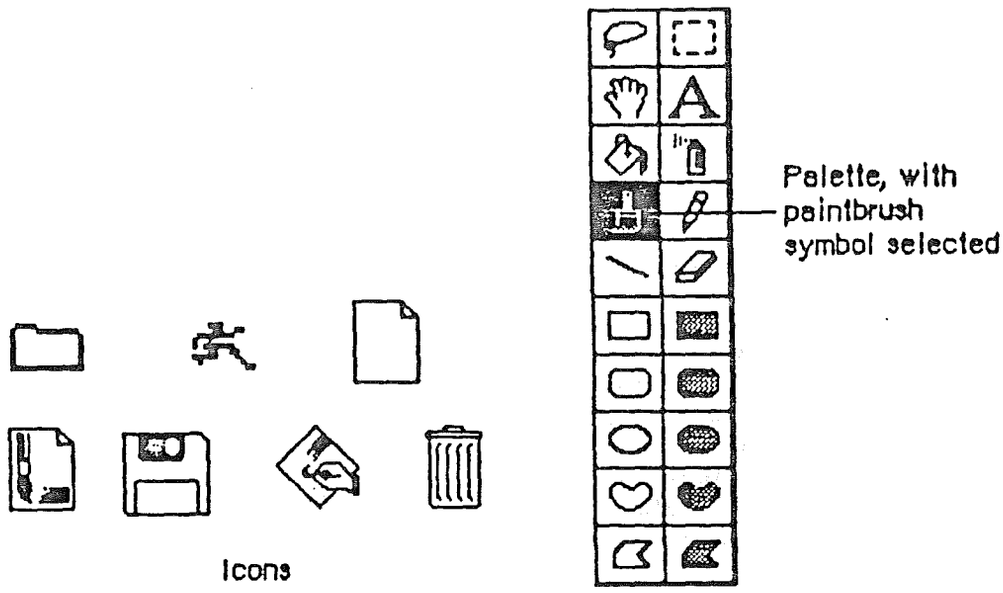


Figure 2. Objects on the Screen

Objects, whenever applicable, resemble the familiar material objects of which they are symbolic.

Objects are designed to look good on the screen. Predefined graphics patterns can give objects a shape and texture beyond simple line graphics. Placing a drop-shadow slightly below and to the right of an object can give it a three-dimensional appearance.

Generally, when the user clicks on an object, it's highlighted to distinguish it from its peers. The most common way to show this highlighting is by inverting the object: reversing its black and white pixels. In some situations, other forms of highlighting, such as the knobs used in MacDraw, may be more appropriate. The important thing is that there should always be some sort of feedback, so that the user knows that the click had an effect.

One special aspect of the appearance of a document on the screen is visual fidelity. This principle is also known as "what you see is what you get". It primarily refers to printing: The version of a document shown on the screen should be as close as possible to its printed version, taking into account inevitable differences due to different media. The ability to achieve visual fidelity in the Apple II world is not as great as that in the Macintosh world: we have more varied printer technologies to support, we cannot depend on as much available memory, and we have a slower processor. Still, the primary reason for choosing to use the MouseGraphics toolkit over the Mousetext toolkit is increased visual fidelity. We should therefor be as thorough and clever as possible in maximizing that fidelity.

Icons

A fundamental object in windowing software is the icon, a small graphic object that is usually symbolic of an operation or of a larger entity such as a document.

Icons should be sprinkled liberally over the screen. Wherever an explanation or label is needed, first consider using an icon instead of using text as the label or explanation. Icons not only contribute to the clarity and attractiveness of the system, they don't need to be translated into foreign languages.

Palettes

Some applications use palettes as a quick way for the user to change from one operation to another. A palette is a collection of small squares, each containing a symbol. A symbol can be an icon, a pattern, a character, or just a drawing, that stands for an operation. When the user clicks on one of the symbols, it's distinguished from the other symbols, such as by highlighting, and the previous symbol goes back to its normal state.

Typically, the symbol that's selected determines what operations the user can perform. Selecting a palette symbol puts the user into a mode. This use of modes can be justified because changing from one mode to another is almost instantaneous, and the use can always see at a glance which mode is in effect. Like all modal features, palettes should be used only when they're the most natural way to structure an application.

A palette can either be part of a window (as in MacDraw), or a separate window (as in MousePaint). Each system has its disadvantages. If the palette is part of the window, then parts of the palette might be concealed if the user makes the window smaller. On the other hand, if it's not part of the window, then it takes up extra space on the desktop. If an application supports multiple documents open at the same time, it might be better to put a separate palette in each window, so that a different palette symbol can be in effect in each document.

COMPONENTS OF THE WINDOWING SYSTEMS

This section explains the relationship among the principal large-scale components of the windowing systems (from an external point of view).

The main vehicle for the interaction of the user and the system is the application. Only one application is active at a time. When an application is active, it's in control of all communications between the user and the system. The application's menus are in the menu bar, and the application is in charge of all windows as well as the desktop.

To the user, the main unit of information is the document. Each document is a unified collection of information--a single business letter or spreadsheet or chart. A complex application, such as a data base, might require several related documents. Some documents can be processed by more than one application, but each document has a principal application, which is usually the one that created it. The other applications that process the document are called secondary applications.

The only way the user can actually see the document (except by printing it) is through a window. The application puts one or more windows on the screen; each window shows a view of a document or of auxiliary information used in processing the document. The part of the screen underlying all the windows is called the desktop.

At the time of this writing, we have not created tools for making a Macintosh-like Finder to change applications. With such a Finder active, if the user double-clicks on either the application's icon or the icon of a document belonging to that application (or opens the document or application by choosing Open from the File menu), the application becomes active and displays the document window. If you are using the MouseGraphics environment for a integrated software package, you still might want to emulate the Macintosh Finder. If you are writing a text-based integrated application or hard-disk filing system, please contact Apple II technical support to find out what kind of metaphor we are using/recommending.

Internally, applications and documents are both kept in files. However, the user never sees files as such, so they don't really enter into the windowing user interface.

THE KEYBOARD MOUSE

At the time of this writing, the majority of Apple II owners did not have a mouse. While this situation was expected to change with the arrival of more and more mouseware, developers felt a need for an interface for text applications that would be fully functional without a mouse. The Apple II windowing interface has been developed to be a powerful, practical tool with or without a mouse.

Figure 3A. The Twin Speres

Conceptually, the Keyboard and Mouse interfaces exist as overlapping spheres, with many operations, such as typing text, in common. They differ in how the non-mouse user performs the various pointing-and-choosing operations:

- Choosing from a menu: pressing Escape takes the user to the menu, the cursor keys moves around the menu, Return accepts the current item, and Escape cancels. See: "The menu bar".

- Moving and sizing a window: Open-Apple-D for drag redefines the cursor keys to move the window; Open-Apple-G for grow redefines the cursor keys to grow or shrink the window. See: "Moving a Window" and "Changing the size of a Window".

- Selecting text: Open-Apple-M for mark begins a text-selection mode. Moving the cursor keys marks the text, Return accepts, Escape cancels. See: "Selecting Text".

- Clicking on controls: Solid-Apple, by itself, acts as a mouse button to click on buttons, check boxes, and radio buttons. See: "Controls"

- Moving the insertion point: Pressing the cursor keys moves the insertion point. See: " Selecting".

As much keyboard support as is practical has been installed within the various mouse toolkits; where you must provide your own, follow the direction and philosophy of these guidelines and the toolkits themselves. When using a toolkit-based application, the keyboard user can directly emulate a mouse by holding down the Open-Apple key and pressing, then releasing Solid-Apple. The cursor keys then affect the mouse cursor so the user can move around the display, using Solid-Apple as the mouse button to click, press, or drag. When the Open-Apple key is released, mouse emulation is terminated.

The mouseless mouse is included as a safety net should a developer fail to discover in testing that there is a necessary feature that the keyboard user cannot get to without a mouse. It is also allows for rather flashy live demonstrations of your software's independence of the mouse. It is not intended as a substitute for the proper design of a keyboard-only interface: it was designed for ease of learning rather than ease of use, and to lower memory and documentation requirements.

THE MOUSE

The mouse is a small device the size of a deck of playing cards, connected to the computer by a long, flexible cable. There's a button on the top of the mouse. The user holds the mouse and rolls it on a flat, smooth surface. A pointer on the screen follows the motion of the mouse.

Simply moving the mouse results only in a corresponding movement of the pointer and no other action. Most actions take place when the user positions the "hot spot" of the pointer over an object on the screen and presses and releases the mouse button. The hot spot should be intuitive, like the point of an arrow or the center of a crossbar.

Mouse Actions

The three basic mouse actions are:

- clicking: positioning the pointer with the mouse, then briefly pressing and releasing the mouse button without moving the mouse

- pressing: positioning the pointer with the mouse, then holding down the mouse button without moving the mouse

- dragging: positioning the pointer with the mouse, holding down the mouse button, moving the mouse to a new position, and releasing the button

The Mouse Toolkits can provide "mouse-ahead"; that is, any mouse actions the user performs when the application isn't ready to process them are saved in a buffer and can be processed at the application's convenience. The application can then choose to ignore saved-up mouse actions, but should do so only to protect the user from possibly damaging consequences.

Clicking something with the mouse performs an instantaneous action, such as selecting a location within the user's document or activating an object.

For certain kinds of objects, pressing on the object has the same effect as clicking it repeatedly. For example, clicking a scroll arrow causes a document to scroll one line; pressing on a scroll arrow causes the document to scroll repeatedly until the mouse button is released or the end of the document is reached.

Dragging can have different effects, depending on what's under the pointer when the mouse button is pressed. The uses of dragging include choosing a menu item, selecting a range of objects, moving an object from one place to another, and shrinking or expanding an object.

Some objects, especially graphic objects, can be moved by dragging. In this case, the application attaches a dotted outline of the object to the pointer and redraws the outline continually as the user moves the pointer. When the user releases the mouse button, the application redraws the complete object at the new location.

An object being moved can be restricted to certain boundaries, such as the edges of a window frame. If the user moves the pointer outside of the boundaries, the application stops drawing the dotted outline of the object. If the user releases the mouse button while the pointer is outside of the boundaries, the object isn't moved. If, on the other hand, the user moves the pointer back within the boundaries again before releasing the mouse button, the outline is drawn again.

In general, moving the mouse changes nothing except the location, and possibly the shape, of the pointer. Pressing the mouse button indicates the intention to do something, and releasing the button

completes the action. Pressing by itself should have no effect except in well-defined areas, such as scroll arrows, where it has the same effect as repeated clicking.

Multiple-Clicking

A variant of clicking involves performing a second click shortly after the end of an initial click. If the downstroke of the second click follows the upstroke of the first by a short amount of time, and if the locations of the two clicks are reasonably close together, the two clicks constitute a double-click.

Because of the difficulty of detecting time-between-clicks on the Apple II, it is permissible to define double-clicking simply as two clicks geographically close together and with no intervening events. Its most common use is as a faster or easier way to perform an action that can also be performed in another way. For example, clicking twice on an icon is a faster way to open it than choosing Open; clicking twice on a word to select it is faster than dragging through it.

An operation invoked by double-clicking an object must be an enhancement, superset, or extension of the feature invoked by single-clicking that object.

Triple-clicking is also possible; it should similarly represent an extension of a double-click.

Changing Pointer Shapes

The pointer may change shape to give feedback on the range of activities that make sense in a particular area of the screen, in a current mode, or both.

- The result of any mouse action depends on the item under the pointer when the mouse button is pressed. To emphasize the differences among mouse actions, the pointer may assume different appearances in different areas to indicate the actions possible in each area.

- Where an application uses modes for different functions, the pointer can be a different shape in each mode. For example, in MousePaint, the pointer shape always reflects the active palette symbol.

Figure 5 shows some examples of pointers and their effect. An application can design additional pointers for other contexts.

MouseGraphics Pointer	MouseText Pointer	Used for
		bar, desktop, and so on
		Scroll bar and other controls, size box title bar, menu bar, desktop, and so on
		Selecting text
	N/A	Drawing, shrinking, or stretching graphic objects
		Selecting fields in an array
		Showing that a lengthy operation is in progress

Figure 5. Pointers

SELECTING

The user selects an object to distinguish it from other objects just before performing an operation on it. Selecting the object of an operation before identifying the operation is a fundamental characteristic of windowing software.

Selecting an object has no effect on the contents of a document. Making a selection shouldn't commit the user to anything; the user is never penalized for making an incorrect selection. The user fixes an incorrect selection by making the correct selection.

Although there is a variety of ways to select objects, they fall into easily recognizable groups. Users get used to doing specific things to select objects, and applications that use these methods are therefore easier to learn. Some of these methods apply to every type of application, and some only to particular types of applications.

This section discusses first the general methods, and then the specific methods that apply to text applications, graphics applications, and arrays. Figure 6 shows a comparison of some of the general methods.

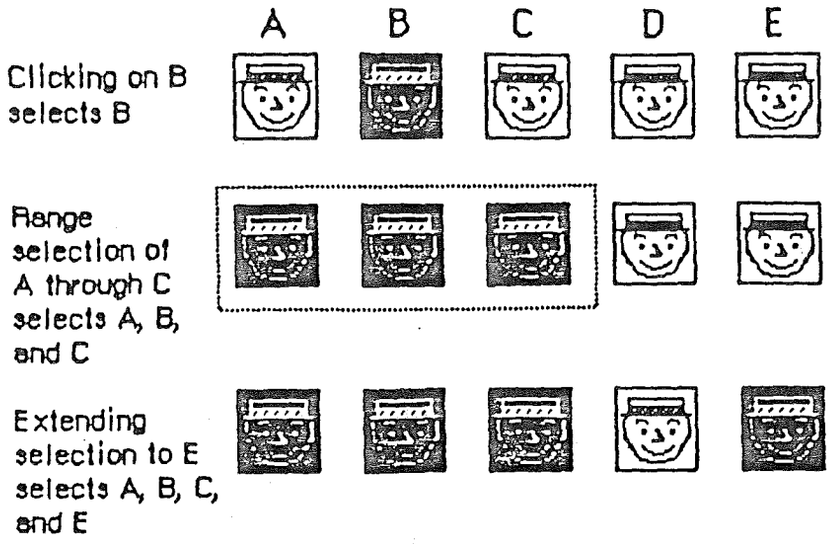


Figure 6. Selection Methods

Selection by Clicking

The most straightforward method of selecting an object is by clicking on it once. Most things that can be selected in windowing applications can be selected this way.

Some applications support selection by double-clicking and triple-clicking. As always with multiple clicks, the second click extends the effect of the first click, and the third click extends the effect of the second click. In the case of selection, this means that the second click selects the same sort of thing as the first click, only more of them. The same holds true for the third click.

For example, in text, the first click selects an insertion point, whereas the second click selects a whole word. The third click might select a whole block or paragraph of text. In graphics, the first click selects a single object, and double- and triple-clicks might select increasingly larger groups of objects.

Range Selection

The user selects a range of objects by dragging through them. Although the exact meaning of the selection depends on the type of application, the procedure is always the same:

1. The user positions the pointer at one corner of the range and presses the mouse button. This position is called the anchor point of the range.

2. The user moves the pointer in any direction. As the pointer is moved, visual feedback keeps the user informed of the objects that

would be selected if the mouse button were released. For text and arrays, the selected area is continually highlighted. For graphics, a dotted rectangle expands or contracts to show the range that will be selected.

3. When the feedback shows the desired range, the user releases the mouse button. The point at which the button is released is called the endpoint of the range.

Extending a Selection

A user can change the extent of an existing selection by holding down the Open-Apple key and clicking the mouse button. (This is an unfortunate but unavoidable difference with MacIntosh, where Shift-click is used instead. Should Apple II hardware ever permit reading the shift key by itself, windowing software should accept either a Shift-click or an Open-Apple-click.) Exactly what happens next depends on the context.

In text or an array, the result of an Open-Apple-click is always a range. The position where the button is clicked becomes the new endpoint or anchor point of the range; the selection can be extended in any direction. If the user clicks within the current range, the new range will be smaller than the old range.

In graphics, a selection is extended by adding objects to it; the added objects do not have to be adjacent to the objects already selected. The user can add either an individual object or a range of objects to the selection by holding down the Open-Apple key before making the additional selection. If the user holds down the Open-Apple key and selects one or more objects that are already highlighted, the objects are deselected.

Extended selections can be made across the panes of a split window. (See "Splitting Windows".)

Making a Discontinuous Selection

In graphics applications, objects aren't usually considered to be in any particular sequence. Therefore, the user can use Open-Apple-click to extend a selection by a single object, even if that object is nowhere near the current selection. When this happens, the objects between the current selection and the new object are not automatically included in the selection. This kind of selection is called a discontinuous selection. In the case of graphics, all selections are discontinuous selections.

This is not the case with arrays and text, however. In these two kinds of applications, an extended selection made by an Open-Apple-click always includes everything between the old selection and the new endpoint. To provide the possibility of a discontinuous selection in

these applications, the user interface includes Solid-Apple-click.

To make a discontinuous selection in a text or array application, the user selects the first piece in the normal way, then holds down the Solid-Apple key before selecting the remaining pieces. (It is useful but not absolutely necessary that the user have two right hands.) Each piece is selected in the same way as if it were the whole selection, but because the Solid-Apple key is held down, the new pieces are added to the existing selection instead of supplanting it.

If one of the pieces selected is already within an existing part of the selection, then instead of being added to the selection it's removed from the selection. Figure 7 shows a sequence in which several pieces are selected and deselected.

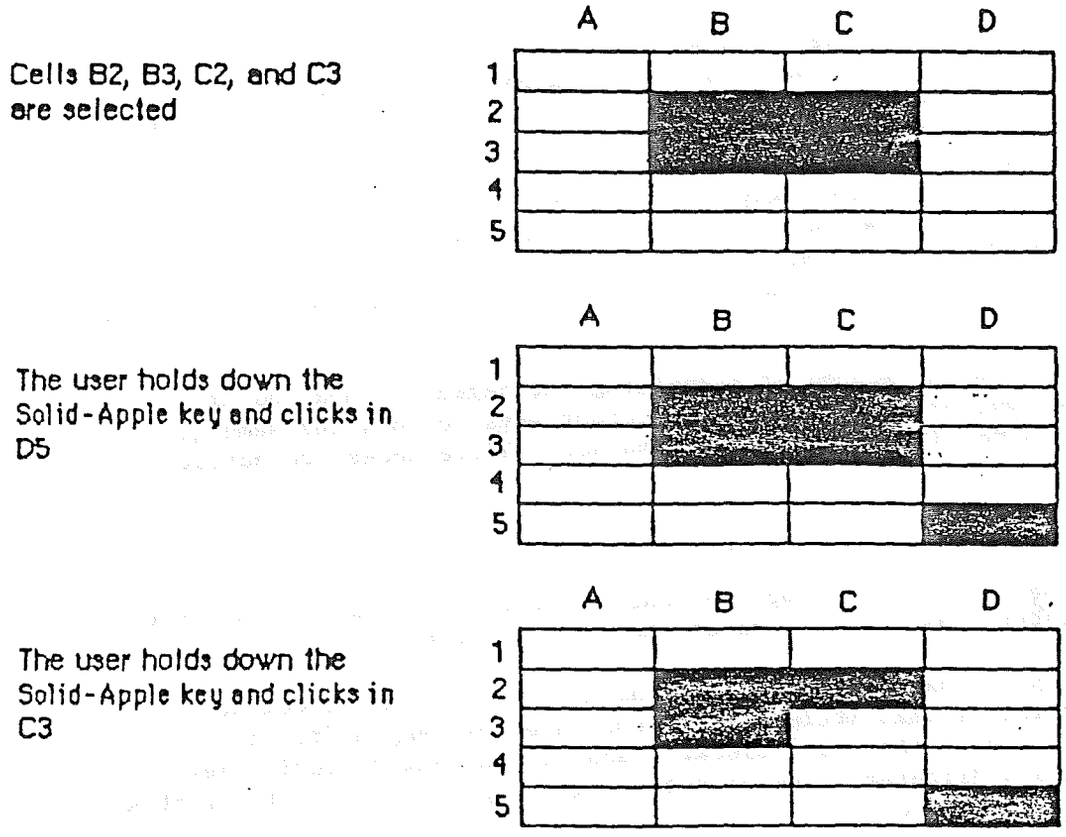


Figure 7. Discontinuous Selection

Not all applications support discontinuous selections, and those that do might restrict the operations that a user can perform on them. For example, a word processor might allow the user to choose a font after making a discontinuous selection, but not to choose Cut or Paste.

Selecting with the Cursor Keys

The user can alternatively mark a selection using the cursor keys. To signal the system that a selection is about to be marked, the user presses Open-Apple-M. The current insertion-point then becomes the anchor of the selection, and the selection can be extended in any direction using the four cursor keys.

Pressing Open-Apple-M twice before using the cursor keys is equivalent to a mouse double-click; pressing Open-Apple-M three times is equivalent to a mouse triple-click. In the case of text, the anchor-point is always at the top-left point of the expanded insertion-point and the current cursor position is always at the bottom-right.

a selection, the user presses Solid-Apple-M as an equivalent of Open-Apple-click. There is no equivalent of Solid-Apple-click. Only provide Solid-Apple-M if it is really needed: One of the primary reasons for being able to extend a text selection in the mouse world is that the user needs to go to the scroll-bar to move any significant distance. The cursor-key user does not need to use the scroll bar. Reaching a window end will scroll the window's contents; pressing Open-Apple at the same time as a cursor key will move the cursor by an appropriately large chunk, such as one word horizontally or one page vertically.

Other than the method of signalling the beginning and end of a selection, selecting using the cursor keys follows the same general guidelines as selecting with a mouse. (Differences are noted.)

Selecting Text

Text is used in most applications; it's selected and edited in a consistent way, regardless of where it appears.

A block of text is a string of characters. A text selection is a substring of this string, which can have any length from zero characters to the whole block. Each of the text selection methods selects a different kind of substring. Figure 8 shows different kinds of text selections.

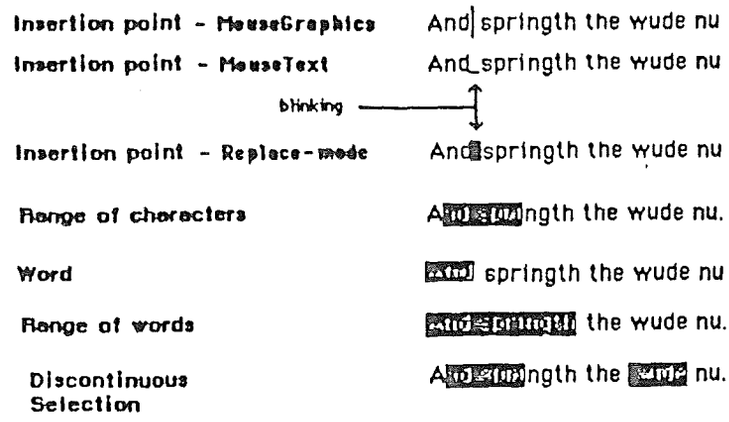


Figure 8. Text Selections

Insertion Point

The insertion point is a zero-length text selection. The user establishes the location of the insertion point by clicking between two characters. The insertion point then appears at the nearest character boundary. If the user clicks to the right of the last character on a line, the insertion point appears immediately after the last character. The converse is true if the user clicks to the left of the first character in the line.

The insertion point shows where text will be inserted when the user begins typing, or where the contents of the Clipboard will be pasted. After each character is typed, the insertion point is relocated to the right of the insertion.

If, between the mouse-down and the mouse-up, the user moves the pointer more than about half the width of a character, the selection is a range selection rather than an insertion point. The cursor-key user begins the same process with Open-Apple-M.

Selecting Words

The user selects a whole word by double-clicking somewhere within that word. If the user begins a double-click sequence, but then drags the mouse between the mouse-down and the mouse-up of the second click, the selection becomes a range of words rather than a single word. As the pointer moves, the application highlights or unhighlights a whole word at a time. The cursor-key user cannot select a range of words, although she can select the whole first word or paragraph by pressing Open-Apple-M repeatedly.

A word, or range of words, can also be selected in the same way as any other range; whether this type of selection is treated as a range of

characters or as a range of words depends on the operation. For example, in MacWrite, a range of individual characters that happens to coincide with a range of words is treated like characters for purposes of extending a selection, but is treated like words for purposes of intelligent cut and paste.

A word is defined as any continuous substring that contains only the following characters:

- a letter (including letters with diacritical marks)
- a digit
- a nonbreaking space (Open-Apple-Space)
- a dollar sign, cent sign, English pound symbol, or yen symbol
- a percent sign
- a comma between digits
- a period before a digit
- an apostrophe between letters or digits
- a hyphen, but not a minus sign (Open-Apple-hyphen)

This is the definition in the United States and Canada; in other countries, it would have to be changed to reflect local formats for numbers, dates, and currency.

If the user double-clicks over any character not on the list above, only that character is selected.

Examples of words:

$123,456.78
shouldn't
3 1/2 [with a nonbreaking space]
.5%

Examples of nonwords:

7/1Ø/6
blue cheese [with a breaking space]
"Yoicks!" [the quotation marks and exclamation point aren't part of the word]

Selecting a Range of Text

The user selects a range of text by dragging through the range. A range is either a range of words or a range of individual characters, as described under "Selecting Words", above.

If the user extends the range, the way the range is extended depends on what kind of range it is. If it's a range of individual characters, it can be extended one character at a time. If it's a range of words (including a single word), it's extended only by whole words.

Graphics Selections

There are several different ways to select graphic objects and to show selection feedback in existing Macintosh and Apple II applications. MacDraw, MousePaint, and the Macintosh Finder all illustrate different possibilities. This section describes the MacDraw paradigm, which is the most extensible to other kinds of applications.

A MacDraw document is a collection of individual graphic objects. To select one of these objects, the user clicks once on the object, which is then shown with knobs. (The knobs are used to stretch or shrink the object, and won't be discussed in this manual.) Figure 9 shows some examples of selection in MacDraw.

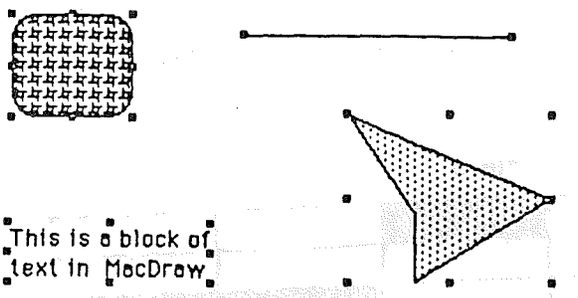


Figure 9. Graphics Selections in MacDraw

To select more than one object, the user can select either a range or a multiple selection. A range selection includes every object completely contained within the dotted rectangle that encloses the range, while an extended selection includes only those objects explicitly selected.

Selections in Arrays

As described above, under "Types of Applications", an array is a one- or two-dimensional arrangement of fields. If the array is one-dimensional, it's called a form; if it's two-dimensional, it's called a table. The user can select one or more fields, or part of the contents of a field.

To select a single field, the user clicks in the field. The user can also implicitly select a field by moving into it with the Tab or Return key.

The Tab key cycles through the fields in an order determined by the application. From each field, the Tab key selects the "next" field. Typically, the sequence of fields is first from left to right, and then from top to bottom. When the last field in a form is selected, pressing the Tab key selects the first field in the form. In a form, an application might prefer to select the fields in logical, rather than physical, order.

The Return key selects the first field in the next row. If the idea of rows doesn't make sense in a particular context, then the Return key should have the same effect as the Tab key.

Tables are more likely than forms to support range selections and extended selections. A table can also support selection of rows and columns. The most convenient way for the mouse user to select a column is to click in the column header. To select more than one column, the user drags through several column headers. The same applies to rows. The keyboard user will need an Open-Apple "short-cut".

To select part of the contents of a field, the user must first select the field. The user then clicks again to select the desired part of the field. Since the contents of a field are either text or graphics, this type of selection follows the rules outlined above. Figure 1Ø shows some selections in an array.

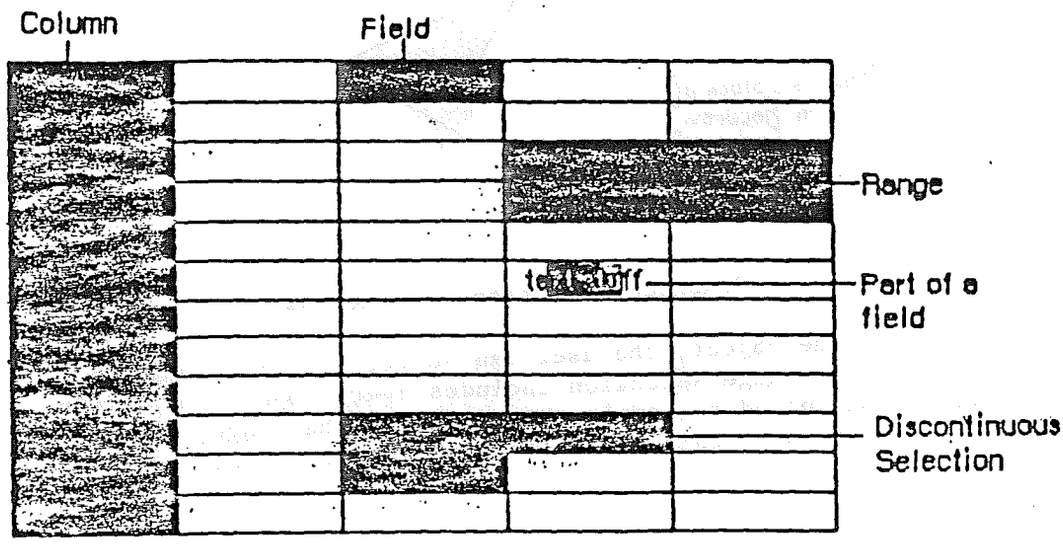


Figure 1Ø. Array Selections

WINDOWS

Windows are the rectangles on the desktop that display information. The most commmon types of windows are document windows, desk accessories, dialog boxes, and alert boxes. (Dialog and alert boxes are discussed under "Dialogs and Alerts".) Some of the features described in this section are applicable only to document windows. Figure 11 shows a typical active window and some of its components.

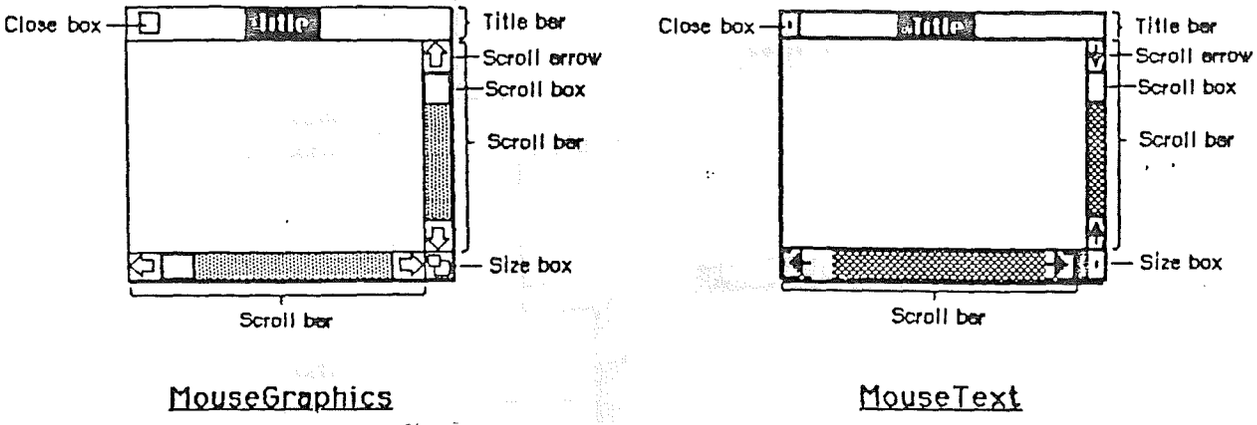


Figure 11. An Active Window

Multiple Windows

Some applications may be able to keep several windows on the desktop at the same time. Each window is in a different plane. Windows can be moved around on the desktop much like pieces of paper can be moved around on a real desktop. Each window can overlap those behind it, and can be overlapped by those in front of it. Even when windows don't overlap, they retain their front-to-back ordering.

Different windows can represent:

- different parts of the same document, such as the beginning and end of a long report
- different interpretations of the same document, such as the tabular and chart forms of a set of numerical data
- related parts of a logical whole, like the listing, execution, and debugging of a program
- separate documents being viewed or edited simultaneously

Each application may deal with the meaning and creation of multiple windows in its own way.

The advantage of multiple windows is that the user can isolate unrelated chunks of information from each other. The disadvantage is that the desktop can become cluttered, especially if some of the windows can't be moved. Figure 12 shows multiple windows.

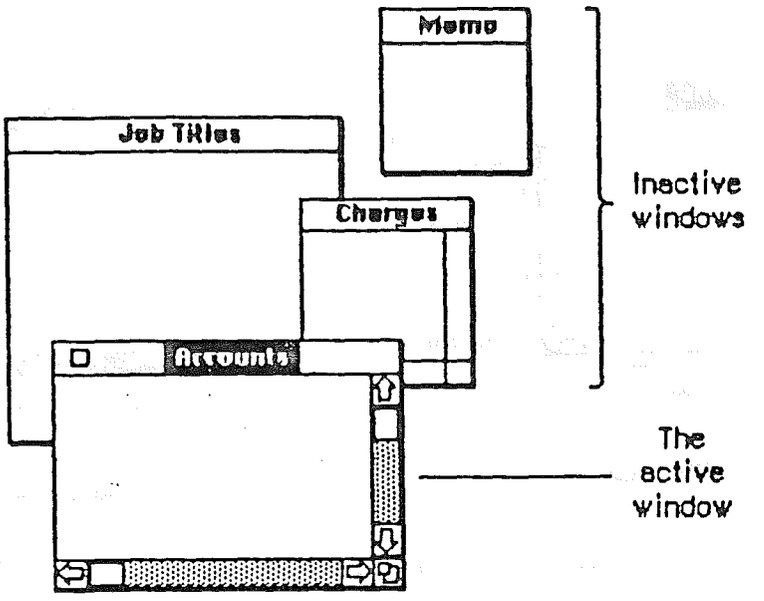


Figure 12. Multiple Windows

Opening and Closing Windows

Windows come up onto the screen in different ways as appropriate to the purpose of the window. The application controls at least the initial size and placement of its windows.

Most windows have a close box that, when clicked, makes the window go away. The application in control of the window determines what's done with the window visually and logically when the close box is clicked. Visually, the window can either shrink to a smaller object such as an icon, or leave no trace behind when it closes. Logically, the information in the window is either retained and then restored when the window is reopened (which is the usual case), or else the window is reinitialized each time it's opened. When a document is closed, the user is given the choice whether to save any changes made to the document since the last time it was saved.

If an application doesn't support closing a window with a close box, it should not include a close box on the window.

The Active Window

Of all the windows that are open on the desktop, the user can work in only one window at a time. This window is called the active window. All other open windows are inactive. To make a window active, the user clicks in it (or uses an Open-Apple short-cut). Making a window active has two immediate consequences:

- The window changes appearance: Its title bar is highlighted, the scrolling apparatus is shown in the scroll bars, and a size or grow box is shown. If the window is being reactivated, the selection that was in effect when it was deactivated is rehighlighted.

- The window is moved to the frontmost plane, so that it's shown in front of any windows that it overlaps.

Clicking in a window does nothing except activate it. To make a selection in the window, the user must click again. When the user clicks in a window that has been deactivated, the window should be reinstated just the way it was when it was deactivated, with the same position of the scroll box, and the same selection highlighted.

When a window becomes inactive, all the visual changes that took place when it was activated are reversed. The title bar becomes unhighlighted, the scrolling apparatus isn't shown in the scroll bars, the size box isn't shown, and no selection is shown in the window.

Moving (Dragging) a Window

Each application initially places windows on the screen wherever it wants them. The user can move a window--to make more room on the desktop or to uncover a window it's overlapping--by dragging it by its title bar. As soon as the user presses in the title bar, that window becomes the active window. A dotted outline of the window follows the pointer until the user releases the mouse button. At the release of the button the full window is drawn in its new location. Moving a window doesn't affect the appearance of the document within the window.

If the user holds down the Open-Apple key while dragging the window outline, the window isn't made active; it moves in the same plane. (At the time of this writing, this feature had not been implemented in the Apple II mouse toolkits.)

The standard keyboard shortcut for dragging a window is Open-Apple-D . The user can then use the cursor keys to drag the window outline. The user exits by pressing ESC to cancel, RETURN to accept the new location, or any other valid Open-Apple combination to accept and begin the next operation.

The application should ensure that a window can never be moved completely off the screen.

Changing the Size of a Window

If a window has a size or grow box in its bottom right corner, where the scroll bars come together, the user can change (grow) the size of the window--enlarging or reducing it to the desired size.

Dragging the size box attaches a dotted outline of the window to the pointer. The outline's top left corner stays fixed, while the bottom right corner follows the pointer. When the mouse button is released, the entire window is redrawn in the shape of the dotted outline. The standard keyboard shortcut for growing a window is Open-Apple-G. The user can then use the cursor keys for growing or shrinking the window. The user exits by pressing ESC to cancel, RETURN to accept the new size, or any other valid Open-Apple combination to accept and move on.

Moving windows and growing them go hand in hand. If a window can be moved, but not shrunk or grown, then the user ends up constantly dragging windows on and off the screen. The reason for this is that if the user drags the window off the right or bottom edge of the screen, the scroll bars are the first thing to disappear. To scroll the window, the user must move the window back onto the screen again. If, on the other hand, the window can be resized, then the user can change its size instead of dragging it off the screen, and will still be able to scroll.

Growing a window doesn't change the position of the top left corner of the window over the document or the appearance of the part of the view that's still showing; it changes only how much of the view is visible inside the window. One exception to this rule is a command such as Reduce to Fit in MacDraw, which changes the scaling of the view to fit the size of the window. If, after choosing this command, the user resizes the window, the application changes the scaling of the view.

The application can define a minimum window size. Any attempt to shrink the window below this size is ignored.

Scroll Bars

Scroll bars are used to change which part of a document view is shown in a window. Only the active window can be scrolled.

A scroll bar (see Figure 11 above) is a light gray shaft, capped on each end with square boxes labeled with arrows; inside the shaft is a white rectangle. The shaft represents one dimension of the entire document; the white rectangle (called the scroll box) represents the location of the portion of the document currently visible inside the window. As the user moves the document under the window, the position of the rectangle in the scroll bar moves correspondingly. If the document is no larger than the window, the scroll bars are inactive; they appear the same as they would in an inactive document.

There are three ways to use the mouse to move the document under the window: by sequential scrolling, by "paging" windowful by windowful through the document, and by directly positioning the scroll box.

Clicking a scroll arrow moves the document in the opposite direction from the scroll arrow. For example, when the user clicks the top scroll arrow, the document moves down, bringing the view closer to the top of the document. The scroll box moves towards the arrow being clicked.

Each click in a scroll arrow causes movement a distance of one unit in the chosen direction, with the unit of distance being appropriate to the application: one line for a word processor, one row or column for a spreadsheet, and so on. Within a document, units should always be the same size, for smooth scrolling. Pressing the scroll arrow causes continuous movement in its direction.

Clicking the mouse anywhere in the gray area of the scroll bar advances the document by windowfuls. The scroll box, and the document view, move toward the place where the user clicked. Clicking below the scroll box, for example, brings the user the next windowful towards the bottom of the document. Pressing in the gray area keeps windowfuls flipping by until the user releases the button, or until the location of the scroll box catches up to the location of the pointer. Each windowful is the height or width of the window, minus one unit overlap (where a unit is the distance the view scrolls when the scroll arrow is clicked once).

In both the above schemes the user moves the document incrementally until it's in the proper position under the window; as the document moves, the scroll box moves accordingly. The user can also move the document directly to any position simply by moving the scroll box to the corresponding position in the scroll bar. To move the scroll box, the user drags it along the scroll bar; an outline of the scroll box follows the pointer. When the mouse button is released, the scroll box jumps to the position last held by the outline, and the document jumps to the position corresponding to the new position of the scroll box.

If the user starts dragging the scroll box, then moves the pointer a certain distance outside the scroll bar, the scroll box detaches itself from the pointer and stops following it; if the user releases the mouse button, the scroll box returns to its original position and the document remains unmoved. But if the user still holds the mouse button and drags the pointer back into the scroll bar, the scroll box reattaches itself to the pointer and can be dragged as usual.

In graphics-based applications, if a document has a fixed size, and the user scrolls to the right or bottom edge of the document, the application displays a small amount of gray background (the same pattern as the desktop) between the edge of the document and the window frame.

Cursor-Key Scrolling

The Apple II interface also includes the ability to move and scroll using the cursor keys. Each press of a cursor key moves the insertion point one unit in the chosen direction, with the unit of distance being appropriate to the application. When the insertion point has been moved to a window edge, the insertion point locks, and the contents of the window begin to be shifted one unit in the opposite direction. At that point, that cursor key acts like the equivalent scroll arrow.

The user can increase the extent of the movement by holding down Open-Apple while pressing the cursor key. The insertion point will then move by the next larger contextual unit. For example, in a word-processor, Open-Apple-Left-Arrow moves one word at a time, Open-Apple-Up-Arrow moves one windowful at a time.

You may also provide a method to substitute for the large leaps the mouse user can make by dragging the scroll box. Text programs, for example, have historically used Open-Apple with the numbers 1 through 9 to move to an absolute position, with 1 being the first character in a file and 9 being the last. Array windows, such as spreadsheets, will probably want to allow the user to enter a column or row designation to move directly there.

Automatic Scrolling

There are several instances when the application, rather than the user, scrolls the document. These instances involve some potentially sticky problems about how to position the document within the window after scrolling.

The first case is when the user moves the pointer out of the window while selecting by dragging. The window keeps up with the selection by scrolling automatically in the direction the pointer has been moved. The rate of scrolling is the same as if the user were pressing on the corresponding scroll arrow or arrows.

The second case is when the selection isn't currently showing in the window, and the user performs an operation on it. When this happens, it's usually because the user has scrolled the document after making a selection. In this case, the application scrolls the window so that the selection is showing before performing the operation.

The third case is when the application performs an operation whose side-effect is to make a new selection. An example is a search operation, after which the object of the search is selected. If this object isn't showing in the window, the application must scroll the window so as to show it.

The second and third cases present the same problem: Where should the selection be positioned within the window after scrolling? The primary rule is that the application should avoid unnecessary scrolling; users

prefer to retain control over the positioning of a document. The following guidelines should be helpful:

- If part of the new selection is already showing in the window, don't scroll at all. An exception to this rule is when the part of the selection that isn't showing is more important than the part that's showing.

- If scrolling in one orientation (horizontal or vertical) is sufficient to reveal the selection, don't scroll in both orientations.

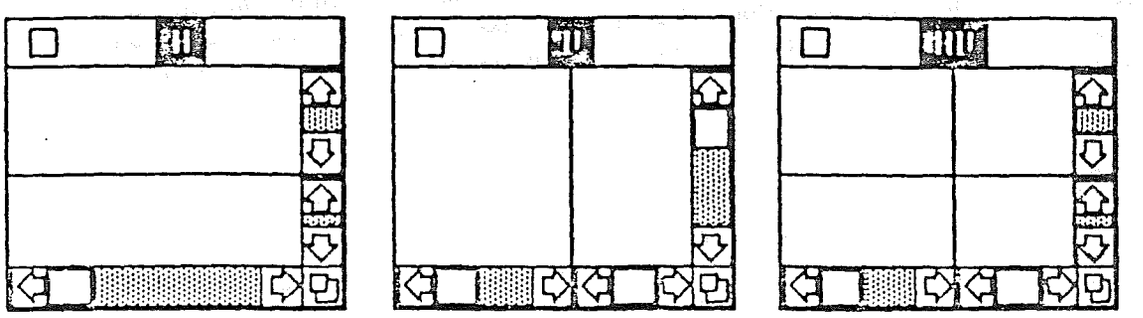
- If the selection is smaller than the window, position the selection so that some of its context is showing on each side. It's better to put the selection somewhere near the middle of the window than right up against the corner.

- Even if the selection is too large to show in the window, it might be preferable to show some context rather than to try to fit as much as possible of the selection in the window.

Splitting a Window

Sometimes it's desirable to be able to see disjoint parts of a document simultaneously. Applications that accommodate such a capability allow the window to be split into independently scrollable panes.

Applications that support splitting a window into panes place split bars at the top of the vertical scroll bar and to the left of the horizontal one. Pressing a split bar attaches it to the pointer. Dragging the split bar positions it anywhere along the scroll bar; releasing the mouse button moves the split bar to a new position, splits the window at that location, and divides the appropriate scroll bar (horizontal or vertical) into separate scroll bars for each pane. Figure 13 shows the ways a window can be split.



Horizontal Split Vertical Split Both Splits

Figure 13. Types of Split Windows

After a split, the document appears the same, except for the split line lying across it. But there are now separate scroll bars for each pane. The panes are still scrolled together in the orientation of the split, but can be scrolled independently in the other orientation. For example, if the split is horizontal, then horizontal scrolling (using the scroll bar along the bottom of the window), is still synchronous. Vertical scrolling is controlled separately for each pane, using the two scroll bars along the right of the window. This is shown in Figure 14.

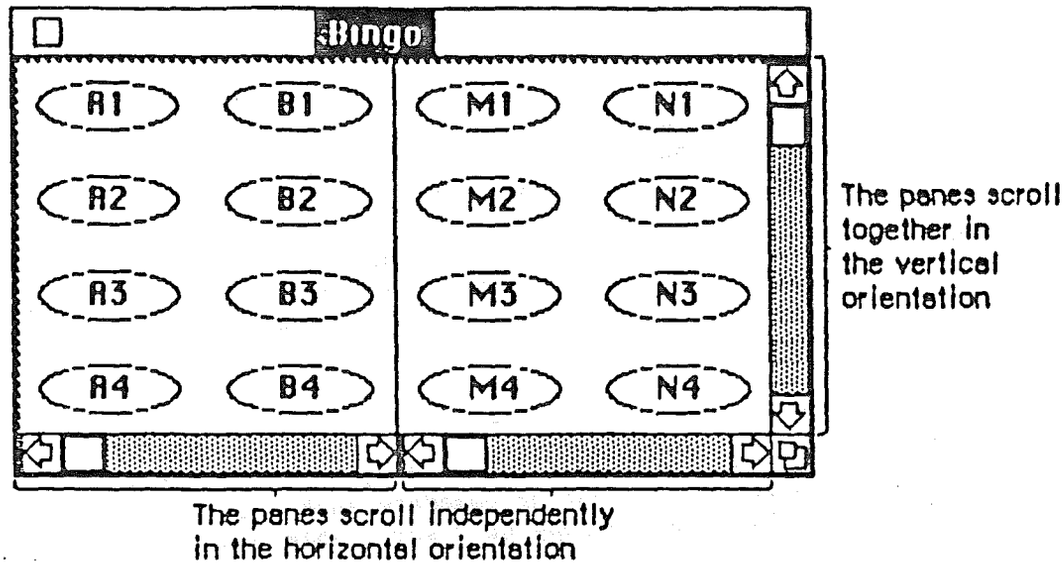


Figure 14. Scrolling a Split Window

To remove a split, the user drags the split bar to the bottom or the right of the window.

The number of views in a document doesn't alter the number of selections per document: that is, one. The active selection appears highlighted in all views that show it. If the application has to scroll automatically to show the selection, the pane that should be scrolled is the last one that the user clicked in. If the selection is already showing in one of the panes, no automatic scrolling takes place.

Panels

If a document window is more or less permanently divided into different regions, each of which has different content, these regions are called panels. Unlike panes, which show different parts of the same document but are functionally identical, panels are functionally different from each other but might show different interpretations of the same part of the document. For example, one panel might show a graphic version of the document while another panel shows a textual version.

Panels can behave much like subwindows; they can have scroll bars, and can even be split into more than one pane. An example of a panel with scroll bars is the list of files in the Open dialog box.

Whether to use panels instead of separate windows is up to the application. Multiple panels in the same window are more compact than separate windows, but they have to be moved, opened, and closed as a unit.

COMMANDS

Once the information to be operated on has been selected, a command to operate on that information can be chosen from lists of commands called menus.

The Apple II's pull-down menus have the advantage that they're not visible until the user wants to see them; at the same time they're easy for the user to see and choose items from.

Most commands either do something, in which case they're verbs or verb phrases, or else they specify an attribute of an object, in which case they're adjectives. They usually apply to the current selection, although some commands apply to the whole document or window.

When you're designing your application, don't assume that everything has to be done through menu commands. Sometimes it's more appropriate for an operation to take place as a result of direct user manipulation of a graphic object on the screen, such as a control or icon. Alternatively, a single command can execute complicated instructions if it brings up a dialog box for the user to fill in.

The Menu Bar

The menu bar is displayed at the top of the screen. It contains a number of words and phrases: These are the titles of the menus associated with the current application. Each application has its own menu bar. The names of the menus do not change, except when the user calls for a desk accessory that uses different menus.

Only menu titles appear in the menu bar. If all of the commands in a menu are currently disabled (that is, the user can't choose them), the menu title should be dimmed (in gray type or flanked by the ASCII 127 checkerboard). The user can pull down the menu to see the commands, but can't choose any of them.

Choosing Menu Commands

... With A Mouse

To choose a command, the user positions the pointer over the menu title and presses the mouse button. The application highlights the title and displays the menu, as shown in Figure 15.

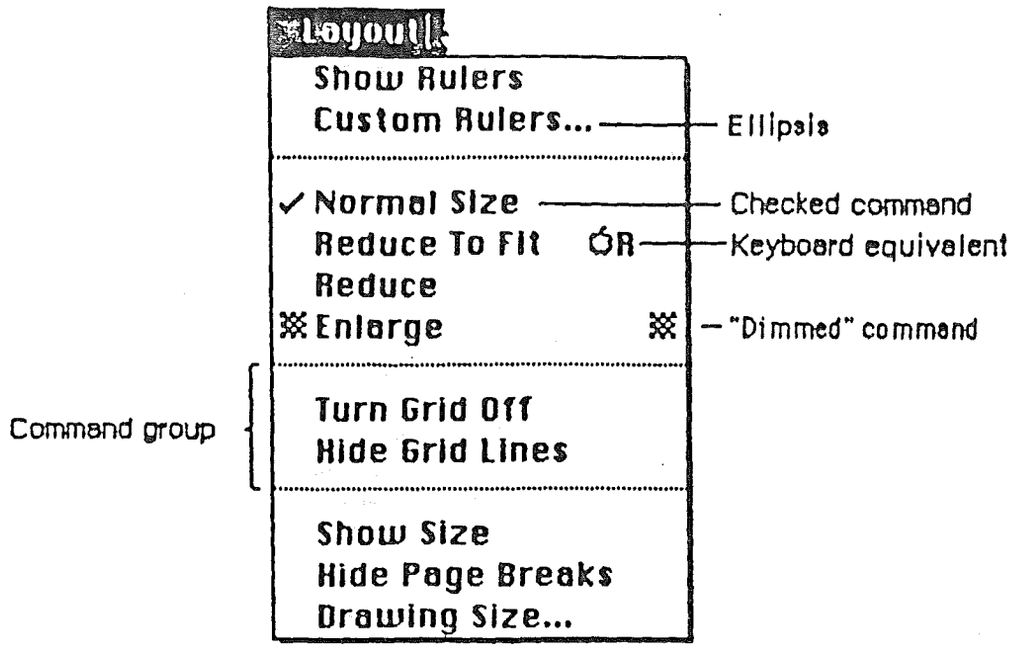


Figure 15. Menu

While holding down the mouse button, the user moves the pointer down the menu. As the pointer moves to each command, the command is highlighted. The command that's highlighted when the user releases the mouse button is chosen. As soon as the mouse button is released, the command blinks briefly, the menu disappears, and the command is executed. The menu title in the menu bar remains highlighted until the command has completed execution.

Nothing actually happens until the user chooses the command; the user can look at any of the menus without making a commitment to do anything.

The most frequently used commands should be at the top of a menu; research shows that the easiest item for the mouse user to choose is the second item from the top. The most dangerous commands should be at the bottom of the menu, preferably isolated from the frequently used commands.

... With the Cursor Keys

Pressing Escape within the application moves the user to the last item chosen on the menu. When the application begins, the initial cursor location should be the title of the first menu. Once at the menu, the user can move up and down the current menu with the Up and Down cursor keys, and move to the top of the adjacent menus using the Left or Right cursor keys. Once the user has reached the desired item, it is

selected by pressing Return. If the user is on the title of a menu or on a disabled item when Return is pressed, no action will be taken.

The user may also select an item when in the menu by pressing it's keyboard equivalent key (see: Keyboard Equivalents, below). The keyboard equivalent command will be carried out and the menu operation will be cancelled. The user can choose simply to cancel the menu operation by pressing Escape to return to the application.

Escape is normally defined on the Apple II as, "move me one level up in the program". This definition is retained in windowing software, as Escape will cancel dialog boxes, current inputs, and so forth. The only addition is that when the user is already at the top level (the application), it will toggle between application and menu.

If a command can be chosen directly from the keyboard, it's followed by the Open-Apple, Solid-Apple, or Control key (diamond) symbol and the character used to choose it. To choose a command this way, the user holds down the appropriate modifier key and then presses the character key.

Whenever practical, make all keyboard equivalents be Open-Apple combinations. If you want to provide a keyboard macro capability, let the user program macros to the Solid-Apple key. Otherwise, accept Solid-Apple keystrokes for Open-Apple commands. Avoid assigning two different commands to the same key, with only the use of Open- or Solid-Apple to differentiate. Generally, users do not recognize the difference between the two modifier keys.

While the toolkits enable you to use control characters for keyboard equivalents, we generally recommend against it for the following reasons:

- Most control keys are either tied to the hardware of the computer or are otherwise reserved. (See: Control combinations.)

- The diamond symbol for control is not generally recognized by users.

- Control keys are generally reserved for basic, simple, repetitive functions, such as moving by or deleting individual characters.

The advantage to control keys is their typeability: the current location of the Open- and Solid-Apple keys is such that they are difficult to touch-type with any speed or accuracy. We therefore recommend that you reserve control keys for only those things that must be done repetitively and unconciously. We suggest that even in these cases, you also enable the same key used with Open-Apple, as we have done with cut, copy, and paste. This tends to make documenting and learning easier, with the experienced user picking up the control shortcut at an appropriate time.

Reserved Key Combinations

Some characters are reserved for special purposes, but there are different degrees of stringency. Since almost every application has a File menu and an Edit menu, the keyboard equivalents in those menus are strongly reserved, and should never be used for any other purpose. All these equivalents may be selected while pressing the Open-Apple key. (All but Quit are also selectable while pressing the Control key, to enable touch-typists to manipulate the mouse while using these editing keys simultaneously):

Character	Command
Z	Undo (Edit menu)
X	Cut (Edit menu)
C	Copy (Edit menu)
V	Paste (Edit menu)
Q	Quit (File menu)

The following Open-Apple combinations are reserved for the keyboard equivalents of mouse operations:

Character	Command
D	Drag or move the currently active window
G	Grow or shrink (size) the currently active window
M	Mark a selection

One Open-Apple keyboard command doesn't have a menu equivalent:

Character	Command
.	Stop current operation

Other Open-Apple keyboard equivalents are conditionally reserved. If an application enables these commands, it shouldn't use these characters for any other purpose, but if it doesn't, it can use them however it likes:

Open-Apple combinations:

Character	Command
P	Print
S	Save

Control combinations:

Character	Command
B	Bold
C	Copy
D	Delete
E	Edit
F	Forward Delete
* H	Left Arrow
* I	Tab
* J	Down Arrow
* K	Up Arrow
L	Begin or End Underline
* M	Carriage Return
P	Print the contents of the screen
S	Save
* U	Right Arrow
V	Paste
X	Cut
Z	Undo
* [	Escape

* These are the control equivalents of the various Apple special keys. Current unmodified Apple II keyboards cannot differentiate between a Control-character sequence and its equivalent special key, for example, Control-M and Return.

Appearance of Menu Commands

The commands in a particular menu should be logically related to the title of the menu. In addition to command names, three features of menus help the user understand what each command does: command groups, toggles, and special visual features.

Command Groups

As mentioned above, menu commands can be divided into two kinds: verbs for actions and adjectives for attributes. An important difference between the two kinds of commands is that an attribute stays in effect until it's cancelled, while an action ceases to be relevant after it has been performed. Each of these two kinds can be grouped within a menu. Groups are separated by lines, which are implemented as disabled commands.

The most basic reason to group commands is to break up a menu so it's easier to read. Commands grouped for this reason are logically related, but independent. Commands that are actions are usually grouped this way, such as Cut, Copy, Paste, and Clear in the Edit menu.

Attribute commands that are interdependent are grouped to show this interdependence. Two kinds of attribute command groups are mutually exclusive groups and accumulating groups.

In a mutually exclusive attribute group, only one command in the group is in effect at the same time. The command that's in effect is preceded by a check mark. If the user chooses a different command in the group, the check mark is moved to the new command. An example is the Font menu in MacWrite; no more than one font can be in effect at a time.

In an accumulating attribute group, any number of attributes can be in effect at the same time. One special command in the group cancels all the other commands. An example is the Style menu in MacWrite: the user can choose any combination of Bold, Italic, Underline, Outline, or Shadow, but Plain Text cancels all the other commands.

Toggles

Another way to show the presence or absence of an attribute is by a toggled command. In this case, the attribute has two states, and a single command allows the user to toggle between the states. For example, when rulers are showing in MacWrite, a command in the Format menu reads "Hide Rulers". If the user chooses this command, the rulers are hidden, and the command is changed to read "Show Rulers". This kind of group should be used only when the wording of the commands makes it obvious that they're opposites.

Special Visual Features

In addition to the command names and how they're grouped, several other features of commands communicate information to the user:

- A check mark indicates whether an attribute command is currently in effect.

- An ellipsis (...) after a command name means that choosing that command brings up a dialog box. The command isn't actually executed until the user has finished filling in the dialog box and has clicked the OK button or its equivalent.

- The application dims a command (or flanks it with the ASCII 127 checkerboard) when the user can't choose it. If the user moves the pointer over a dimmed item, it isn't highlighted.

- If a command can be chosen from the keyboard, it's followed by the specific modifier key symbol (Open-Apple, Solid-Apple, or diamond

symbol for Control) and the character used to choose it.

The application can draw its own type of menu. An example of this is the Fill menu in MacDraw.

STANDARD MENUS

One of the strongest ways in which applications can take advantage of the consistency of the windowing user interface is by using standard menus. The operations controlled by these menus occur so frequently that it saves considerable time for users if they always match exactly. Three of these menus, the ? or Solid-Apple, File, and Edit menus, appear in almost every application.

The Solid-Apple Menu

..., OR MOUSETEXT CHARACTERS I C I,
(1 in a MouseText-based application, Solid-Apple in a graphics-based application)

Desk accessories are mini-applications that you may make available to your user while using your application. You can enable a user to issue a command at any time to call up one of several desk accessories. More than one accessory can be on the desktop at a time. An example of a menu of accessories is shown in Figure 16.

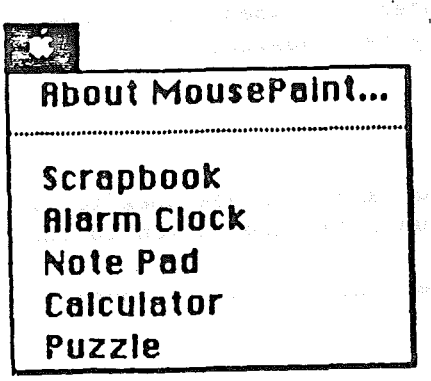


Figure 16. Apple Menu

The ? menu also contains the "About xxx" menu item, where "xxx" is the name of the application. Choosing this item brings up a dialog box with the name and copyright information for the application, as well as any other information the application wants to display.

The File Menu

The File menu allows the user to perform certain simple filing operations. It also contains the commands for printing and for leaving the application. The standard File menu includes the commands shown in Figure 17.

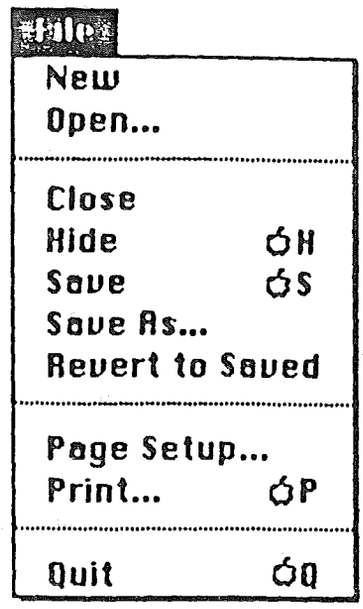


Figure 17. File Menu

Other frequently used commands are Print Draft, Print Final, and Print One. All of these commands are described below.

New

New opens a new, untitled document. The user names the document the first time it's saved. This command is disabled when the maximum number of documents allowed by the application is already open.

Open

Open opens an existing document. To select the document, the user is presented with a dialog box (Figure 18). This dialog box shows a list of all the documents on the disk whose name is displayed that can be handled by the current application. The user can scroll this list forward and backward. The dialog box also gives the user the chance to look at the documents on other disks in other disk drives that belong to the current application.

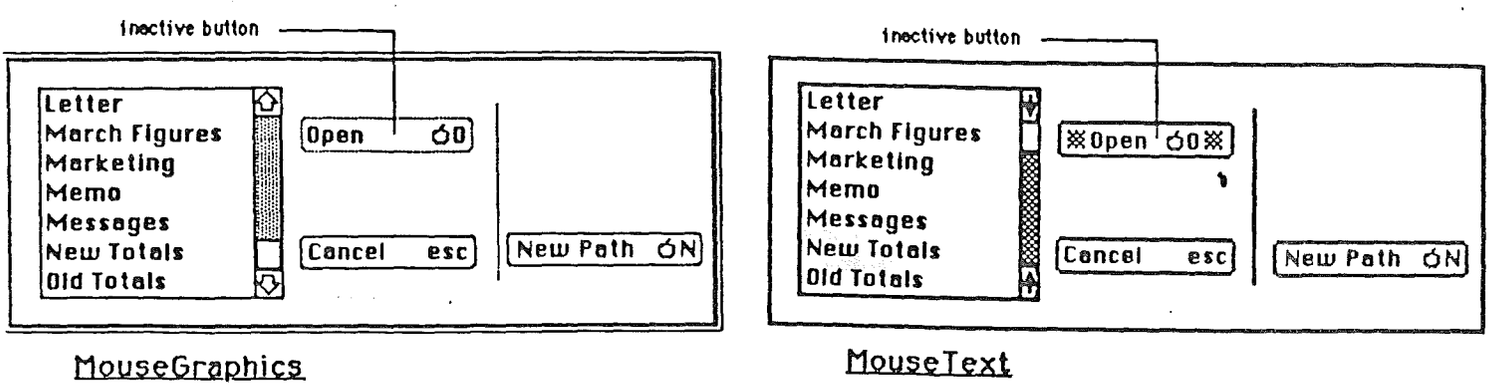


Figure 18. Open Dialog Box

Using the Open command, the user can only open a document that can be processed by the current application. Opening a document that can only be processed by a different application requires leaving the application.

This command is disabled when the maximum number of documents allowed by the application is already open.

Close

Close closes the active document or desk accessory. If the user has changed the document since the last time it was saved, the command presents an alert box giving the user the choice of whether or not to save the changes.

Clicking in the close box of a window is the same as choosing Close.

Save

Save makes permanent any changes to the active document since the last time it was saved. It leaves the document open.

If the user chooses Save for a new document that hasn't been named yet, the application presents the Save As dialog (see below) to name the document, and then continues with the save. The active document remains active.

If there's not enough room on the disk to save the document, the application asks if the user wants to save the document on another disk. If the answer is yes, the application goes through the Save As dialog to find out which disk.

Save As

Save As saves a copy of the active document under a file name provided by the user.

If the document already has a name, Save As closes the old version of the document, creates a copy, and displays the copy in the window.

If the document is untitled, Save As saves the original document under the specified name. The active document remains active.

Revert to Saved

Revert to Saved returns the document to the state it was in the last time it was saved. Before doing so, it puts up an alert box to confirm that this is what the user wants.

Page Setup

Page Setup lets the user specify printing parameters such as what the document's paper size and printing orientation are. These parameters remain with the document.

Print

Print lets the user specify various parameters such as print quality and number of copies, then prints the document. The parameters apply only to the current printing operation.

Quit

Quit leaves the application. If any open documents have been changed since the last time they were saved, the application presents the same alert box as for Close, once for each document.

Other Commands

Other commands that are in the File menu in some applications include:

- Print Draft. This command prints one copy of a rough version of a document more quickly than Print. It's useful in applications and with printers where ordinary printing is slow. If an application has this command, it should change the name of the Print command to Print Final.

- Print One. This command saves time by printing one copy using default parameters without bringing up the Print dialog box. If an application has this command, Open-Apple-P should be its

keyboard equivalent.

The Edit Menu

The Edit menu contains the commands that delete, move, and copy objects, as well as commands such as Undo, Show Clipboard, and Select All. This section also discusses the Clipboard, which is controlled by the Edit menu commands. Text editing methods that don't use menu commands are discussed under "Text Editing".

The standard order of commands in the Edit menu is shown in Figure 19.

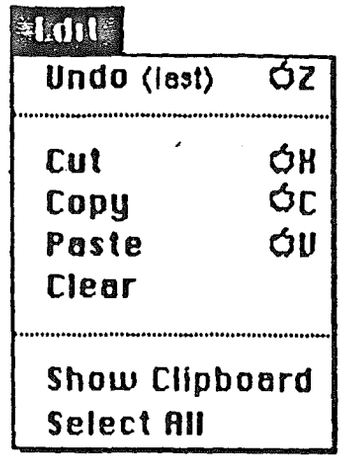


Figure 19. Edit Menu

The Clipboard

The Clipboard is a special kind of window with a well-defined function: it holds whatever is cut or copied from a document. Its contents stay intact when the user changes documents, opens a desk accessory, or leaves the application. An application can choose whether to have the Clipboard open or closed when the application starts up.

The Clipboard looks like a document window, with a close box but with no scroll bars. Its contents cannot be edited.

Every time the user performs a Cut or Copy on the current selection, a copy of the selection replaces the previous contents of the Clipboard. The previous contents are kept around in case the user chooses Undo.

The user can see the contents of the Clipboard but can't edit them. In most other ways the Clipboard behaves just like any other window.

There is only one Clipboard, which is present for all applications that support Cut, Copy, and Paste. The user can see the Clipboard window by choosing Show Clipboard from the Edit menu. If the window is already

showing, it's hidden by choosing Hide Clipboard. (Show Clipboard and Hide Clipboard are a single, toggled command.)

Undo

Undo reverses the effect of the previous operation. Not all operations can be undone; the definition of an undoable operation is somewhat application-dependent. The general rule is that operations that change the contents of the document are undoable, and operations that don't are not. Most menu items are undoable, and so are typing sequences.

A typing sequence is any sequence of characters typed from the keyboard or numeric keypad, including Delete, Return, and Tab, but not including keyboard equivalents of commands.

Operations that aren't undoable include selecting, scrolling, and splitting the window or changing its size or location. None of these operations interrupts a typing sequence. That is, if the user types a few characters and then scrolls the document, the Undo command still undoes the typing. Whenever the location affected by the Undo operation isn't currently showing on the screen, the application should scroll the document so the user can see the effect of the Undo.

An application should also allow the user to undo any operations that are initiated directly on the screen, without a menu command. This includes operations controlled by setting dials, clicking check boxes, and so on, as well as drawing graphic objects with the mouse.

The actual wording of the Undo command as it appears in the Edit menu is "Undo xxx", where xxx is the name of the last operation. If the last operation isn't a menu command, use some suitable term after the word Undo. If the last operation can't be undone, the command reads "Undo", but is disabled.

If the last operation was Undo, the menu command says "Redo xxx", where xxx is the operation that was undone. If this command is chosen, the Undo is undone.

Cut

The user chooses Cut either to delete the current selection or to move it. If it's a move, it's eventually completed by choosing Paste.

When the user chooses Cut, the application removes the current selection from the document and puts it in the Clipboard, replacing the Clipboard's previous contents. The place where the selection used to be becomes the new selection; the visual implications of this vary among applications. For example, in text, the new selection is an insertion point, while in an array, it's an empty but highlighted cell. If the user chooses Paste immediately after choosing Cut, the document should be just as it was before the cut; the Clipboard is unchanged.

When the user chooses Cut, the application doesn't know if it's a deletion or the first step of a move. Therefore, it must be prepared for either possibility.

Copy

Copy is the first stage of a copy operation. Copy puts a copy of the selection in the Clipboard, but the selection also remains in the document.

Paste

Paste is the last stage of a copy or move operation. It pastes the contents of the Clipboard to the document, replacing the current selection. The user can choose Paste several times in a row to paste multiple copies. After a paste, the new selection is the object that was pasted, except in text, where it's an insertion point immediately after the pasted text. The Clipboard remains unchanged.

Clear

When the user chooses Clear, the application removes the selection, but doesn't put it on the Clipboard. The new selection is the same as it would be after a Cut.

Show Clipboard

Show Clipboard is a toggled command. Initially, the Clipboard isn't displayed, and the command is "Show Clipboard". If the user chooses the command, the Clipboard is displayed and the command changes to "Hide Clipboard".

Select All

Select All selects every object in the document.

Font-Related Menus

Three standard Macintosh menus affect the appearance of text: Font, which determines the font of a text selection; FontSize, which determines the size of the characters; and Style, which determines aspects of its appearance such as boldface, italics, and so on.

Because of the proliferation of printers on the Apple II family, you may find it too expensive to implement the kind of power and range of font options available on the Macintosh. We have still including the full specification; use that portion necessary for your particular application.

Font Menu

A font is a set of typographical characters created with a consistent design. Things that relate characters in a font, include the thickness of vertical and horizontal lines, the degree and position of curves and swirls, and the use of serifs. A font has the same general appearance, regardless of the size of the characters. The Font menu always lists the fonts that are currently available. Figure 2Ø shows a Font menu with some of the most common fonts.

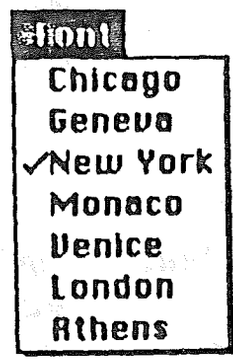


Figure 2Ø. Font Menu

FontSize Menu

Font sizes are measured in points; a point is about 1/72 of an inch. Each font is available in predefined sizes. The numbers of these sizes for each font are shown outlined in the FontSize menu. The font can also be scaled to other sizes, but it may not look as good. Figure 21 shows a FontSize menu with the standard Macintosh font sizes.

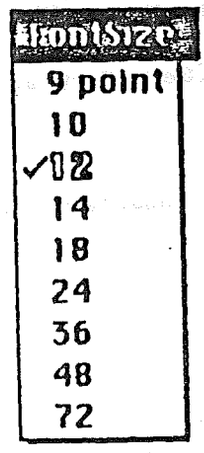


Figure 21. FontSize Menu

If there's insufficient room in the menu bar for the word FontSize, it can be abbreviated to Size. If there's insufficient room for both a Font menu and a Size menu, the sizes can be put at the end of the Font or Style menu.

Style Menu

The commands in the Style menu are Plain Text, Bold, Underline, Italic, Outline, and Shadow. The first three are reasonably implemented on most printers and should be considered standard on Apple. All the commands except Plain Text are accumulating attributes; the user can choose any combination. They are also toggled commands; a command that's in effect for the current selection is preceded by a check mark. Plain Text cancels all the other choices. Figure 22 shows these styles. II software.

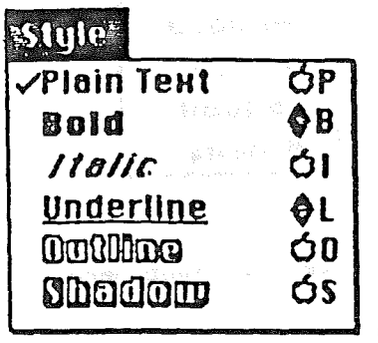


Figure 22. Style Menu

MouseText

If you are working in a MouseText-based toolkit, mark the beginning and end of a font-related change with the solid-diamond character. As the user passes over this character, open a view box to let the user see what the particular change is. (See: View Boxes under Dialogs and Alerts)

Figure 22A. MouseText Example

TEXT EDITING

In addition to the operations described under "The Edit Menu" above, there are other ways to edit text that don't use menu items.

Inserting Text

To insert text, the user selects an insertion point by clicking where the text is to go, then starts typing it. Alternatively, the user can move the current insertion point location, using the cursor keys, and then resume typing. As the user types, the application continually moves the insertion point to the right of each new character.

Applications with multiline text blocks should support word wraparound, according to the definition of a word detailed above under "Selecting a Word". The intent is that no word be broken between lines.

Delete

When the user presses the Delete key (or Control-D), one of two things happens:

- If the current selection is one or more characters, it's deleted.
- If the current selection is an insertion point, the previous character is deleted.

In both cases, the deleted characters don't go into the Clipboard, and the insertion point replaces the deleted characters in the document.

Forward Delete

When the user presses Control-F, one of two things happens:

- If the current selection is one or more characters, it's deleted (exactly as with Delete).
- If the current selection is an insertion point, the character to the right of the insertion point is deleted. (The cursor in a MouseText-based program is a blinking underscore. Since the underscore itself is to the right of the insertion-point, the effect is that the character immediately above the underscore is deleted.)

In both cases, the deleted characters don't go into the Clipboard.

Replacing Text

If the user starts typing when the selection is one or more characters, the characters that are typed replace the selection. The deleted characters don't go into the Clipboard, but the replacement can be undone by immediately choosing Undo.

Intelligent Cut and Paste

An application that lets the user select a word by double-clicking should also see to it that the user doesn't regret using this feature. The only way to do this is by providing "intelligent" cut and paste.

To understand why this feature is necessary, consider the following sequence of events in an application that doesn't provide it:

1. A sentence in the user's document reads: "Returns are only accepted if the merchandise is damaged." The user wants to change this to: "Returns are accepted only if the merchandise is damaged."

2. The user selects the word "only" by double-clicking. The letters are highlighted, but not either of the adjacent spaces.

3. The user chooses Cut, clicks just before the word "if", and chooses Paste.

4. The sentence now reads: "Returns are accepted onlyif the merchandise is damaged." To correct the sentence, the user has to remove a space between "are" and "accepted", and add one between "only" and "if". At this point he or she may be wondering why Apple computers are supposed to be easier to use than other computers.

If an application supports intelligent cut and paste, the rules to follow are:

- If the user selects a word or a range of words, highlight the selection, but not any adjacent spaces.

- When the user chooses Cut, if the character to the left of the selection is a space, discard it.

- When the user chooses Paste, if the character to the left of the current selection isn't a space, add a space. If the character to the right of the current selection isn't a punctuation mark or a space, add a space. Punctuation marks include the period, comma, exclamation point, question mark, apostrophe, colon, semicolon, and quotation mark.

This feature makes more sense if the application supports the full definition of a word (as detailed above under "Selecting a Word"), rather than the definition of a word as anything between two spaces.

These rules apply to any selection that's one or more whole words, whether it was chosen with a double-click or as a range selection.

Figure 23 shows some examples of intelligent cut and paste.

Example 1:

1. Select a word.	Drink to me only with thine eyes.
2. Choose Cut.	Drink to me|with thine eyes.
3. Select an insertion point.	Drink to me with |thine eyes.
4. Choose Paste.	Drink to me with only|thine eyes.

Example 2:

1. Select a word.	How, now brown cow
2. Choose Cut.	How,|brown cow
3. Select an insertion point	How| brown cow
4. Choose Paste.	How now| brown cow

Figure 23. Intelligent Cut and Paste

Editing Fields

If an application isn't primarily a text application, but does use text in fields (such as in a dialog box), it may not be able to provide the full text editing capabilities described so far.

It's important, however, that whatever editing capabilities the application provides under these circumstances be upward-compatible with the full text editing capability. The following list shows the capabilities that can be provided, going from the minimal to the most sophisticated:

- The user can move around using the cursor keys or mouse.
- The user can select the whole field and type in a new value.
- The user can backspace delete.
- The user can forward delete.
- The user can select a substring of the field and replace it.
- The user can select a word by double-clicking.
- The user can choose Undo, Cut, Copy, Paste, and Clear, as described above under "The Edit Menu". In the most sophisticated version, the application implements intelligent cut and paste.

An application should also perform appropriate edit checks. For example, if the only legitimate value for a field is a string of digits, the application might issue an alert if the user typed any nondigits. Alternatively, the application could wait until the user is through typing before checking the validity of the contents of the field. In this case, the appropriate time to check the field is when the user clicks anywhere other than within the field.

DIALOGS ALERTS

The "select-then-choose" paradigm is sufficient whenever operations are simple and act on only one object. But occasionally a command will require more than one object, or will need additional parameters before it can be executed. And sometimes a command won't be able to carry out its normal function, or will be unsure of the user's real intent. For these special circumstances the windowing user interface includes three additional features:

- dialogs, to allow the user to provide additional information before a command is executed
- alerts, to notify the user whenever an unusual situation occurs
- view, to enable the user of a text-based program to "look inside" the diamond icon

Since all of these features lean heavily on controls, controls are described in this section, even though controls are also used in other places.

Controls

Friendly systems act by direct cause-and-effect; they do what they're told. Performing actions on a system in an indirect fashion reduces the sense of direct manipulation. To give users the feeling that they're in control of their machines, many of a windowing application's features are implemented with controls: graphic objects that, when directly manipulated by the mouse, cause instant action with visible results.

There are four main types of controls: buttons, check boxes, radio buttons, and dials. These four kinds are shown in Figure 24.

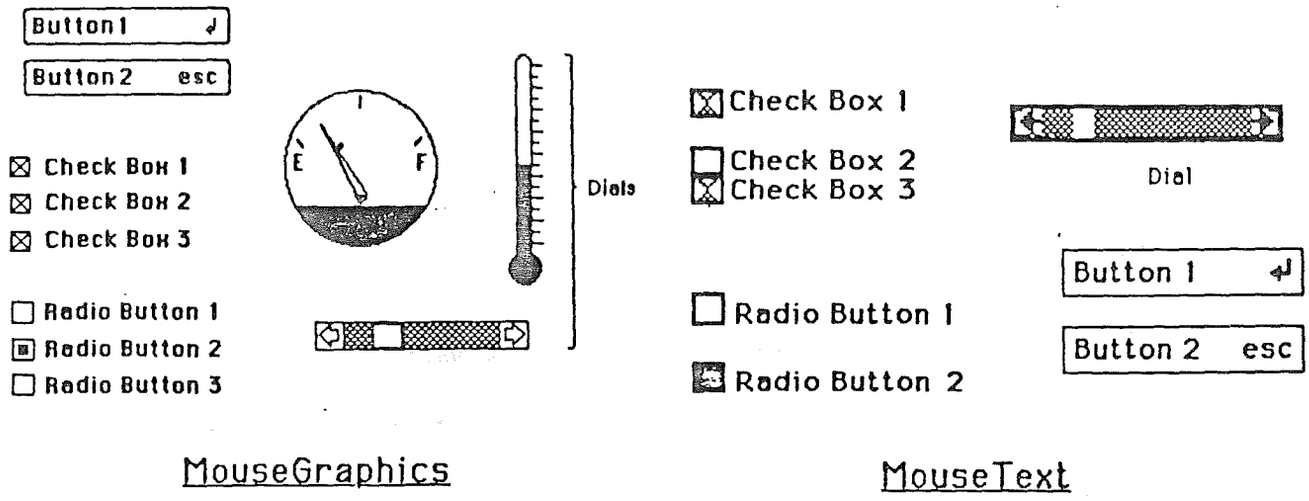


Figure 24. Controls

Buttons

Buttons are small objects, usually inside a window, labeled with text. Clicking or pressing a button performs the action described by the button's label.

Buttons perform instantaneous actions, such as completing operations defined by a dialog box or acknowledging error messages. Conceivably they could perform continuous actions, in which case the effect of pressing on the button would be the same as the effect of clicking it repeatedly.

Two particular buttons, OK and Cancel, are especially important in dialogs and alerts; they're discussed under those headings below.

Check Boxes and Radio Buttons

Whereas buttons perform instantaneous or continuous actions, check boxes and radio buttons let the user choose among alternative values for a parameter.

Check boxes act like toggle switches; they're used to indicate the state of a parameter that must be either off or on. The parameter is on if the box is checked, otherwise it's off. The check boxes appearing together in a given context are independent of each other; any number of them can be off or on.

Radio buttons typically occur in groups; they're round and are filled in with a black circle when on. (In a MouseText-based program, they

are rectangular and are filled in with a white rectangle when on.) They're called radio buttons because they act like the buttons on a car radio. At any given time, exactly one button in the group is on. Clicking one button in a group turns off the current button.

Both check boxes and radio buttons are accompanied by text that identifies what each button does.

Dials

Dials display the value, magnitude, or position of something in the application or system, and optionally allow the user to alter that value. Dials are predominantly analog devices, displaying their values graphically and allowing the user to change the value by dragging an indicator; dials may also have a digital display.

The most common example of a dial is the scroll bar. The indicator of the scroll bar is the scroll box; it represents the position of the window over the length of the document. The user can drag the scroll box to change that position.

Dialogs

Commands in menus normally act on only one object. If a command needs more information before it can be performed, it presents a dialog box to gather the additional information from the user. The user can tell which commands bring up dialog boxes because they're followed by an ellipsis (...) in the menu.

A dialog box is a rectangle that may contain text, controls, and icons. There should be some text in the box that indicates which command brought up the dialog box.

Other than explanatory text, the contents of a dialog box are all objects that the user sets to provide the needed information. These objects include controls and text fields. When the application puts up the dialog box, it should set the controls to some default setting and fill in the text fields with default values, if possible. One of the text fields (the "first" field) should be highlighted, so that the user can change its value just by typing in the new value. If all the text fields are blank, there should be an insertion point in the first field.

Editing text fields in a dialog box should conform to the guidelines detailed above, under "Text Editing".

When the user is through editing an item:

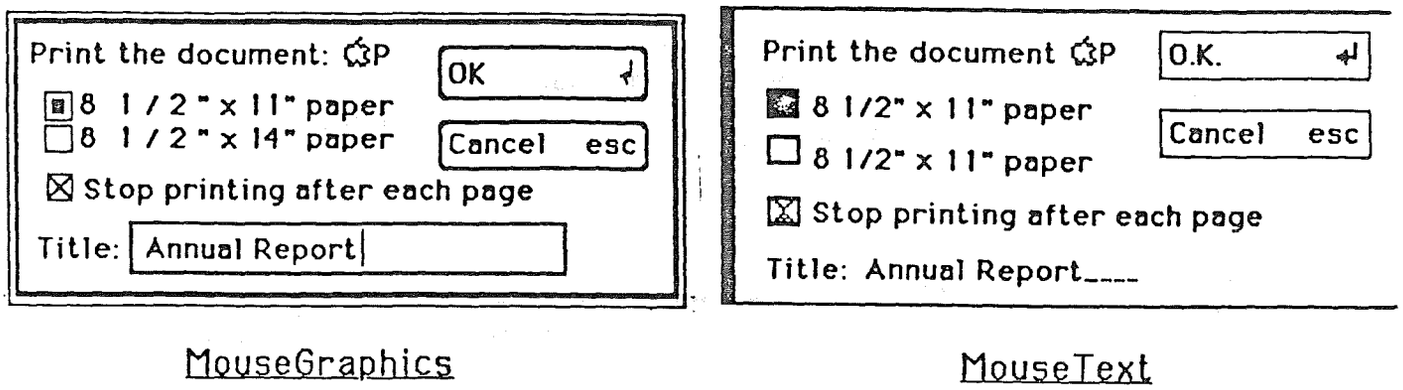
- Pressing Tab accepts the changes made to the item, and selects the next item in sequence.

- Clicking in another item accepts the changes made to the previous item and selects the newly clicked item.

Dialog boxes are either modal or modeless, as described below.

Modal Dialog Boxes

A modal dialog box is one that the user must explicitly dismiss before doing anything else, such as making a selection outside the dialog box or choosing a command. Figure 25 shows a modal dialog box.



MouseGraphics MouseText

Figure 25. A Modal Dialog Box

Because it restricts the user's freedom of action, this type of dialog box should be used sparingly. In particular, the user can't choose a menu item while a modal dialog box is up, and therefore can only do the simplest kinds of text editing.

For these reasons, the main use of a modal dialog box is when it's important for the user to complete an operation before doing anything else.

A modal dialog box usually has at least two buttons: OK and Cancel. clicking on OK or pressing Return dismisses the dialog box and performs the original command according to the information provided; it can be given a more descriptive name than "OK". Clicking on Cancel or pressing Escape dismisses the dialog box and cancels the original command; it must always be called "Cancel".

A dialog box can have other kinds of buttons as well; these may or may not dismiss the dialog box.

[Note to reviewers: Because of the needs of keyboard users, I have tentitively decided to make OK always be selectable with Return and Cancel always be selectable with Escape. This is in conflict with the MacIntosh guideline that follows:

"One of the buttons in the dialog box may be outlined boldly. The outlined button is the default button; if no button is outlined, then

the OK button is the default button. The default button should be the safest button in the current situation. Pressing the Return or Enter key has the same effect as clicking the default button. If there is no default button, then Return and Enter have no effect."

If you have any ideas on how we could retain the MacIntosh guideline and still make the boxes reasonable for the keyboard user, please let me know.]

A special type of modal dialog box is one with no buttons. This type of box is just to inform the user of a situation without eliciting any response. Usually, it would describe the progress of an ongoing operation. Since it has no buttons, the user has no way to dismiss it. Therefore, the application must leave it up long enough for the user to read it before taking it down again.

Modeless Dialog Boxes

A modeless dialog box allows the user to perform other operations without dismissing the dialog box. Figure 26 shows a modeless dialog box.

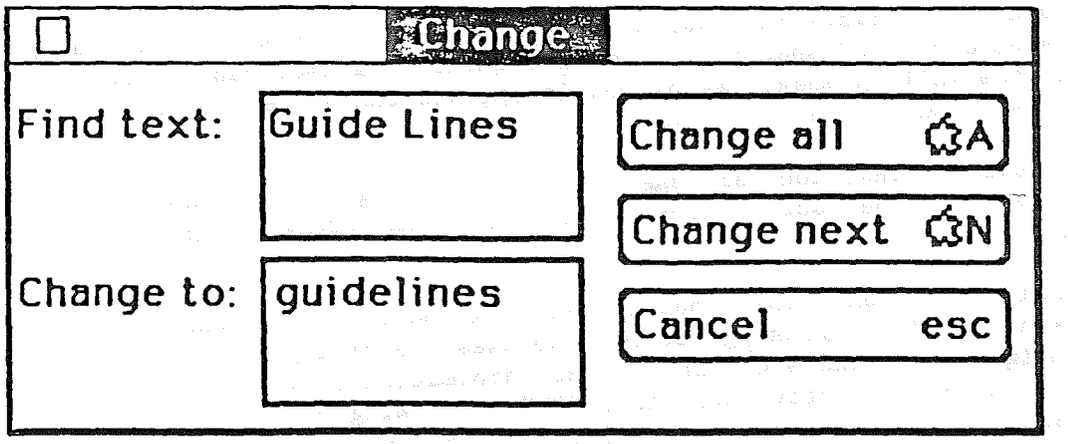


Figure 26. A Modeless Dialog Box

A modeless dialog box is dismissed by clicking in the close box or by choosing Close when the dialog is active. The dialog box is also dismissed implicitly when the user chooses Quit. It's usually a good idea for the application to remember the contents of the dialog box after it's dismissed, so that when it's opened again, it can be restored exactly as it was.

Controls work the same way in modeless dialog boxes as in modal dialog boxes, except that buttons never dismiss the dialog box. In this

context, the OK button means "go ahead and perform the operation, but leave the dialog box up", while Cancel usually terminates an ongoing operation.

A modeless dialog box can also have text fields; since the user can choose menu commands, the full range of editing capabilities can be made available.

Alerts

An alert box looks like a modal dialog box, except that it's somewhat narrower and appears lower on the screen. An alert box is primarily a one-way communication from the system to the user; the only way the user can respond is by clicking buttons. Therefore alert boxes might contain dials and buttons, but usually not text fields, radio buttons, or check boxes. Figure 27 shows a typical alert box.

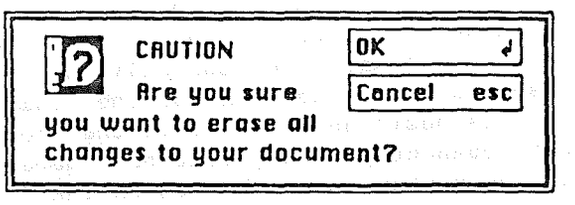


Figure 27. An Alert Box

How the buttons in an alert box are labeled depends on the nature of the box. If the box presents the user with a situation in which no alternative actions are available, the box has a single button that says OK. Clicking this button means "I have read the alert." If the user is given alternatives, then typically the alert is phrased as a question that can be answered "yes" or "no". In this case, buttons labeled Yes and No are appropriate, although some variation such as Save and Don't Save is also acceptable. OK and Cancel can be used, as long as their meaning isn't ambiguous.

[As noted above, the following paragraph needs to be excised if we go with the "OK equals Return" scheme.]

The preferred (safest) button to use in the current situation is boldly outlined. This is the alert's default button; its effect occurs if the user presses Return or Enter.

For further information on beeps, the types of alert messages, and how and when to write one, read Alert Messages in the Generic Interface

section.

View Boxes — SUBJECT TO CHANGE! —

MouseText-based programs have a restricted ability to deliver what-you-see-is-what-you-get: standard printer features such as bold, underline, and super- and subscript are impossible to produce on the text screen. To enable the user to see where such changes begin and end, flank the changed area with a pair of solid-diamond icons. These icons let the user know that there is some control information at those locations, but not what the information is.

The user can "look through the keyhole" of the diamond icon by opening a view box. A view box looks like an alert box, except it has no buttons. It displays the information hidden within the solid-diamond icon:

"Begin Underline" "End Bold Text" "Begin Plain Text"

The view box is normally located beginning two lines below the icon's position, so that the user need look no further than necessary to see it. (The single unaffected line below the diamond enables the user to continue seeing the diamond in context.) Horizontally, approximately one-third of the view box should lie to the left of the diamond; two-thirds, to the right. This position relative to the icon should be as consistent as possible throughout your application: move it above or slide it toward one side only when you lack room on the display for its normal position.

The contents of a view box cannot be edited. The diamond icon itself (along with its contents) can be deleted in the same manner as any other text character, and a new icon can be created with the appropriate menu command.

The user can open a view box in one of two ways:

- By moving the insertion point to the left of the diamond icon. In this position, the blinking underscore is directly beneath the diamond icon. (Recall that the insertion point itself lies between characters; the blinking underscore is a necessary compromise with the text hardware and appears under the character to the right of the insertion point.) The view box remains open until the insertion point is moved away from the icon, the mouse cursor is moved, or any valid short-cut key is pressed. In all of these cases, the view box remains closed until the user formally reopens it: it is not suddenly reopened when the mouse stops moving.

- By covering the diamond with the mouse cursor. It is irrelevant whether the mouse button is pressed or not: if the mouse cursor is over the diamond, the view box is opened. This enables the user to quickly view all diamonds on the display without having to relocate the insertion point, and lets the user see the

information in the same way whether simply moving the cursor around or actively marking a selection. The view box remains open until either the mouse cursor or the insertion point moves, or until any valid shortcut key is pressed.

DO'S AND DON'TS OF A FRIENDLY USER INTERFACE

Do:

- Let the user have as much control as possible over the appearance of objects on the screen--their arrangement, size, and visibility.
- Use verbs as menu commands.
- Make alert messages self-explanatory.
- Use controls and other graphics instead of just menu commands.
- Take the time to use good graphic design; it really helps.

Don't:

- Overuse modes, including modal dialog boxes.
- Require using the keyboard for an operation that would be easier with the mouse, or require using the mouse for an operation that would be easier with the keyboard.
- Change the way the screen looks unexpectedly, especially by scrolling automatically more than necessary.
- Make up your own menus and then give them the same names as standard menus.
- Take an old-fashioned prompt-based application and pass it off as a mouse-based application.

User Input Routine
External Reference Specification

Lou Infeld

04/10/85

TABLE OF CONTENTS

1. Introduction

Most Applications at one point or another require that the user key in some textual information. In the past there has been little standardization in the way that an "Input Routine" interfaces with the user. Pascal and Basic each have different user input conventions. In fact they completely contradict each other; a user has to completely relearn how to interact with either language after using the other. Many applications use the "Input Routine" built into the language environment being used. Other applications use independently developed "Input Routines" which are more sophisticated and user friendly. However, the poor user of several applications has different interfaces to contend with, each with its own standards and idiosyncrasies.

To try to solve these problems, Apple Computer has published several documents encouraging "standard" design guidelines including how an "Input Routine" should look and behave. Now this "User Input Routine" is being made available to Apple // developers. It encorporates all the standards proposed by Apple Computer and is available for the following environments:

Apple // Assembler	(with or without Console Driver)
Apple // AppleSoft	(with or without Console Driver)
Apple // Pascal	(with Console Driver)

2. General Description

The "User Input Routine" attempts to fulfill the standards published in the "Apple //e Design Guidelines" manual (pp 24-37), Bruce Tognazzini's memo "UPDATE: Human Interface Design Guidelines" dated August 9, 1983, as well as de facto standards used in the popular AppleWorks program.

The "User Input Routine" is called by the application with a string variable containing a default (can be null) as well as the maximum number of characters that will fit in the string variable. A "string variable" is basically a buffer in which the first byte contains the "length" of the string. The following bytes are the actual characters in the string.

The "User Input Routine" will display a field on the screen consisting of the default string followed by a series of "fill" characters. A cursor will be visible to the right of the default string. The cursor is the "Insert Cursor" as described in the Guidelines. When this cursor is present, typing any printing character will place that character in the field at the current cursor position. All characters in the field to the right of the cursor are shifted one position. If the user presses the CONTROL key and "E" together, the "Replace Cursor" appears. When this cursor is present, typing any printing character will place that character in the field replacing the current character under the cursor. Pressing the CONTROL key and "E" again will return the "Insert Cursor".

The user can edit the field by adding or replacing characters or by using editing commands. When the user is satisfied with the string in the field, he presses the RETURN key. This will terminate the "User Input Routine" and return control back to the application. The user's response will be in the string variable specified when the "User Input Routine" was called.

If the application specifies a string variable that can contain more characters than the width of the field, the "User Input Routine" will retain characters that "fall off" the right edge of the field. These characters will "reappear" if characters in the field are deleted.

The following editing commands are supported:

LEFT-ARROW	Moves cursor left within field
RIGHT-ARROW	Moves cursor right within field
CONTROL-D	Deletes character to the left of the cursor
DELETE	Deletes character to the left of the cursor
CONTROL-F	Deletes character under the cursor (Forward Delete)
CONTROL-E	Toggle between insert and delete cursors
CONTROL-X	Deletes all characters in the field
CONTROL-Y	Deletes all characters from present cursor position to end of field (including characters saved by insert)
CONTROL-Z	Restores default string

3. Customization and Advanced Uses

In general, the "User Input Routine" will behave as described in section 2. However, the "User Input Routine" can be customized to the particular needs of the application. A structure called the Information Block is used as a conduit between the application and the "User Input Routine". The application tells the "User Input Routine" how to react to the user's keystrokes and conversely the "User Input Routine" tells the application all about its current status.

If a viewport (window) has been defined, the "User Input Routine" will respect it with the one restriction: the last two positions in the window can not be included in the input field. This restriction is necessary to eliminate scrolling and wrapping problems. A field as large as 254 characters can be specified.

Normally, when the RETURN key or the ESCAPE key is pressed, the "User Input Routine" will terminate with the Input String set to the characters currently in the field on the screen (without the fill characters). However, other terminating characters can be configured to cause termination instead or in addition to RETURN and ESCAPE. Also the "User Input Routine" can be interrupted rather than terminated. In this case, when the "User Input Routine" is called again, it continues in the state it was in when it was interrupted (assuming the application program has not changed any parameters in the Information Block). This feature is useful for a help facility. A help character (e.g. Open Apple-?) can be configured to interrupt the "User Input Routine" for a help message in the middle of editing.

Up to 20 characters can be specified as termination characters. For each termination character, the application can specify whether the Open Apple or Solid Apple key must be pressed with the character. Additionally, for each termination character, the application can specify whether to completely terminate the "User Input Routine" or just "interrupt" it temporarily.

An "immediate" mode is optionally available that allows the application to constantly gain control during the input process. This feature can be used by the application, for instance, to update a clock display, check for mouse movements or run in demonstration mode.

4. Information Block

4.1 Format

The Information Block is divided into three logical sections: General Information, Termination Information, and Internal Information.

```
max_terms       equ     20      ;Maximum number of terminators

Input_Info      equ     *
;                                General Information
;                                -------------------
width           db      0       ;Width of the field on the screen
fill_char       db      " "     ;Fill character
mouse_fill      db      0       ;0-use "fill_char" as fill character
                                ;1-use MouseText ghost underline
cursor          db      0       ;current cursor being used
                                  ;0-insert cursor
                                  ;1-replacement cursor
control         db      0       ;0-Control chars will be ignored
                                ;1-Control chars allowed as input
beep            db      0       ;0-errors will not be beeped
                                ;1-errors will be beeped
immediate       db      0       ;0-calling routine gets control after the
                                ;  complete input is keyed in by user
                                ;1-calling routine gets control after each
                                ;  keypress check
entry_type      db      0       ;Indicates type of entry into routine
                                ;0-initial entry
                                ;1-interrupt re-entry
                                ;2-immediate re-entry
bord_ch         db      0       ;char to blink outside of field

;                                Termination Information
;                                -----------------------
exit_type       db      0       ;Indicates which termination condition
                                ;  occurred
                                ;0-not terminated yet
                                ;not 0-index into terminating char list
last_event      db      0       ;last event type (not used)
last_ch         db      0       ;character user keyed in
last_mod        db      0       ;keypress modifier
n_chars         db      0       ;Number of terminator chars currently
                                ;  defined

                                ;The next 3 items define what keystrokes
                                ;  will terminate or interrupt the routine.

char_list       ds      max_terms ;Chars which will terminate input
mod_list        ds      max_terms ;Modifiers for each char in"char_list"
                                    ;0-none
                                    ;1-Open Apple
                                    ;2-Solid Apple
                                    ;3-Either Open or Solid Apple
term_list       ds      max_terms ;Termination types for each char in
```



```
                                  ;  "char_list"
                                     ;0-terminate input
                                     ;1-interrupt input

;                                 Internal Information
;                                 --------------------

origin_x        db      0       ;x coordinate of start of field
origin_y        db      0       ;y coordinate of start of field
cursor_x        db      0       ;x coordinate of cursor in field
cursor_y        db      0       ;y coordinate of cursor in field
cursor_pos      db      0       ;position     of cursor in field (1..width)
input_length    db      0       ;length of Input String (incl invisib part)
slow_blink      dw      0       ;slow blink rate
fast_blink      dw      0       ;fast blink rate
```


4.2 Description

4.2.1 General Information section

4.2.1.1 "width"

This parameter tells the "User Input Routine" how wide to make the field on the screen. When the "User Input Routine" is called, it displays the default value in the Input String on the screen at the current cursor position. If there is any room left in the field on the screen, fill characters are displayed. The parameter "fill_char" is used as the fill character. The number of fill characters displayed is "width" minus "length of Input String". "Width" is initially 254 characters.

If the value of "width" is greater than the number of character positions from the start of the field to the end of the window -2, the "User Input Routine" will reduce "width" accordingly.

4.2.1.2 "fill char"

This is the fill character that is used in the field. "Fill_char" is initially the blank character.

4.2.1.3 "mouse fill"

If this parameter is 1, the MouseText ghost underline is used as the fill character. If "mouse_fill" is 0 the character in "fill_char" will be used as described above. "Mouse_fill" is initially 0.

It is the application's responsibility to determine whether MouseText is available in ROM before using this option. The following algorithm can be used to determine whether MouseText is available:

If memory location $FBB3 contains $06
AND
memory location $FBC0 does not contain $EA

then MouseText is available

4.2.1.4 "cursor"

This parameter represents the current cursor type being used. If it is 0, the "Insert Cursor" is in effect. If it is 1, the "Replace Cursor" is in effect. If the user presses CONTROL and "E", this parameter changes value. The application can force the "User Input Routine" to start with either of the cursor types by setting "cursor" accordingly before calling the routine. "Cursor" is initially 0.

4.2.1.5 "control"

If this parameter is 1, control characters (ASCII values less than 32) are allowed as input from the keyboard. To insert a control character, the user must press the CONTROL key, the Open Apple key and the corresponding alphabetic key. The alphabetic character is obtained by added 64 to the ASCII value of the control that is desired. The actual value inserted in the string is the ASCII value + 128 which will appear on the screen as the inverse of the corresponding character. For example, to insert the Carriage Return character (ASCII 13), the user presses CONTROL, Open Apple and "M" (ASCII 77). The screen

will show an inverse "M" and the string will contain the value 205 (77+128). If "control" is 0, control characters will not be allowed and will result in a beep. Note that editing characters and termination characters are not affected by the setting of "control". "Control" is initially 0.

4.2.1.6 "beep"

If this parameter is 1, any illegal keypresses will cause the "User Input Routine" to beep. If this parameter is 0, there will be no beeps. "Beep" is initially 1.

4.2.1.7 "immediate"

If this parameter is 1, the "User Input Routine" will return to the application program after each keypress check. When the application next calls the "User Input Routine", it will be considered an "immediate" re-entry. If this parameter is 0, the "User Input Routine" will return to the application program only after a termination character is pressed.

During "immediate" processing, the application can tell whether a key has been pressed by checking the parameter "last_key". If it is not 0, a key has been pressed and its ASCII value is in that parameter (its corresponding keypress modifier is in "last_mod"). When the "User Input Routine" is re-entered, it will check "last_key" and "last_mod". If there is a keystroke, it will "process" it, otherwise it will look for the next keystroke. The application can therefore "process" the keystroke before the "User Input Routine". At this point, the application can leave the keystroke intact and re-enter the "User Input Routine" which will also "process" the keystroke. Alternatively, the application can set "last_key" and "last_mod" to 0 which will cause the keystroke to be ignored by the "User Input Routine".

Applications using "immediate" mode have the additional responsibility to keep the cursor blinking at the correct rate. See the description of "slow_blink" and "fast_blink" for the necessary considerations.

"Immediate" is initially 0.

4.2.1.8 "entry type"

This parameter tells the "User Input Routine" what type of entry is being made. If the value of "entry_type" is 0, this is an initial entry and a new field is established. If the value is 1, the routine assumes it is being re-entered after an interrupt termination. If the value is 2, the routine assumes it is being re-entered after "immediate" processing by the application. This parameter is managed by the "User Input Routine" and normally does not need to be changed by the application.

4.2.1.9 "bord ch"

This character will be used by the "User Input Routine" as the blink character when the cursor is outside of the field. This condition occurs when the field is completely filled in. "Bord_ch" is initially blank.

4.2.2 Termination Information section

4.2.2.1 "exit type"

When the "User Input Routine" terminates, this parameter contains the "type" of termination that occurred. Termination characters are numbered from 1 to 20. "Exit_type" will contain the number of the termination character that caused the termination. If "exit_type" is 0, this indicates that the "User Input Routine" has not terminated yet (i.e."immediate" mode is in effect).

4.2.2.2 "last event"

This parameter is not currently used.

4.2.2.3 "last ch"

This parameter contains an ASCII value if the last keypress check sensed a keystroke or 0. It is useful for applications using the "immediate" mode of the "User Input Routine".

4.2.2.4 "last mod"

This parameter contains the keystroke modifier if a keystroke was sensed by the last keypress check. Otherwise it is 0. The possible values of "last_mod" are:

0 - no modifier pressed
1 - Open Apple key pressed together with key
2 - Solid Apple key pressed together with key
3 - Either Apple keys pressed together with key

4.2.2.5 "n chars"

This parameter is the number of termination characters that have been configured. "N_chars" is initially 2 (for RETURN and ESCAPE).

4.2.2.6 "char list"

"Char_list" is a 20 byte table containing the ASCII values of the configured termination characters. For the alphabetic characters "A" to "Z", only the upper case ASCII values need be in the table.

Only the first "n_chars" bytes are looked at by the "User Input Routine". The first 2 bytes in this list are initially 13 and 27 respectively (these values are the ASCII codes for RETURN and ESCAPE).

4.2.2.7 "mod list"

"Mod_list" is a 20 byte table which specifies what keystroke modifiers are needed for each termination character to be recognized. A value of 0 indicates that no modifiers can be pressed. A value of 1 indicates that the Open Apple key must be pressed together with the termination character. A value of 2 indicates that the Solid Apple key must be pressed. A value of 3 indicates that either the Open Apple or Solid Apple keys must be pressed together with the termination character.

4.2.2.8 "term list"

"Term_list" is a 20 byte table which specifies the termination type of each termination character. A value of 0 indicates that a normal termination will occur when the termination character (along with any keystroke modifiers) is pressed. A value of 1 indicates that an "interrupt" termination will occur.

4.2.3 Internal Information section

4.2.3.1 "origin x" and "origin y"

These parameters contain the relative coordinates of the start of the field within the current window. When the "User Input Routine" is entered initially (not reentered after an "interrupt" termination or "immediate" termination), "origin_x" and "origin_y" are set to the current relative cursor position.

4.2.3.2 "cursor x" and "cursor y"

These parameters contain the relative coordinates of the cursor within the current window. When the "User Input Routine" is entered initially, the cursor is positioned after the default Input String in the field and "cursor_x" and "cursor_y" are set to that coordinate location.

4.2.3.3 "cursor pos"

This parameter contains the relative position of the cursor in the field (not in the window). The value of "cursor_pos" is in the range 1.."width".

4.2.3.4 "input length"

This parameter contains the current length of the Input String. If the maximum size of the Input String is larger than the width of the field on the screen, the "User Input Routine" uses the "invisible" part of the Input String to save characters that were "pushed" out of the field by insertions. Therefore, "input_length" may have a value greater than "width". However, in this case, the length of the Input String actually returned to the user is still in the range 1.."width". The returned length of the Input String is contained in the first byte of the Input String.

4.2.3.5 "slow blink" and "fast blink"

These parameters are the count-down timers used to get the correct blinking frequency for the cursor. The cursor should blink at 80 cycles per minute with one phase taking twice as long as the other. Assuming that the cursor is "under" a character in the field and the "insert" cursor is on, the character should be visible twice as long as the underline. If the "replace" cursor is on, the inverse character should be visible twice as long as the normal character. The initial values of "slow_blink" and "fast_blink" will cause the correct cursor blink rate. However, if "immediate" mode is turned on, the cursor will no longer blink at the correct rate because the application program will get control in the middle of the blink loop. It is up to the application program to change "slow_blink" and "fast_blink" so that the cursor will again blink at the correct rate.

4.3 Default values

The default values of the Input Information Block are:

```
width=254;
fill_char=' ';
mouse_fill=0;
cursor=0;
control=0;
beep=1;
immediate=0;
entry_type=0;
exit_type=0;
bord_ch=' ';
last_event=0;
last_ch=0;
last_mod=0;
n_chars=2;
char_list[1]=chr(13);     {RETURN}
char_list[2]=chr(27);     {ESCAPE}
mod_list[1]=0;
mod_list[2]=0;
exit_list[1]=0;
exit_list[2]=0;
origin_x=  |
origin_y=  |      Current relative cursor coordinate in window
cursor_x=  |      defined by Console Driver
cursor_y=  |
cursor_pos=0;
input_length=0;
slow_blink= |     Values necessary to blink cursor
fast_blink= |     80 times per minute
```


5. Interface Description

5.1 Apple // Pascal

5.1.1 General Description

The Apple // Pascal version of the "User Input Routine" is part of the Console Driver and therefore requires that the Pascal environment be loaded with the correct Attach files. The Console Driver is configured as unit number 130.

To access the "User Input Routine", a Pascal program must make calls to the Console Driver. Three "unitstatus" calls are provided to initialize, set and get the Information Block. The actual call to the "User Input Routine" is in the form of a "unitread".

Sections 5.1.3 to 5.1.6 will describe each of the Console Driver calls in detail.

5.1.2 Format of the Information Block

The following is the Pascal equivalent of the Information Block:

```
const max_terms=20;          {Maximum number of terminators}
type  byte=0..255;
var   Input_Info:packed record

                             {General Information}
                             {-------------------}

      width:byte;            {Width of the field on the screen}
      fill_char:char;        {Fill character}
      mouse_fill:byte;       {0-use "fill_char" as fill character
                              1-use MouseText ghost underline}
      cursor:byte            {current cursor being used
                                0-insert cursor
                                1-replacement cursor}
      control:byte;          {0-Control chars will be ignored
                              1-Control chars allowed as input}
      beep:byte;             {0-errors will not be beeped
                              1-errors will be beeped}
      immediate:byte;        {0-calling routine gets control after the
                                   complete input is keyed in by user
                              1-calling routine gets control after each
                                   printable character is input}
      entry_type:byte;       {Indicates type of entry into routine
                                0-initial entry
                                1-interrupt re-entry
                                2-immediate re-entry}
      bord_ch:char;          {char to blink outside of field}

                             {Termination Information}
                             {-----------------------}

      exit_type:byte;        {Indicates which termination condition occurred
                                0-not terminated yet
                                not 0-index into terminating char list}
      last_event:byte;       {last event type (not used)}
      last_ch:char;          {character user keyed in}
      last_mod:byte;         {keypress modifier}
      n_chars:byte;          {Number of termination chars defined}

                             {The next 3 items define what keystrokes will
                              terminate or interrupt the routine.  The case
                              of each character is ignored}

      char_list:packed array [1..max_terms] of char;
                             {Chars which will terminate input}
      mod_list :packed array [1..max_terms] of byte;
                             {Modifiers for each char in "char_list"
                                0-none
                                1-Open Apple
                                2-Solid Apple
```



```
                              3-Either Open or Solid Apple}
term_list:packed array [1..max_terms] of byte;
                            {Termination types for each char in "char_list"
                              0-terminate input
                              1-interrupt input}

                            {Internal Information}
                            {--------------------}


origin_x :    byte;     {x coordinate of start of field}
origin_y :    byte;     {y coordinate of start of field}
cursor_x:     byte;     {x coordinate of cursor in field}
cursor_y:     byte;     {y coordinate of cursor in field}
cursor_pos:   byte;     {position     of cursor in field (1..width)}
input_length:byte;      {length of Input String (incl invisible part)}
slow_blink:integer;     {slow blink rate}
fast_blink:integer;     {fast blink rate}

end {Input_Info};
```

The text of this data structure is in the file "INPUT.INFO.TEXT" on the release disk.

5.1.3 Initializing Input Information

To set the User Input Information Block to its default values, call the procedure:

```
init_mode:=24577;       {Console Driver command $6001}
unitstatus(130,Input_Info,init_mode);
```

OR if the console driver is also to be initialized use:

```
unitclear(130);
```

Note: the variable "Input Info" in the unitstatus call above is not actually used by the "User Input Routine". It is needed in the "unitstatus" call because of its parameter structure.

An automatic "unitclear" is performed by the Pascal system when it is booted.

5.1.4 Retrieving Input Information

To get the current settings of all the Input Information parameters, call the procedure:

```
get_info:=16385;        {Console Driver command $4001}
unitstatus(130,Input_Info,get_info);
```

where "Input_Info" is a record with the format specified in 5.1.2.

5.1.5 Setting Input Information

To change the data in the User Input Information Block, call the procedure:

```
set_info:=8193;        {Console Driver command $2001}
unitstatus(130,Input_Info,set_info);
```

where "Input_Info" is a record with the format specified in 5.1.2. If this call is never made, the "User Input Routine" uses the default values.

Note that changing any parameters in the record will not have any effect until the "unitstatus" call is made.

5.1.6 Calling the User Input Routine

To call the "User Input Routine", call the procedure:

```
unitread(130,Input_Str,max_length);
```

where "Input_Str" is a string supplied by the calling routine where the "User Input Routine" will store the user's keystrokes. "Max_length" specifies the maximum number of characters which will fit in the string (usually 80 unless "Input_Str" is defined as an extended string).

If the Input String has an initial value, the "User Input Routine" will assume that it is a default value and display it.

Upon return from "unitread", IORESULT will contain the "exit_type" value which is the index into the "char_list" of terminating characters.

5.1.7 Examples

The program "Demo" is a good example of the "User Input Routine" in action. It can be used to try out many of the features.

In the simplest use of the "User Input Routine", the application displays a question on the screen using the Console Driver and then calls the "User Input Routine" for the answer. The following program segment illustrates the above:

```
VAR
  question,answer:string;
  ...
  ...
  question:='What is your name ? ';
  answer:='';
  unitwrite(130,question[1],length(question));
  unitread(130,answer,80);
```

If the application wants to provide a default name:

```
VAR
  question,answer:string;
  ...
  ...
  question:='What is your name ? ';
  answer:='Fred';
  unitwrite(130,question[1],length(question));
  unitread(130,answer,80);
```

If the application wants to provide the user with a small visible field:

```
CONST
  get_info=16385;        {Console Driver command $4001}
  set_info=8193;         {Console Driver command $2001}
VAR
  question,answer:string;
  Input_Info:packed record
             {use record structure in 5.1.2}
```



```
            end;
...
...
{Get the current Information Block}

unitstatus(130,Input_Info,get_info);

{Change the desired parameters}

Input_Info.width:=12;
Input_Info.fill_char:='.';

{Set the updated Information Block}

unitstatus(130,Input_Info,set_info);

{The rest of the logic is the same}

question:='What is your name ? ';
answer:='Fred';
unitwrite(130,question[1],length(question));
unitread(130,answer,80);
```


5.2 Basic

The Basic version of the "User Input Routine" is in the form of several AMPERSAND ('&') calls. The AMPERSAND facility allows a machine-language program to be loaded from a Basic program and its functions called in the form of Basic commands. The following commands are available:

```
&INITINPUT      --- Initialize Information Block
&GETINFO(IB%)   --- Get Information Block
&SETINFO(IB%)   --- Set Information Block
&INPUT(IS$)     --- Call "User Input Routine"
&EXITINPUT      --- Removes package from Ampersand hooks
```

The release disk contains the "User Input Routine" in a "relocatable" file "INPUT.REL". The EdAsm RLOAD facility must be used to load "INPUT.REL" from within the application program.

5.2.1 &INITINPUT

This call will initialize the Information Block to its default values. See 4.3 for the default values associated with each parameter.

5.2.2 &GETINFO(IB%)

This call retrieves the current Information Block and stores it into the integer array IB%. The array IB% should be dimensioned for at least 22+3*max_terms integers where "max_terms" is currently 20. The contents of each integer in IB% is as follows:

IB%(1) =	width	IB%(2) =	fill_char	IB%(3) =	mouse_fill
IB%(4) =	cursor	IB%(5) =	control	IB%(6) =	beep
IB%(7) =	immediate	IB%(8) =	entry_type	IB%(9) =	bord_ch
IB%(10) =	exit_type	IB%(11) =	last_event	IB%(12) =	last_ch
IB%(13) =	last_mod	IB%(14) =	n_chars	IB%(15) =	char_list
IB%(35) =	mod_list	IB%(55) =	term_lis	IB%(75) =	origin_x
IB%(76) =	origin_y	IB%(77) =	cursor_x	IB%(78) =	cursor_y
IB%(79) =	cursor_pos	IB%(80) =	input_length	IB%(81) =	slow_blink
IB%(82) =	fast_blink				

5.2.3 &SETINFO(IB%)

This call moves the contents of the integer array IB% into the Input Information Block. The format of IB% is assumed to be the same as described above.

5.2.4 &INPUT(IS$)

This is the actual call to the "User Input Routine". The variable "IS$" is a string which contains the default Input String and will contain the result of the user's input.

5.2.5 &EXITINPUT

This call will terminate the "User Input Routine" and disconnect the ampersand package.

5.2.6 Examples

User Input Routine

The program "Demo" is a good example of the "User Input Routine" in action. It can be used to try out many of the features.

In the simplest use of the "User Input Routine", the application displays a question on the screen and then calls the "User Input Routine" for the answer. The following program segment illustrates the above:

```
PRINT CHR$(4);"BLOAD INPUT.OBJ"
PRINT "What is your name ? ";
&INPUT(IS$)
```

If the application wants to provide a default name:

```
PRINT CHR$(4);"BLOAD INPUT.OBJ"
PRINT "What is your name ? ";
IS$="Fred"
&INPUT(IS$)
```

If the application wants to provide the user with a small visible field:

```
DIM IB%(82)
PRINT CHR$(4);"BLOAD INPUT.OBJ"
&GETINFO(IB%)
IB%(1)=20:REM width
IB%(2)=".":REM fill_char
&SETINFO(IB%)
PRINT "What is your name ? ";
IS$="Fred"
&INPUT(IS$)
```


5.3 Assembler

The Assembler version of the "User Input Routine" provides a set of calls similar to ProDOS MLI calls which provide the following functions:.

Initializing Input Information
Retrieving Input Information
Setting Input Information
Calling the User Input Routine

The release disk contains an "absolute" binary file "INPUT.OBJ" and a "relocatable" file "INPUT.REL". "INPUT.OBJ" was generated from "INPUT.REL" with the starting address $4000. If this starting address is not satisfactory for the application, the program "RELOCATOR" must be used to generate a new "absolute" file which starts at the desired location.

5.3.1 Format of the Information Block

The following is the Assembler equivalent of the Information Block:

```
maxterms        equ     20      ;Maximum number of terminators

InputInfo       equ     *
;                                General Information
;                                -------------------
width           db      0       ;Width of the field on the screen
fillchar        db      " "     ;Fill character
mousefill       db      0       ;0-use "fillchar" as fill character
                                ;1-use MouseText ghost underline
cursor          db      0       ;current cursor being used
                                  ;0-insert cursor
                                  ;1-replacement cursor
control         db      0       ;0-Control chars will be ignored
                                ;1-Control chars allowed as input
beep            db      0       ;0-errors will not be beeped
                                ;1-errors will be beeped
immediate       db      0       ;0-calling routine gets control after the
                                ;  complete input is keyed in by user
                                ;1-calling routine gets control after each
                                ;  printable character is input
entrytype       db      0       ;Indicates type of entry into routine
                                ;0-initial entry
                                ;1-interrupt re-entry
                                ;2-immediate re-entry
bordch          db      0       ;char to blink outside of field

;                                Termination Information
;                                -----------------------
exittype        db      0       ;Indicates which termination condition
                                ;  occurred
                                ;0-not terminated yet
                                ;not 0-index into terminating char list
lastevent       db      0       ;last event type (not used)
lastch          db      0       ;character user keyed in
lastmod         db      0       ;keypress modifier
```



```
nchars          db      0       ;Number of terminator chars currently
                                ;  defined

                                ;The next 3 items define what keystrokes
                                ;  will terminate or interrupt the routine.

charlist        ds      maxterms ;Chars which will terminate input
modlist         ds      maxterms ;Modifiers for each char in"charlist"
                                   ;0-none
                                   ;1-Open Apple
                                   ;2-Solid Apple
                                   ;3-Either Open or Solid Apple
termlist        ds      maxterms ;Termination types for each char in
                                 ;  "charlist"
                                   ;0-terminate input
                                   ;1-interrupt input

;                                 Internal Information
;                                 --------------------

originx         db      0       ;x coordinate of start of field
originy         db      0       ;y coordinate of start of field
cursorx         db      0       ;x coordinate of cursor in field
cursory         db      0       ;y coordinate of cursor in field
cursorpos       db      0       ;position     of cursor in field (1..width)
inputlength     db      0       ;length of Input String (incl invisib part)
slowblink       dw      0       ;slow blink rate
fastblink       dw      0       ;fast blink rate
```

5.3.2 Format of Calls

The "User Input Routine" has only one entry for all the functions. It is located at the beginning of the code. A call is made as follows:

```
        JSR     INPUT
        DB      COMMAND
        DW      PARAMPTR
        BNE     ERROR
```

The label "INPUT" is the starting address of the "User Input Routine". The programmer will determine this location when he relocates the routine in memory. In the application, there should be a statement of the form:

```
        INPUT   EQU     nnnn
```

where "nnnn" is the starting address of the "User Input Routine".

"COMMAND" is a number which specifies which function is requested. "PARAMPTR" is a two byte pointer to a parameter list.

When the "User Input Routine" returns to the calling program, the carry flag will be set if an error has been detected. The only possible error that is detected by the "User Input Routine" is an illegal command error (3). This occurs if "COMMAND" is not one of the available function numbers.

The calling program should check the carry flag (as in the BNE instruction above) and report the appropriate error. The actual error type is passed to the calling program in the A-register.

5.3.3 Initializing Input Information

This call will initialize the Information Block to its default values. See 4.3 for the default values associated with each parameter. This call has the following format:

```
        JSR     INPUT
        DB      10              ;command number for Initialize
        DW      0
```

5.3.4 Retrieving Input Information

This call will retrieve the current contents of the Input Information Block. The format of the call is:

```
        JSR     INPUT
        DB      11              ;command number for Get Input Information
        DW      INPUTINFO
```

where "INPUTINFO" is the address of a buffer where the contents of the Information Block is to be moved. This buffer will have the format as described in 4.1.

5.3.5 Setting Input Information

This call will set the Input Information Block to values in the specified buffer. The format of the call is:

```
        JSR     INPUT
        DB      12              ;command number for Set Input Information
        DW      INPUTINFO
```

where "INPUTINFO" is the address of the buffer. This buffer must have the format as described in 4.1.

5.3.6 Calling the User Input Routine

This call will perform the actual input. The format of the call is:

```
        JSR     INPUT
        DB      13              ;command number for Input
        DW      PARAM
```

where the format of "PARAM" is:

```
PARAM   DW STRING
        DB maxlength
STRING  STR "This is the default"
```

Upon return from this call, the A register will contain the "exittype".

5.3.7 Examples

The program "Demo" is a good example of the "User Input Routine" in action. It can be used to try out many of the features.

In the simplest use of the "User Input Routine", the application displays a question on the screen and then calls the "User Input Routine" for the answer. The following program segment illustrates the above:

```
QUESTION STR "What is your name ? "
ANSWER   STR ""
         DS  81-*+ANSWER
MAXLEN   EQU *-ANSWER-1
PARAM    DW  ANSWER
         DB  MAXLEN
         ...
         ...
;
;        display question
;
         LDY #0
LOOP     LDA QUESTION+1,Y
         JSR $FDED           ;display the char
         INY
         CPY QUESTION
         BCC LOOP
;
;        get answer
;
         JSR INPUT
         DB  13
         DW  PARAM
```

If the application wants to provide a default name:

```
QUESTION STR "What is your name ? "
ANSWER   STR "Fred"
         DS  81-*+ANSWER
MAXLEN   EQU *-ANSWER-1
PARAM    DW  ANSWER
         DB  MAXLEN
         ...
         ...
;
;        display question
;
         LDY #0
LOOP     LDA QUESTION+1,Y
         JSR $FDED           ;display the char
         INY
         CPY QUESTION
         BCC LOOP
;
;        get answer
;
```



```
        JSR INPUT
        DB  13
        DW  PARAM
```

If the application wants to provide the user with a small visible field:

```
QUESTION STR "What is your name ? "
ANSWER   STR "Fred"
         DS  81-*+ANSWER
MAXLEN   EQU *-ANSWER-1
PARAM    DW  ANSWER
         DB  MAXLEN
INPUTINFO DS 84
         ...
         ...
;
;        get current Information Block
;
         JSR INPUT
         DS  11
         DW  INPUTINFO
;
;        change values in Information Block
;
         LDA #80
         STA INPUTINFO       ;width
         LDA #"."
         STA INPUTINFO+1     ;fillchar
;
;        set Information Block
;
         JSR INPUT
         DS  12
         DW  INPUTINFO
;
;        display question
;
         LDY #0
LOOP     LDA QUESTION+1,Y
         JSR $FDED           ;display the char
         INY
         CPY QUESTION
         BCC LOOP
;
;        get answer
;
         JSR INPUT
         DB  13
         DW  PARAM
```


ConsoleStuff Library
External Reference Specification

11/02/84

Lou Infeld

BETA Release Version

TABLE OF CONTENTS

1. Introduction

The Console Driver provides a quick method for an application to display text on the screen. It provides many commands to quickly move around the screen, blank portions of the screen, divide the screen into windows and other useful facilities. To use these, a buffer containing text as well as console commands has to be sent to the Console Driver.

The ConsoleStuff Library is a Pascal unit that an application can use to build the necessary console buffer which can then be sent to the Apple // Console Driver. Many text formatting routines are in this library as well as other utility routines to write on the screen including Boxes, Message Areas and Help Screens.

Since this unit requires the Apple // Console Driver to be in the application's environment, the Console Driver must be loaded when the Pascal system is booted.

The ConsoleStuff Library contains a large (1024 bytes) Console Buffer which is used to send console data and commands to the Console Driver. Most of the routines in the library build up this buffer until it is full or until the calling routine explicitly requests that the buffer be sent to the Console Driver. Only when the Console Driver gets the buffer does anything happen on the screen. Since the Console Driver writes onto the screen very rapidly, a Pascal application gets machine language speed from its console output. This is in dramatic contrast to the built-in Pascal console interface.

In addition to formatting routines, this library also contains routines which display Boxes for Comments or Help Screens. These Boxes appear "on top" of the screen overwriting whatever was beneath. When the application is finished with these Boxes, they disappear and the text that was underneath reappears.

2. Description

2.1 Interface Constants and Data Structures

Constants and variables defined in the Interface section of the ConsoleStuff unit define mnemonics for the Console Driver commands and the Console Buffer and the Box Comment procedures.

2.1.1 Console Driver commands

Instead of using the numerical values of the Console Driver commands, the ConsoleStuff unit defines a character constant for each command which the application can use. The following character constants are defined as Console Driver commands:

```
c_Reset_View   {chr(01)},    c_Set_View     {chr(02)},
c_Beg_Line     {chr(03)},    c_Restore_View {chr(04)},
c_On_Cursor    {chr(05)},    c_Off_Cursor   {chr(06)},
c_Bell         {chr(07)},    c_L_Cursor     {chr(08)},
c_D_Cursor     {chr(10)},    c_Cl_To_End    {chr(11)},
c_Cl_View      {chr(12)},    c_Return_Cursor{chr(13)},
c_Normal       {chr(14)},    c_Inverse      {chr(15)},
c_DLE          {chr(16)},    c_Horiz_Shift  {chr(17)},
c_Vert_Pos     {chr(18)},    c_Cl_Beg_View  {chr(19)},
c_Horiz_Pos    {chr(20)},    c_Cursor_Move  {chr(21)},
c_D_Scroll     {chr(22)},    c_U_Scroll     {chr(23)},
c_Off_Mouse    {chr(24)},    c_Home_Cursor  {chr(25)},
c_Cl_Line      {chr(26)},    c_On_Mouse     {chr(27)},
c_Escape       {chr(27)},    c_R_Cursor     {chr(28)},
c_Cl_End_Line  {chr(29)},    c_Abs_Pos      {chr(30)},
c_U_Cursor     {chr(31)}
```

Note that the commands "c_On_Cursor" and "c_Off_Cursor" are ignored by the Apple // Console Driver since they affect the Pascal cursor. If these characters are sent to the Pascal console using a "write" command, they will have the desired effect.

2.1.2 MouseText characters

Additional character constants are defined for some useful MouseText characters. To use any of these characters, the "c_On_Mouse" command must first be sent to the Apple // Console Driver. These are the defined variables:

```
{Line drawing characters}

f_R_Side   {chr(95)}, f_L_Side   {chr(90)}, f_U_Side   {chr(95)},
f_D_Side   {chr(76)}, f_Horiz    {chr(83)}, f_Vert     {chr(124)},

{Arrows}

f_U_Arrow  {chr(75)}, f_D_Arrow  {chr(74)}, f_R_Arrow  {chr(85)},
f_L_Arrow  {chr(72)},

{Specials}
```

f_Open_Apple {chr(65)}, f_Closed_Apple{chr(64)}

2.1.3 Console Buffer data structure

The Console Buffer itself is available as an interface variable. Additionally, a set of variables are available which indicate the current status of the buffer:

```
CBuff:   CBuffType;                    {Console Buffer}

{CBuff data structure}

CBuff_Globals:  record
                  size: integer;      {# of characters in Console Buffer}
                  lines:integer;      {# of lines in Console Buffer}
                  width:integer;      {Max width of lines in Console Buffer}
                end;
```

2.1.4 Box Comment data structure

The "BoxComment" procedure displays a one line message on the screen. See 2.3 for a detailed discussion of the procedure. The configuration data structure associated with this procedure are:

```
{Box Comment data structure}

Box_Globals:  record
                Y:     integer;  {Y coordinate of Box}
                Stat:  integer;  {Status to be inserted instead of "&" in
                                  comment}
                Ch:    char;     {Character read if comment ended in "?"}
                Clear: boolean;  {If true, comment will be cleared from
                                  screen. If false, comment remains on
                                  screen until next BoxComment call}
                Beep:  boolean;  {if true, a beep will be sounded with
                                  comment}
                Time:  integer;  {# of secs comment stays on screen if no
                                  keypress}
              end;
```

2.2 Console Buffer procedures

These procedures allow the application to prepare the Console Buffer for subsequent transmission to the Console Driver. Some provide formatting utilities similar to the Pascal "write" function. Some provide Window and Line drawing abilities.

Each procedure will add to the Console Buffer the necessary text and console commands to perform the requested function. When the "CWrite" procedure is called, the Console Buffer is sent to the Console Driver. The Console Driver will interpret each character in the Console Buffer and will either display the character or perform one of its console functions. The Console Buffer is emptied and can be again "filled" by the ConsoleStuff procedures.

Since the Console Driver supports the concept of a "window", all coordinate parameters should be specified relative to the window in effect. The one exception is the "Window" procedure which requires absolute coordinates since it establishes a new window relative to a screen coordinate systeem in which (0,0) indicates the upper left and (79,23) the lower right corners.

2.2.1 CWrite

This procedure sends the Console Buffer to the Console Driver and initializes the Console Buffer data structure to zeroes.

Example: CWrite;

2.2.2 CWriteCh

This procedure adds the specified character to the Console Buffer.

```
Examples:  CWriteCh('a');
           CWriteCh(c_Cl_View);
```

2.2.3 CWriteStr

This procedure adds the specified string to the Console Buffer. Any size string up to 255 characters is allowed.

```
Examples:  CWriteStr('This will be displayed');
           CWriteStr(strvar);
```

2.2.4 CWriteIStr

This procedure adds the specified string to the Console Buffer. However, the string will display in Inverse Mode.

```
Examples:  CWriteIStr('This will be in inverse');
           CWriteIStr(strvar);
```

2.2.5 CWriteln

This procedure is similar to "CWriteStr" except a Carriage Return is added to the Console Buffer after the string.

```
Examples:  CWriteln('This is a title');
           CWriteln('---------------');
```

2.2.6 CWriteNum

The specified integer is converted to an ascii string and added to the Console Buffer. The size of the field and the fill character can be specified. The integer will be right justified in the field unless the field size is 0.

```
Examples:  CWriteNum(10,5,' ');    {resulting field -- "   10"}
           i:=9;
           CWriteNum(i,2,'0');     {resulting field -- "09"}
           CWriteNum(100,0,' ');   {resulting field -- "100"}
```

2.2.7 CGotoxy

Calling this routine adds the Console Driver commands necessary to change the character position to the specified relative coordinates.

```
Example:  CGotoxy(10,10);          {char position changed to (10,10)}
```

2.2.8 CPlace

This routine combines the "CGotoxy" and "CWriteCh" procedures. It effectively puts the specified character into the specified position.

```
Example:  CPlace(10,10,'X');       {char "X" displayed at (10,10)}
```

2.2.9 Window

This procedure changes the Console Driver window to that specified by the given coordinates. The "absolute" coordinates of the upper left corner and the lower right corner must be specified. These coordinates are not checked for validity and illogical values will have strange effects.

```
Example:  Window(10,15,60,20);     {changes window so that upper left
                                    corner is at (10,15) and lower right
                                    corner is at (60,20)}
```

Note that this procedure is the only Console Buffer routine which uses absolute coordinates. All others use coordinates relative to the current window in effect.

2.2.10 Line

This procedure causes a line to be drawn with the specified character between the two sets of coordinates. Only vertical or horizontal lines can be drawn (others will be ignored). The coordinates specified are not checked for validity (other than defining a line) and illogical values will have strange effects.

```
Examples:  Line(5,10,5,20)         {result: vertical line between
                                    coordinates (5,10) and (5,20)}
           Line(10,5,60,5)         {result: horizontal line between
                                    coordinates (10,5) and (60,5)}
```

2.2.11 Box

This procedure draws a box (using MouseText fonts) at the specified coordinates. These coordinates are not checked for validity and illogical values will have strange effects.

```
Example:  Box(10,15,60,20);        {result: box with upper left corner at
                                    (10,15) and lower right corner at
                                    (60,20)}
```

2.3 Box Comment procedure

This procedure places the specified comment on the screen inside of a narrow box.

If the message is not a question (ends with a question mark), the box stays on the screen for a period of time or until any readable key is pressed.

If the message is a question, the box stays on the screen until a key is pressed. This key is assumed to be the answer to the question and is stored in the Box Comment data structure field "Box_Globals.Ch".

If an "&" is embedded in the comment, it is replaced with the ASCII equivalent of the integer in the "Stat" field of "Box_Globals".

Other fields of the Box Comment data structure can be set to configure the "BoxComment" procedure:

```
Y        -- Y coordinate of the Box (default is 21)
Clear    -- If TRUE (default), comment will be cleared from screen
            If FALSE, comment stays on screen until next call
Beep     -- If TRUE (default), a beep will be sounded with comment
Time     -- Number of secs comment stays on screen if no key pressed
            (default is 15)
```

```
Examples:  BoxComment('This is a comment');
           Box_Globals.Stat:=10;
           BoxComment('The status is "&"');
           BoxComment('Do you want to continue (Y/N) ?');
```

2.4 Help procedures

Two procedures are available to aid in displaying Help Screens. The first opens up the Help Screen and the second closes it down and redisplays the original screen contents.

The calling routine first sets up the Console Buffer using the Console Buffer routines without calling the "CWrite" procedure. Next the "OpenHelp" routine is called. It puts a Box on the screen just large enough to contain the Help lines. When the "CloseHelp" procedure is called, the Help Box disappears and the screen environment is restored.

```
Examples:  CWriteln('This is the first line of the Help Screen');
           CWriteln('This is the second line of the Help Screen');
           CWriteln('This is the last line of the Help Screen');
           OpenHelp;
           read(keyboard,ch);
           CloseHelp;
```

2.5 GetXY procedure

This routine returns the current relative coordinates of the character position within the current window as well as the window coordinates themselves. Note that this procedure is not a Console Buffer formatting procedure. Coordinates returned are those currently in effect.

```
Example:  GetXY(x,y,ulx,uly,lrx,lry);     {char position is (x,y) and
                                           upper left corner of window is
                                           (ulx,uly) and lower right
```

```
                                        corner of window is (lrx,lry)}
```

Console Driver/User Input Routine
Release 1.0B1 Notes

Lou Infeld

04/16/85

Version 1.0B1 is the first Beta release for the Console Driver and User Input Routine. Previous versions are cosidered Alpha releases. The following changes were made in the Console Driver and User Input Routines since the last release:

Console Driver

- Documentation corrected -- Several of the control codes were incorrectly specified in the documentation.
- Bug fixed -- Calling the Horizontal, Vertical or Absolute Position commands with values outside of the current window sometimes resulted in positioning the cursor to the top or left side of the window rather than the bottom or right side.
- Bug fixed -- Clearing viewports that are two lines high caused Console Driver to hang.

User Input Routine

- Bug fixed -- Sometimes cursor remnants remained on screen.
- Bug fixed -- Control F didn't work in Pascal version. The fix was to disable all special Pascal control characters including Control @, Control S, Control Z, etc. as well as Control F.
- Standard change -- Control R (restore) changed to Control Z (undo).
- Enhancement -- Border character added to Information Block. This character will be blinked (rather than a Blank) whenever the field is filled and the cursor is forced outside.
- Enhancement -- Upon initial entry in immediate mode, the application will get control before the cursor starts blinking. This will allow initial cursor repositioning without cursor remnants.
- Enhancement -- Last event type parameter added to Information Block. This parameter is not currently used.

EXTERNAL REFERENCE SPECIFICATION

APPLE // CONSOLE DRIVER

Neal Johnson

April 10, 1985

BETA RELEASE VERSION

TABLE OF CONTENTS

1. Introduction

The Console Driver (henceforth known as "the driver") is an implementation of the Apple /// Console Driver, with special modifications, for the Apple // series of computers (//+, //e, and //c). The driver supplies a simple and consistent interface to a nearly complete set of display format and control procedures contained in a relatively small and fast package. Both display and control commands are sent to the driver in the same manner. This allows a programmer to build up a set of data structures that contain both display and control information. Presentation of the information to the driver can be made with one call. This simplifies the programming of the human interface for a program, in that the programmer does not have to make a sequence of calls to set up for text to be displayed. Instead, the format information can be imbedded in the text itself.

The driver supports a form of "window" known as a "viewport". The viewport is a rectangular portion of the screen where all console functions take place. This feature allows the programmer to define a portion of the screen where s/he wants text to be displayed. All text outside the viewport is protected. Any display of the text will occur within the bounds of the viewport.

The console driver can serve as a low level tool for the implementation of different styles of human interface. Much of the implementation for the various styles of human interface would be in the design of the data structures describing the format and text to be displayed.

NOTE: This release (1.0) of the Console Driver only supports an 80-column screen. Sections describing the 40-column screen should be ignored at this time.

2. Functional Description

2.1 Screen Map

2.1.1 40-Column Screen

The 40-Column screen consists of 40 columns of text in 24 lines. The upper left corner is column 0, line 0 (or simply 0,0.) Columns are number left to right, 0 to 39. Lines are numbered top to bottom, 0 to 23.

2.2.2 80-Column Screen

The 80-Column screen consists of 80 columns of text in 24 lines. The upper left corner is column 0, line 0 (or simply 0,0.) Columns are number left to right, 0 to 79. Lines are numbered top to bottom, 0 to 23.

2.2.3 The Viewport

The Viewport is a rectangular portion of the screen

where all current text is displayed. Portions of the screen outside the viewport are not affected by either format or display commands.

The driver maintains a "cursor", which is not visible on the screen, that represents the current location that a displayable character will be placed. This cursor is specified by the value of the two variables CH and CV. (See Section 2.2.1 below)

When the console driver is first used, the viewport defaults to the whole screen (either 40 or 80 column display). The programmer can set the viewport by a special control and four parameter bytes which specify the upper left and the lower right corners of the viewport.From that point on, all console functions will take place within the bounds of the viewport.

The current viewport specifications can be saved and the viewport can then be set to the specifications of the previously saved viewport. The programmer can then return to the original viewport settings with another command.

2.2 Console Driver Environment Controls

2.2.1 Cursor Position

The current cursor position is maintained in two variables:

CH - current horizontal position

CV - current vertical postion

When the driver is first used, these values are set to zero signifying the upper-left corner of the screen.

The values of CH and CV always represent the absolute screen coordinates (actual column and line number) and are not relative to the current viewport.

2.2.2 Viewport Specification

The viewport is specified by six variables that specify the top, bottom, left, and right edge of the viewport and also its width (in columns) and its length (in lines).

WNDTOP - top line of viewport

WNDBOT - bottom line of viewport

WNDLFT - left column of viewport

WNDRGT - right column of viewport

WNDWTH - width of viewport in columns

WNDLEN - length of viewport in lines

2.2.3 Cursor Movement Controls

The cursor movement controls specify the rules for moving the cursor within a viewport. These controls are flags directing the driver how to move the cursor. If set to zero, they are false and if set to one, they are true. The four cursor movement controls are:

CONLFD (Line Feed) - If true, the console driver will automatically perform a line feed after every carriage return (control code 13 decimal or $0D hex.) When false, no automatic line feed is performed. The programmer can perform a line feed by explicitly sending a line feed character (10 decimal or $0A hex.) Scrolling is controlled by the other cursor movement control settings.

CONADV (Advance) - If true, the cursor will advance one space to the right after each display character is placed on the screen. When false, the cursor will not advance (it will remain in the same position) after each character. In this case the programmer would have to explicitly move the cursor by sending a Move Cursor Right control (09 decimal or $09 hex.) Wrapping and/or scrolling is controlled by the other cursor movement control settings.

CONWRAP (Wrap) - If true, an attempt to move the cursor beyond the right or left edge of the viewport will cause the cursor to be placed at the opposite edge of the next or previous line, respectively, of the viewport. If false, the cursor remains at the edge of the viewport on the current line. To move to either the next or previous line requires the programmer to send a Move Cursor Up (11 decimal or $0B hex) or a Move Cursor Down (10 decimal or $0A hex) character, followed by either a Return Cursor (13 decimal or $0D hex) to move the cursor to the beginning of the previous line or a Horizontal Position (24 decimal or $18 hex) with the appropriate parameter value to send the cursor to the end of line. Scrolling is controlled by the other cursor movement control.

CONSCRL (Scroll) - If true, an attempt to move the cursor beyond the top or bottom line of the viewport, will cause the contents of the viewport to be scrolled either down or up. The cursor will then be placed at the beginning of the new top or bottom line. If

false, the cursor will remain at the top or bottom of the viewport.

DLEFLAG (Space Expansion) - If true, the DLE's ($10 hex or 16 decimal) will be interpreted as space expansion controls with a following parameter byte. (See section 2.3.17) If false, then they are ignored.

2.2.4 Fill Character

The fill character is the character used to clear the contents of the viewport. This value is a Space (32 decimal or $20 hex). Its value is in the variable CONFILL. Due to the Apple // character mapping the actual binary value of the fill character is $0A0 hex or 160 decimal for a normal Space character or $20 hex or 32 decimal for an inverse Space character.

2.2.5 Default Settings and Environment

2.2.5.1 The Default Viewport

The default viewport is the entire screen (either 40 or 80 columns).

Viewport Parameter	40-Col Value	80-Col Value
WNDTOP	0	0
WNDBOT	23	23
WNDLFT	0	0
WNDRGT	39	79
WNDWTH	40	80
WNDLEN	24	24

2.2.5.2 The Default Cursor Movement Controls

The default settings for the Cursor Movement Controls are:

CONLFD (Line Feed) - TRUE

CONADV (Advance) - TRUE

CONWRAP (Wrap) - TRUE

CONSCRL (Scroll) - TRUE

DLEFLAG (Space Expansion) - TRUE

2.2.5.3 The Default Screen Environment

The default screen environment is the default viewport (See section 2.2.5.1), the text display mode is normal, the cursor is off, the fill character is space, and the initial position of the cursor is in the upper-left hand corner (0,0).

2.2.5.4 Mousetext

The flag MOUSE, specifies whether or not the driver will display mousetext characters. If MOUSE is true then character is the range $40 to 5F hex or 64 to 95 decimal will be mapped into the mousetext character set. If false, the mapping will not take place. Control codes will always be processed as is. The default is MOUSE false.

2.2.5.5 Normal and Inverse Text

The flag CONVID controls the display of text in either normal or inverse modes. If CONVID is $80 hex or 128 decimal, text is displayed in normal mode. If CONVID is 0, then text is displayed in inverse. The setting of CONVID is handled via two control codes described below (Set Normal Text or Set Inverse Text.)

2.3 Screen Control Codes

2.3.1 No Operation

CONTROL CODE: $00 (hex) or 00 (decimal)

OPERATION: No Operation

DESCRIPTION: This control code has no effect and is ignored.

2.3.2 Save and Reset Viewport

CONTROL CODE: $01 (hex) or 01 (decimal)

OPERATION: Save and Reset Viewport

DESCRIPTION: This control code saves the current settings of the viewport: its coordinates, cursor position, cursor motion controls, mousetext, and normal/inverse setting. The viewport will then be set to the default values of the full

screen. (See section 2.3.5 Restore Viewport)

2.3.3 Set Viewport

CONTROL CODE: $02 (hex) or 02 (decimal)

OPERATION: Set Viewport

DESCRIPTION: This control code will set the viewport. It requires four parameter bytes which specify the absolute coordinates for the upper-left and lower-right corners of the viewport. The order of the parameters is:

upper-left corner X (or column) value

upper-left corner Y (or line) value

lower-right corner X (or column) value

lower-right corner Y (or line) value

If less than four parameters are passed, this control code will be ignored. This control simply sets the boundaries for the viewport. It does not affect the cursor motion controls, normal/inverse, or mousetext setting. It will not save the current viewport. The cursor will be placed in the upper-left corner of the new viewport.

The parameters are checked for validity prior to setting the viewport values. The rules for validity are as follows:

If any paramter byte is > 127, i.e. minus value because bit 7 is set, this command will be ignored.

For any X coordinate (UL corner or LR corner), if it is > 39 or 79 (depending on the screen size) then it will be set to 39 or 79.

For any Y coordinate (UL corner or LR corner), if it is > 23 then it will be set to 23.

UL corner X will be used for WNDLFT.

UL corner Y will be used for WNDTOP.

LR corner X, if greater than WNDLFT, will be used for WNDRGT, else this command will be ignored.

LR corner Y, if greater than WNDTOP, will be used for WNDBOT, else this command will be ignored.

If for any reason the command is ignored, it will not change the current viewport settings.

2.3.4 Clear from Beginning of Line

CONTROL CODE: $03 (hex) or 03 (decimal)

OPERATION: Clear from Beginning of Line

DESCRIPTION: This control code will clear the current line from the beginning of the line to and including the current cursor position in that line.

2.3.5 Restore Viewport

CONTROL CODE: $04 (hex) or 04 (decimal)

OPERATION: Restore Viewport

DESCRIPTION: This control code will restore the viewport to the values of the last previously saved viewport. If no viewport has been saved, then the values will be set to the default values for the whole screen. (See section 2.3.2 Save and Reset Viewport)

2.3.6 Undefined

CONTROL CODE: $05 (hex) or 05 (decimal)

OPERATION: Undefined

DESCRIPTION: This control code is undefined and is ignored.

2.3.7 Undefined

CONTROL CODE: $06 (hex) or 06 (decimal)

OPERATION: Undefined

DESCRIPTION: This control code is undefined and is ignored.

2.3.8 Sound the Bell

CONTROL CODE: $07 (hex) or 07 (decimal)

OPERATION: Sound the Bell

DESCRIPTION: This control code will cause the ProDOS recommended "beep" to be sounded. It has no effect on the screen. Sequencial control codes will have the effect of producing a longer sound.

2.3.9 Move Cursor Left

CONTROL CODE: $08 (hex) or 08 (decimal)

OPERATION: Move Cursor Left

DESCRIPTION: This control code will move the cursor left one position. Wrapping around and scrolling are performed in accordance with the settings of the cursor motion controls. (See sections 2.2.3 and 2.3.22 Cursor Movement Controls)

2.3.10 2.3.11 Move Cursor Down

CONTROL CODE: $0A (hex) or 10 (decimal)

OPERATION: Move Cursor Down (Line Feed)

DESCRIPTION: This control code moves the cursor down one line. Scrolling is performed in accordance with the cursor motion controls. (See sections 2.2.3 and 2.3.22 Cursor Movement Controls)

2.3.12 Clear to End of Viewport

CONTROL CODE: $0B (hex) or 11 (decimal)

OPERATION: Clear to End of Viewport

DESCRIPTION: This control code will clear the contents of the viewport, starting from and including the current cursor position to the end of the line and all the lines below the cursor. The cursor is not moved.

2.3.13 Clear Viewport

CONTROL CODE: $0C (hex) or 12 (decimal)

OPERATION: Clear Viewport

DESCRIPTION: This control character will move the cursor to the upper-left corner of the viewport and then clear the viewport by setting the contents to space characters. The space characters will be either normal or inverse depending on the setting of this mode. (See sections 2.3.15 and 2.3.16)

2.3.14 Return Cursor

CONTROL CODE: $0D (hex) or 13 (decimal)

OPERATION: Return Cursor (Carriage Return)

DESCRIPTION: This control code moves the cursor to the

beginning of the current line (the left edge of the viewport.) A line feed may also be issued automatically after the return depending on the setting of the cursor motion controls. Scrolling may also take place. (See sections 2.2.3 and 2.3.22 Cursor Movement Controls)

2.3.15 Set Normal Text

CONTROL CODE: $0E (hex) or 14 (decimal)

OPERATION: Set Normal Text

DESCRIPTION: This control code specifies that all subsequent characters will be displayed as white characters on a black background. It does not affect any characters already on the screen. This control code will set the flag CONVID to $80 hex or 128 decimal. (See section 2.3.16 Set Inverse Text)

2.3.16 Set Inverse Text

CONTROL CODE: $0F (hex) or 15 (decimal)

OPERATION: Set Inverse Text

DESCRIPTION: This control code specifies that all subsequent characters will be displayed as black characters on a white background. It does not affect any characters already on the screen. This control code will set the flag CONVID to 0. (See section 2.3.15 Set Normal Text)

2.3.17 Space Expansion

CONTROL CODE: $10 (hex) or 16 (decimal)

OPERATION: Space Expansion

DESCRIPTION: This control code supports the DLE space expansion that exists in Pascal text files. It takes one parameter which represents the number of spaces to output plus 32. The driver subtracts 32 from the parameter to determine the number of spaces to output to the screen. If the parameter does not exist, then the driver will ignore this control. DLE expansion can be turned off using the mode value of 4 or 12 in the UNITWRITE call to the driver. (See section 3.1.2 below.) It can also be turned on or off with the Cursor Movement Control. (See Section 2.3.22 below) The default is on.

2.3.18 Horizontal Shift

CONTROL CODE: $11 (hex) or 17 (decimal)

OPERATION: Horizontal Shift

DESCRIPTION: This control code will cause the contents of the viewport to be shifted right or left the number of columns specified by the single byte parameter following the control code. If the parameter does not exist or is set to 0, the control will have no effect. The parameter is interpreted as an eight-bit two's complement number. If it is positive (less than 128 decimal or $7F hex) the contents will be shifted right the number of columns equal to the value of the number. If it is negative (greater than or equal to 128 decimal or $7F hex), the contents will be shifted left the number of columns equal to the negative value of the number. In both cases, if the value is greater than or equal to the width of the viewport, it will cause the viewport to be cleared.

The shifted characters are moved directly to their destination location. The space vacated by the shifted characters is set to blanks. Characters shifted out of the viewport are removed from the screen and are not recoverable.

2.3.19 Vertical Position

CONTROL CODE: $12 (hex) or 18 (decimal)

OPERATION: Vertical Position

DESCRIPTION: This control code will move the cursor vertically to the relative line number passed in a single byte parameter (0 to 23 for both 40-columns or 80-columns). A parameter whose value is 10 means to move to the tenth line in the viewport, not to line 10 of the whole screen. A parameter of 0 will move the cursor to the topmost line. To determine the correct relative line, the parameter is added to the value of WNDTOP (See Section 2.2.2 Viewport Specifications). This is an eight-bit add. If the resulting value is greater than the value of WNDBOT (the bottommost line of the viewport) but less than 127 then the cursor will be placed in the bottommost line of the viewport. If the sum is greater than 127 (negative) then the cursor will be placed in the topmost line. If the parameter is missing, this control will be ignored. This control has no effect on the horizontal position of the cursor.

2.3.20 Clear from Beginning of Viewport

CONTROL CODE: $13 (hex) or 19 (decimal)

OPERATION:Clear from Beginning of Viewport

DESCRIPTION: This control code will clear the viewport from its beginning (0, 0 or home position) to and

including the cursor. The cursor is not moved.

2.3.21 Horizontal Position

CONTROL CODE: $14 (hex) or 20 (decimal)

OPERATION: Horizontal Position

DESCRIPTION: This control code will move the cursor horizontally to the relative column number passed in a single byte parameter (0 to 39 for 40-columns or 0 to 79 for 80-columns). A parameter whose value is 10 means to move to the tenth column in the viewport, not to column 10 of the whole screen. A parameter of 0 will move the cursor to the left-most column. To determine the correct relative column, the parameter is added to the value of WNDLFT (See Section 2.2.2 Viewport Specifications). This is an eight-bit add. If the resulting value is greater than the value of WNDRGT (the rightmost column of the viewport) but less than 127 then the cursor will be placed in the rightmost column of the viewport. If the sum is greater than 127 (negative) then the cursor will be placed in the leftmost column. If the parameter is missing, this control will be ignored. This control has no effect on the vertical position of the cursor.

2.3.22 Cursor Movement Controls

CONTROL CODE: $15 (hex) or 21 (decimal)

OPERATION: Cursor Movement Controls

DESCRIPTION: This control code and its parameter will set the cursor movement controls as specified by the parameter. The parameter is a single byte value, with only the lower five bits as significant. The upper four bits are to be set to zero. A zero will reset the control and a one will set it. If the parameter does not exist or the upper three bits are non-zero, the command is ignored. (See section 2.2.3 Cursor Movement Controls)

Bit	Control
Bit 0	Advance
Bit 1	Line Feed
Bit 2	Wrap
Bit 3	Scroll
Bit 4	DLE Space Expansion

2.3.23 Scroll Down

CONTROL CODE: $16 (hex) or 22 (decimal)

OPERATION: Scroll Down

DESCRIPTION: This control code will cause the contents of the viewport to scrolled down, leaving a blank line at the top of the viewport. The cursor position will remain the same after the scroll.

2.3.24 Scroll Up

CONTROL CODE: $17 (hex) or 23 (decimal)

OPERATION: Scroll Up

DESCRIPTION: This control code will cause the contents of the viewport to be scrolled up, leaving a blank line at the bottom of the viewport. The cursor position will remain the same after the scroll.

2.3.25 Turn Mousetext Off

CONTROL CODE: $18 (hex) or 24 (decimal)

OPERATION: Turn Mousetext Off

DESCRIPTION: This control code turns off the display of mousetext (See Section 2.3.28).

2.3.26 Home Cursor

CONTROL CODE: $19 (hex) or 25 (decimal)

OPERATION: Home Cursor

DESCRIPTION: This control code moves the cursor to the upper-left corner of the current viewport. It does not clear any portion of the viewport, nor does it change any of the viewport settings.

2.3.27 Clear Line

CONTROL CODE: $1A (hex) or 26 (decimal)

OPERATION: Clear Line

DESCRIPTION: This control code moves the cursor to the beginning of the current line and then clears the entire line.

2.3.28 Turn Mousetext On

CONTROL CODE: $1B (hex) or 27 (decimal)

OPERATION: Turn Mousetext On

DESCRIPTION: This control code turns on the display of mousetext characters. All displayable characters (See Section 2.4 Displayable Characters) in the range $40 - $5F hex or 64 - 95 decimal will be mapped into the mousetext characters for display. (See Section 2.3.25)

2.3.29 Move Cursor Right

CONTROL CODE: $1C (hex) or 28 (decimal)

OPERATION: Move Cursor Right

DESCRIPTION: This control code will move the cursor right one position. Wrapping around and scrolling are performed in accordance with the settings of the cursor motion controls. (See sections 2.2.3 and 2.3.22 Cursor Movement Controls)

2.3.30 Clear to End Of Line

CONTROL CODE: $1D (hex) or 29 (decimal)

OPERATION: Clear to End of Line

DESCRIPTION: This control code clears the current line starting from and including the current cursor position in the line. The cursor is not moved.

2.3.31 Absolute Position

CONTROL CODE: $1E (hex) or 30 (decimal)

OPERATION: Absolute Position

DESCRIPTION: This control code combines the actions of the Horizontal Position and Vertical Position control codes. (See sections 2.3.25 and 2.3.26). It requires two single byte parameters. The first specifies the horizontal position and the second specifies the vertical position of the cursor. Placement of the cursor follows the rules given under both Horizontal and Vertical Position control codes. If both parameter bytes are missing, the command is ignored.

2.3.32 Move Cursor Up

CONTROL CODE: $1F (hex) or 31 (decimal)

OPERATION: Move Cusor Up (Vertical Tab)

DESCRIPTION: This control code moves the cursor up one line. Scrolling is performed in accordance with the cursor motion controls. (See sections 2.2.3 and 2.3.22 Cursor Movement Controls)

2.4 Displayable Characters

The Console Driver uses the Alternate Character set of the Apple // for the display of characters. It assumes however, that all characters passed to it are in the standard ASCII character set (range $00 to $7F hex or 0 to 127 decimal). These characters will be mapped into the appropriate character set for display purposes, e.g. normal or inverse or mousetext.

A special case is made for characters passed to the driver in the range $80 to $FF hex or 128 to 255 decimal. The characters are displayed after reseting the 7th bit. This results in the mapping shown in the chart below:

$80 - $9F	mapped to	Inverse upper case letters
$A0 - $BF	mapped to	Inverse special characters
$C0 - $DF	mapped to	Mousetext characters
$E0 - $FF	mapped to	Inverse lower case letters

This is independent of the settings for normal/inverse and mousetext in the driver. Refer to the Apple // Reference Manuals for more details on the character sets.

All characters in the range $00 to $1F hex or 0 to 31 decimal are defined as control codes which invoke the operations listed above in Section 2.3.

All characters in the range $20 to $7F hex or 32 to 127 decimal are defined as displayable characters and will be displayed given the various settings of the console driver on the screen.

The use of mousetext requires that the mousetext-on control code be sent to the console driver. Then any characters in the range $40 to $5F hex or 64 to 95 decimal will be mapped into the appropriate mousetext character. For example, to get the "running man" characters would require:

27 - mousetext-on control code
"F" - first part of "running man"
"G" - second part of "running man"

At the end of a sequence of mousetext characters, it is important to turn off mousetext with the mousetext-off control code. Any' characters not in the mousetext range will be displayed as is

given the settings of the console driver.

3. Interface Description

3.1 Pascal

3.1.1 Data Interface

Both control codes and text to be displayed are passed to the driver as a contiguous array of data. For example, if the programmer wished to print "Hello" on line 10, column 15, in inverse, and then home the cursor and return back to normal text, s/he would create the following array of data (all numbers are decimal):

30	- absolute position
15	- parameter (column 15)
10	- parameter (line 10)
15	- inverse text
72	- "H"
101	- "e"
108	- "l"
108	- "l"
111	- "o"
25	- home cursor
14	- normal text

This array is not a string in the Pascal sense of the word, in that the first byte is data and not the length of the array (as in a string.) The console driver can accept an array up to 32767 bytes long (Pascal limit on integers).

The second required bit of data is an integer that denotes the length of the array to be processed by the driver. In the above example, the integer could either be a variable with the value 11 or the constant "11".

3.1.2 Calling the Console Driver

The driver is an "Attach" driver for Pascal. For information on Pascal Attach drivers, please refer to APPLE // PASCAL 1.2 DEVICE AND INTERRUPT SUPPORT TOOLS MANUAL. The unit number for the driver is #130.

To transfer data to the driver to be displayed on the screen, requires a UNITWRITE call from a Pascal program. The format for the call is shown below:

UNITWRITE(130, ARRAY_ADDR, LENGTH_ARRAY, MODE)

where 130 is the unit number for the driver

ARRAY_ADDR is a VAR parameter denoting the address of the array of data

LENGTH_ARRAY is the length of the array passed

MODE is the mode expression which is an integer. This can have four values:

value	DLE-expansion	Auto linefeed
0	TRUE	TRUE
2	FALSE	TRUE
8	TRUE	FALSE
12	FALSE	FALSE

When passing a string to the driver, it is important to always reference the string as:

STRING_VAR[1]

so as not to pass the length byte found in STRING_VAR[0].

3.1.3 Status Calls

The driver only accepts one status call that returns a data structure that describes the current state of the driver. (See section 2 for a description of these variables.) The form of the UNITSTATUS call is shown below:

UNITSTATUS(130, CON_STAT_BLK, 0)

where 130 is the unit number of the driver

CON_STAT_BLK is a record with the format:

```
TYPE BYTE = 0..255

VAR CON_STAT_BLK: PACKED RECORD OF
        CV:BYTE;
        CH:BYTE;
        WNDTOP:BYTE;
        WNDBOT:BYTE;
        WNDLFT:BYTE;
        WNDRGT:BYTE;
        WNDWTH:BYTE;
        WNDLEN:BYTE;
        CONWRAP:BYTE;
        CONADV:BYTE;
        CONLFD:BYTE;
        CONSCRL:BYTE;
        CONVID:BYTE;
        DLEFLAG:BYTE;
        CONFILL:BYTE;
        MOUSE:BYTE;
    END;
```

This call will instruct the driver to copy its values into this record so the programmer may inspect the current state of the driver.

3.1.4 Control Calls

The driver accepts four control calls. These calls allow the programmer to get the current location of the cursor, the text character at the current cursor location, or save and restore either the contents of the current viewport. The buffer in which this data is stored must be supplied by the programmer, it is not in the driver itself. For programs that do not require this function, this saves them space. It is recommended that the programmer allocate some space on the heap for this storage. This allows this space to be reclaimed as needed. To calculate the amount of space required for a viewport, multiply its width (WNDWTH) by its length (WNDLEN).

3.1.4.1 Getting the Current Cursor Position

To get the current location of the cursor on the text screen, the programmer can make a UNITSTATUS call of the form:

```
UNITSTATUS(130, LOCATION, 2);
```

where LOCATION is a record of the form:

```
LOCATION = RECORD
             HORIZONTAL: INTEGER;
             VERTICAL: INTEGER;
           END;
```

The driver will set these values equal to the screen coordinates, CH and CV. These are integer values. These values are not relative to the viewport but represent the actual column and line number.

3.1.4.2 Getting the Current Text Screen Character

By making a UNITSTATUS call of the form:

```
UNITSTATUS(130, CHARACTER, 8194);
```

where CHARACTER is a byte (0..255) variable,

the driver will return the current binary value of the character found at the current cursor location.

3.1.4.3 Saving and Restoring the Viewport

To save the contents of the viewport, requires

a UNITSTATUS call of the form:

UNITSTATUS(130, VWPORT_BUF, 16386);

where 130 is the unit number for the driver

VWPORT_BUF is a buffer to hold the contents of the viewport.

To restore the contents of the viewport, requires a UNITSTATUS call of the form:

UNITSTATUS(130, SCREEN_BUF, 24578);

where 130 is the unit number for the driver

VWPORT_BUF is a buffer to hold the contents of the screen.

It is up to the programmer to keep track of which viewport has been saved in which buffer. When restoring a viewport, the programmer must have already set the required viewport prior to the restore call.

3.2 BASIC

The version of the console driver that is used with BASIC programs supports the following functions:

Output Data to the Console

Save the Current Viewport

Restore the Current Viewport

Get the Status of the Console Driver

Get the Current Cursor Position

Get the Current Text Screen Character

Initialize the Console Driver

Get A Segment of Memory

Get a Console Driver Error

Get the Console Driver Version

Get the Console Driver Copyright Notice

Release the Console Driver

The console driver functions are AMPERSAND ('&') routines.

3.2.1 Console Driver Functions

3.2.1.1 Calling the Console Driver

Calls the the Console Driver are done using the "Ampersand Hook". BASIC statements of the form:

&name(parameter list)

are used to call the Console Driver. Specific formats for the calls are described below.

3.2.1.2 Output Data to the Console

There are two calls to the driver to output data to the display. The first is of the form:

&WRTSTR(S$)

where S$ is a string

This call will output the contents of S$ to the display. S$ can include both control codes and ASCII characters.

The second form is:

&WRITE(I1%, I2%, SA%)

where SA$ is a one-dimensional string array and I1% is a starting index and I2% is an ending index

This call will output a sequence of strings contained in the string array SA$. The sequence begins with the string selected by the index I1% and will end with the string indexed by I2%. These strings can contain both control codes and ASCII characters.

3.2.1.3 Save the Current Viewport Contents

In order to save the contents of the viewport, a buffer must be allocated to store the contents. This is done through a call to the special function "Get memory" whose form is:

&GTMEM(P%, A%)

where P% is an integer specifies the number of pages (256 bytes) of memory to allocate and A% will be the address of that memory

This call allocates the number of pages required to store the viewport contents. The number of pages required can be calculated by

(WNDWTH * WNDLEN) / 256

rounding up to the nearest integer

For example to store the whole screen contents requires 8 pages to be allocated. The memory address of the memory allocated is returned in the variable A%. If the required number of pages is not available, then a BASIC "OUT OF MEMORY" error will occur.

Once a call to &GTMEM has been made, then a call to save the contents of the viewport can be made. The call is of the form:

&SVVP(A%)

where A% is the address returned from a call to &GTMEM

3.2.1.4 Restore the Current Viewport Contents

To restore the viewport contents, a call of the form:

&RSTRVP(A%)

where A% is the address used in the call to &SVVP

This will restore the previously saved contents to the viewport. The programmer must be careful to restore contents that are of the same size as the current viewport.

3.2.1.5 Get the Status of the Console Driver

To get the status of the console driver, a call of the form:

&CDINFO(CI%)

where CI% is a 16 element array, i.e.

DIM CI%(16)

This will return the contents of the status block to the array CI%. To inspect the contents, the following is a mapping of the array elements to

the status block elements:

CI%(1)	=	CV
CI%(2)	=	CH
CI%(3)	=	WNDTOP
CI%(4)	=	WNDBOT
CI%(5)	=	WNDLFT
CI%(6)	=	WNDRGT
CI%(7)	=	WNDWTH
CI%(8)	=	WNDLEN
CI%(9)	=	CONWRAP
CI%(10)	=	CONADV
CI%(11)	=	CONLFD
CI%(12)	=	CONSCRL
CI%(13)	=	CONVID
CI%(14)	=	DLEFLAG
CI%(15)	=	CONFILL
CI%(16)	=	MOUSE

3.2.1.6 Get the Current Cursor Position

To get the current position of the cursor, a call of the form:

&GTCP(H%, V%)

where H% is the value of CH (x-position) and V% is the value of CV (y-position)

This call returns the absolute coordinates of the cursor.

3.2.1.7 Get the Current Text Screen Character

To get the value of the text character at the current cursor position, a call of the form:

&GTCHR(C%)

where C% is the character returned

This call returns the binary value of the text character at the current cursor position.

3.2.1.8 Initialize the Console Driver

To initialize the Console Driver to its default environment, a call of the form:

&INITCD

This call sets the driver environment to its default state described above.

3.2.1.9 Release Console Driver

To release the Console Driver Ampersand package and to restore the screen to a normal BASIC environment, a call of the form:

&STPCD(C%)

where C% is 40 to set up a normal 40-column display or 80 to set up a normal 80-column display

3.2.1.10 Console Driver Version and Copyright

To access the version number of the driver, a call of the form:

&CDVRSN(V%, R%)

where V% is the version number returned and R% is the revision number returned

To access the copyright notice of the driver, a call of the form:

&CDCPYRT(CM%)

where CM% is the copyright notice returned

3.2.1.11 Setting the Console Driver Address

Before the Ampersand package can use the Console Driver, it must have the location of the driver passed to it with the call:

&STCDADR(A%)

where A% is the starting address (which is also of the entry-point) of the console driver

This call must be made before any other calls to the Ampersand package.

3.2.2 Using the Console Driver with Your Program

A BASIC program using the console driver should do no console display through BASIC. All display should be done with the driver.

A sample use of the driver to place the string "Hello there" at position 10, 15 would be:

```
10 DIM ABS$(3)
```

```
20 DIM STR$(11)
30 ABS$(1) = CHR$(30): REM ABSOLUTE POSITION
40 ABS$(2) = CHR$(10): REM X COORDINATE
50 ABS$(3) = CHR$(15): REM Y COORDINATE
60 STR$ = "Hello there"
70 &WRTSTR(ABS$)
80 &WRTSTR(STR$)
```

3.2.3 Locating the Console Driver in Memory

The Console Driver is an EDASM produced REL file. This requires that it be relocated in memory before it can be used. Following the instructions in either the ProDOS or DOS Assembler Tools Manual, use RBOOT and RLOAD to perform the relocation.

3.3 Assembler

The version of the console driver that is used with assembly language programs supports the following functions:

Output Data to the Console

Save the Current Viewport

Restore the Current Viewport

Get the Status of the Console Driver

Get the Current Cursor Position

Get the Current Text Screen Character

Initialize the Console Driver

The console driver has a single entry point. Calling the driver is done in much the same way as ProDOS MLI calls.

3.3.1 Console Driver Functions

3.3.1.1 Calling the Console Driver

Calls to the console driver are done in much the same way as calls to the ProDOS MLI. The driver has only one entry point located at the beginning. Once the driver has been relocated in memory, its starting address is the entry point of the driver. A call is made as shown below:

```
JSR     PCONSOLE
DFB     COMMAND
DW      PARAMPTR
BNE     ERROR_HANDLER
```

The label PCONSOLE is the starting address of the driver. The programmer will determine this through deciding where to relocate the driver in memory. In the calling program there should be a ststement of the form:

```
PCONSOLE        EQU     nnnn
```

where nnnn is the starting address of the driver.

The JSR is followed by a byte that holds the command value which is a number that selects the appropriate console driver function. For specific values, see below.

Following the command value byte is a two byte pointer to a parameter list. The format for the parameter list verifies per console driver function. The specific formats are described below.

The driver will return to the caller with the carry flag clear if no error occured, or with the carry flag set if an error did occur. The calling program should check the carry flag (the BNE instruction shown above) and report an appropriate error. The actual error type is passed back to the caller in the A-register. The error handler can check this value to determine the specific error that occured.

3.3.1.2 Output Data to the Console

CALLING FORMAT:

```
        JSR     PCONSOLE
        DFB     0               ;output to screen
        DW      OUTPUTDATA
```

PARAMETER LIST FORMAT:

```
OUTPUTDATA      DW      DATA1
                DW      LENGTH1
```

This call will output data (both text and control codes) to the console driver. The parameter list is a pointer to a data string followed by a length value. For example, DATA1 would point to

```
DATA1   DFB     30      ;absolute position
        DFB     10      ;x position
        DFB     15      ;y position
        ASC     "Hello there!!"
```

```
LENGTH1 EQU     16      ;length of DATA1
```

This call returns no errors. The A-register value will be 0 and the carry flag will be clear.

3.3.1.3 Save the Current Viewport Contents

CALLING FORMAT:

```
        JSR     PCONSOLE
        DFB     1               ;save viewport
        DW      SAVEBUFFER
```

PARAMETER LIST FORMAT:

```
BUFFERSIZE      EQU     1920    ;full screen

SAVEBUFFER      DS      BUFFERSIZE
```

This call will save the contents of the current viewport in the buffer pointed to in the call, in this case SAVEBUFFER. This buffer must be large enough to hold the entire contents of the viewport. The number of bytes required is equal to the width of the viewport (WNDWTH) times the length (WNDLEN). In the example shown above, the buffer is large enough to hold the contents of the entire screen (80 columns by 24 lines).

This call returns no errors. The A-register value will be 0 and the carry flag will be clear.

3.3.1.4 Restore the Current Viewport Contents

CALLING FORMAT:

```
        JSR     PCONSOLE
        DFB     2               ;restore viewport
        DW      SAVEBUFFER
```

PARAMETER LIST FORMAT:

```
SAVEBUFFER      DS      BUFFERSIZE
```

This call will restore the contents of the current viewport from the buffer pointed to in the call, in this case SAVEBUFFER. The programmer should be careful that the viewport contents to be restored matches the size of the current viewport. A viewport can be defined, its contents saved, and then the viewport can be redefined as the same size but at a different location on the screen. Then the contents can be restored back to it. This gives the programmer

the ability to move a viewport and its contents around the screen.

This call returns no errors, the A-register is 0 and the carry flag is cleared.

3.3.1.5 Get the Status of the Console Driver

CALLING FORMAT:

```
        JSR     PCONSOLE
        DFB     3               ;get status
        DW      STATUSBLK
```

PARAMETER LIST FORMAT:

```
STATUSBLK       EQU     *

CV      DFB     0
CH      DFB     0
WNDTOP  DFB     0
WNDBOT  DFB     0
WNDLFT  DFB     0
WNDRGT  DFB     0
WNDWTH  DFB     0
WNDLEN  DFB     0
CONWRAP DFB     0
CONADV  DFB     0
CONLFD  DFB     0
CONSCRL DFB     0
CONVID  DFB     0
DLEFLAG DFB     0
CONFILL DFB     0
MOUSE   DFB     0
```

This call will return the current status of the console driver in the status block pointed to in the call, in this case STATUSBLK. The programmer must insure that the status block used matches this description exactly or data may be destroyed if the status block is smaller than the one described.

This call returns no errors, the A-register will be 0 and the carry flag will be clear.

3.3.1.6 Get the Current Cursor Position

CALLING FORMAT:

```
        JSR     PCONSOLE
        DFB     4               ;get cursor position
        DW      CURSORPOS
```

PARAMETER LIST FORMAT:

```
CURSORPOS       EQU     *

XPOS    DFB     0
YPOS    DFB     0
```

This call will return the absolute screen coordinates of the current cursor position. XPOS is the column and YPOS is the line. These values correspond the values of CH and CV described above.

This call returns no errors, the A-register will be 0 and the carry flag will be clear.

3.3.1.7 Get the Current Text Screen Character

CALLING FORMAT:

```
        JSR     PCONSOLE
        DFB     5               ;get text character
        DW      TEXTCHAR
```

PARAMETER LIST FORMAT:

```
TEXTCHAR        DFB     0
```

This call will return the binary value of the text character located at the current cursor position. This value will reflect whether or not the character is inverse, normal, or mousetext. It is up to the calling program to decipher the value.

This call returns no errors, the A-register will be 0 and the carry flag will be clear.

3.3.1.8 Initialize the Console Driver

CALLING FORMAT:

```
        JSR     PCONSOLE
        DFB     6               ;initialize
        DW      0
```

PARAMETER LIST FORMAT:

No parameter list required.

This call will set the console driver back to its default state. No parameter list is required.

This call returns no errors. The A-register

will be 0 and the carry flag will be clear.

3.3.2 Using the Console Driver with Your Program

3.3.2.1 Console Driver Zero Page Usage

The console driver uses zero page locations $20 to $40. The contents of these locations are saved when the driver is called and restored upon exit.

3.3.2.2 Console Driver Softswitch Usage

The console driver uses certain softswitches to control its use of the display memory. They are:

80COL ($C00D) - turn on 80-column card

80STORE ($C001) - use auxilary memory for display

PAGE2 ($C055, $C054) - to switch between even and odd locations on the 80-column card

ALTCHARSET ($C00F) - to use alternate character set

When the console driver is called these switches are set to their appropriate value. Since the console driver is intended to be the SOLE means by which console display is managed, these switches are NOT reset when the driver returns to the calling program. It is up to the program to reset back to the normal environment.

3.3.2.3 Relocating the Console Driver

The Console Driver is an EDASM produced REL file. This requires that it be relocated in memory before it can be used. Following the instructions in either the ProDOS or DOS Assembler Tools Manual, use RBOOT and RLOAD to perform the relocation.

"Filecard" Menu Support Unit

EXTERNAL RFFERENCE SPECIFICATION

APPLE // PASCAL "FILECARD" MENU SUPPORT UNIT

Neal Johnson

Novemeber 10, 1984

Final Release Version

TABLE OF CONTENTS

"Filecard" Menu Support Unit

1. Introduction

The Apple // Pascal "Filecard" Menu Support Unit (herein known as the unit) is an implementation of a simple "filecard" style human interface similar to that used in the products Appleworks and Access II. This unit allows a Pascal-based application to use a "filecard" style human interface without the programmer having to implement the details of such an interface.

The unit requires the Apple // Console Driver to be present in order to function (see the APPLE // CONSOLE DRIVER E.R.S.).

The unit is an intrinsic unit that can be either in SYSTEM.LIBRARY or in a program library (Pascal 1.2 128K system).

The unit supplies data structures to define and manage a "filecard" style, hierarchical menu structure as well as the routines necessary to perform the required functions of such a human interface.

This document describes the interface in detail, and explains how to go about designing an application that will use this style of interface. It then descibes the unit used to implement the interface in an application and a sample program that uses that uses the interface.

2. "Filecard" Menu Description

2.1 The Screen Layout

The screen layout for the "filecard" style interface is divided into three portions:

The Top

The "Filecard" Area

The Bottom

Each of these areas serves a primarily different function within the scope of the whole human interface. See Figure 2.1 for a picture of the layout.

There are routines in the unit to manage each of these portions, either as a whole or in parts.

2.2 The Top Portion

The top portion occupies lines 0 - 2 (three lines) on the screen. The top line (0) is divided into three portions known as the Left, the Middle, and the Right. Line 1 is left blank. Line 2 is a line of "_" dividing this portion of the screen from the center portion. See Figure 2.1 for an illustration.

The top portion is used in conjunction with the "filecard" area to display information about where the user is in the hierarchy of

menus.

The Left hand side is primarily used for application specific information, such as the name of the application, the current disk drive selected, the file name of a selected file, etc. The contents for this side are up to the application.

The Middle section is used to display the title of the currently selected filecard. This section changes as other filecards are selected. The unit manages this area for you during the menu selection process (see below).

The Right hand side is used to display the "escape route" for the user. During the menu selection process, if a selection calls up a filecard (lower in the hierarchy), this section will display the name of the previous filecard. See Figures 2.4.1 to 2.4.4. When a filecard is displayed, typing an ESCAPE will revert back to the previous (or higher level) filecard. This section is also managed for you during the selection process.

2.3 The Bottom Portion

The Bottom portion occupies lines 21 - 23 (three lines). Line 21 is a line of "_" to divide this portion from the center section. Line 22 is used to display text. For the sample program described below, line 23 is left blank. See Figure 2.1 for illustration.

The Bottom portion is divided into two sections, the Right and the Left. This portion of the screen is used primarily to give the user instructions, such as what things to type during the selection process, or as an area for input. The Right section is used for these types of activities. The Left section is used for application specific information; its content being left up to the programmer. The unit supplies routines to manage this portion of the screen.

2.4 The "Filecard" Area

The "Filecard" area of the screen occupies the center portion, from line 3 to line 20 (18 lines). It is used for the display of filecards during the selection process. The unit allows up to four levels of filecards to be displayed at one time (each card overlays the previous card.) See Figures 2.4.1-4 for illustrations of the different levels and the interaction with the top portion of the screen.

The unit supplies the necessary routines to manage this area of the screen.

2.5 Error Boxes

The unit supplies a simple mechanism to put error messages on the screen, overlaying the current screen contents. An error box is placed on lines 12 - 16 and between columns 10 and 71. A viewport is defined within the error box that allows for up to

three lines of text, up to 59 characters long. When an error box is displayed, the console will beep. There are two routines to support error boxes, one to display them and one to remove them from the display restoring the previous screen contents. See Figure 2.5 for an illustration.

3. "Filecards"

3.1 Introduction

The primary "pictorial" framework for a menu in this style of human interface is called a "filecard". This is not to suggest that a menu is like a filecard. The name is a result of the shape of the menu and not its function! A "filecard" is a rectangular box with a tab on the left top edge which holds the name of the menu displayed. The actual menu items are listed within the box.

It is suggested that the name of the "filecard" represent the generic relationship of the menu items. For example, a set of menu items dealing with files could have the name "File Activities".

Each menu item can be selected by the user of the program. The action taken may select another menu which causes another "filecard" to be displayed or it may result in performing some function which does not use the "filecard" interface. A menu item which selects another menu is called a "menu selector". A menu item that selects a function to be performed is called an "action selector". See Figure 3.1 for an illustration.

3.2 How to Design a Hierarchical Menu Structure

When designing an application, one of the most difficult problems is the design of the human interface. Using this unit makes designing the "look" of the interface quite simple. As an application developer, however, you are still faced with designing how you want to split the different activities that can be performed in your application into a series of menus and actions.

One approach that makes this type of designing manageable, is to conceptualize the actions in a "hierarchical" manner. For example, if your application supports a set of 4 major activities:

File Management

Printing

Configuration

Doing the Real Work

these become the menu items on the top most "filecard". Selecting any one of these would then display another "filecard" with items appropriate for that activity.

Selecting "File Management" would then display a "filecard" one level down with the following menu:

Create a File

Delete a File

Catalog a Disk

Rename a File

Selecting any of these items could either display another "filecard" or branch off to do the activity selected.

For every activity supported by your application, you would define a "path" to that activity moving from a general description such as "File Management" to a specific description such as "Create a File." This path moves through a hierarchy of "filecards" and menus. The idea behind this style of interface is that it helps lead the user to the activity they wish to perform. In some cases s/he may not know the actual "name" for the activity, but s/he knows the general type of activity it is. Using this style of human interface allows the application to present the range of activities in such a way that the user can find his/her way to the the desired goal.

As you specify the types of activities and the menus used to select them, it helps to draw a picture of the emerging "menu tree". Figure 3.2.3 shows such a drawing. It is from this drawing that you then design the initialization procedures for the actual "filecards" and menus in your program and the main body of your program where the selection process takes place.

3.3 Card Numbers

Each "filecard" is assigned a number that is used for identification. The number has no other meaning. When the "menu tree" is designed, a number can be assigned. See Figure 3.3 for an example. These numbers are used to refer to individual cards in a program. The numbers assigned are arbitrary, but they must be sequential starting from 0 to the highest numbered card. Card #0 is a special card that exists only as a placeholder for the "escape path" for card #1. For details see below.

3.4 Card Levels

The "menu tree" defines a set of levels where each card resides. More than one card can be at a particular level. The topmost card is level 1. There is only one card at this level. "Filecards" that are displayed as a result of selecting a menu item from the topmost card (level 1) are level 2 cards. "Filecards" displayed as a result of selecting an item at level 2 are level 3 cards, and so on. The unit only supports up to four levels of "filecards", i.e. only four "filecards" can be displayed at one time on the screen. See Figure

2.4.4 for an illustration. Since each "filecard" can have up to 9 menu items this allows for up to 6551 different "filecards" for one application!

3.5 Card Titles

Each "filecard" has a title which is displayed in the tab on the upper left corner of the card. This title can be up to 22 characters in length. The title should reflect the nature of the menu displayed in the card. The title is also displayed in the middle of the top portion of the screen. This represents "where" the user is in the "menu tree". See Figure 3.1 for an example.

3.6 The "Filecard" Data Structure

The unit supplies a data structure to represent each "filecard" used by your application. The format for the data structure is:

```
A_CARD = PACKED RECORD
           MENU_NUMBER: INTEGER;
           MENU_LEVEL: 0..4;
           P_CARD: INTEGER;
           MENU_TITLE: STR22;
         END;
```

The MENU_NUMBER is simply the number you have assigned to the card. The MENU_LEVEL is the level in the "menu tree" of the card. This can have a value between 1 and 4. The value of 0 is a special case. It represents the "top_most" card which is not displayed. It is used to specify the right hand side of the top display to show what happens when the user types ESCAPE at the level 1 card. The integer P_CARD is the card number of the previous card in the "menu tree". This value is used to update the top display Escape path. The MENU_TITLE is a string whose length is limited to 22 characters. This is the name of the "filecard" displayed in the tab on the left hand side.

In your application you should define an array of A_CARD's, one for each "filecard" in your "menu tree". Using your "menu tree" diagram, define a procedure in your program to initialize this data structure. The SAMPLE program provided as an appendix shows such a procedure (set_cards).

4. "Filecard" Menus

4.1 Introduction

Each "filecard" presents the user with a menu of items. The user then selects one of the items to perform. The user uses the UP or DOWN ARROW keys to move through the items, or they can type the item number displayed in the menu to choose an item. Once an item is chosen, they then type RETURN to select that item.

4.2 Menu Description

4.2.1 Menu Items that Select other Menus - "Menu Selectors"

Certain menu items will select another menu to be displayed on another "filecard". This new "filecard" will be one level lower than the "filecard" displaying the item selected. These menu items are called "menu selectors". They do not perform any other action than to specify the next "filecard" to be displayed.

4.2.2 Menu Items that Select Operations - "Action Selectors"

Other menu items select an action to be performed. These items are called "action selectors". In this case, the application branches off to perform some action that has been selected. Here the application may prompt the user for input, display new information for the user, or other such things. In most cases, this implies that the screen display of "filecards" will go away for the duration of that activity. When the activity is done, the application should return to the original "filecard" display shown prior to branching into the activity. Though the unit does not supply the means of performing the activities for your application, it does supply the means of selecting these activities and for coming back to the original "filecard" display. The SAMPLE program described below shows how this is done.

4.3 The Menu Item Data Structure

The unit supplies a data structure to define a single item in the menu. This is the MENU_ITEM data structure, whose format is:

```
MENU_ITEM = PACKED RECORD
              DO_POSITION: BYTE;
              XPOS: BYTE;
              YPOS: BYTE;
              STATE: BYTE;
              DSPLY_TEXT: STR60;
            END;
```

The first three bytes of the record contain control codes used by the Apple // Console Driver to do an Absolute Position. DO_POSITION holds the control code for absolute position, XPOS has the x-position value and YPOS has the y-position value. These are used to position the menu item in the "filecard" for display. The STATE value specifies whether or not the item is displayed in normal text or inverse text. This is used during the selection process. The DSPLY_TEXT is a string of up to 60 characters which is the actual text of the menu item that is displayed. The unit supplies an initialization routine to set up the values of DO_POSITION, XPOS, YPOS, and STATE (INIT_A_MENU).

This record holds all the information necessary to print it

at a given location (XPOS, YPOS) in the current viewport, which is the inside of the current "filecard" displayed on the screen. The text of the item (DSPLY_TEXT) is printed either in normal or inverse depending on the control code in the variable STATE. For details on the control codes used (DO_POSITION and STATE) see the Apple // Console Driver ERS.

4.4 The Menu Data Structure

A set of menu items belonging in one menu are linked togther via another unit-supplied data structure, A_MENU. The format is shown below:

```
A_MENU = PACKED RECORD
           NUM_ITEMS: 1..9;
           CURRENT_ITEM: INTEGER;
           LIST: ARRAY[1..9] OF MENU_ITEM;
         END;
```

NUM_ITEMS specifies the number of menu items in this menu. This can be between 1 and 9. LIST is simply the list of menu items for this menu. The field CURRENT_ITEM is used to maintian the number of the most recently selected item in the menu. This is done so that when a user "escapes" back to a menu, the unit can display the item last selected as highlighted.

In your application you should define an array from 1 to the number of "filecards". For example,

```
MENU: ARRAY[1..9] OF A_MENU;
```

Each element of this array corresponds to one of the "filecards" you have defined. The index into this array is to match the number of the card. Thus MENU[3] specifies the menu to be displayed with the card whose number is 3.

Each element in MENU (MENU[1], MENU[2], ...) requires initialization. In your program you should set up a procedure to set up this array. If you used the INIT_A_MENU procedure, the only elements that require your input are:

```
MENU[n].NUM_ITEMS - gets the number of items for this menu
MENU[n].LIST[nn].DSPLY_TEXT - gets the text for the menu item
```

The SAMPLE program found in the appendix illustrates this procedure (P1_SET_MENU_TEXT and P2_SET_MENU_TEXT.)

5. The "Filecard" Menu Support Unit

5.1 Introduction

The "Filecard" Menu Support Unit supplies the necessary routines to support a simple "filecard" style human interface. In includes data structure definitions to help in setting up

the necessary information about the "menu tree" and procedures to help initialize these data structures.

There are 10 low level routines that allow access to the console driver to support the display of text in conjuction with the unit.

The rest of the routines are designed to help set up the interface, display and remove "filecards" from the display, and get the user's selection from the menu.

Many of the details described below are based on the Apple // Console Driver E.R.S. You should be familiar with its contents before reading this section.

5.2 Unit Interface Data Structures

5.2.1 Console Driver Control Codes

The unit supplies as constants the set of console driver control codes. These can be used by the program to perform other console display activities. The list is:

```
NOOP    =    0
SAVEVP  =    1
SETVP   =    2
CLRBOL  =    3
RESTVP  =    4
BELL    =    7
CURLFT  =    8
CURDWN  =    10
CLREOV  =    11
CLRVP   =    12
CURRET  =    13
NORMAL  =    14
INVERSE =    15
DLE     =    16
HORSHFT =    17
VPOS    =    18
CLRBOV  =    19
HPOS    =    20
CMCONT  =    21
SCRDWN  =    22
SCRUP   =    23
MOFF    =    24
HOME    =    25
CLRLINE =    26
MON     =    27
CURRGT  =    28
CLREOL  =    29
APOS    =    30
CURUP   =    31
```


See the Apple // Console Driver for details on these control codes.

5.2.2 Constant Declarations

The following constants are defined in the unit, though in most cases they are not needed in a program using the unit:

The is the console driver unit number:

```
CONSOLE = 130
```

The following are the required control mode values for UNITSTATUS calls to the console driver:

```
GET_STATUS = 0
GET_CURSOR = 2
SAVE_VP_CONTENTS = 16386
REST_VP_CONTENTS = 24578
```

See the Apple // Console Driver ERS for details on these control mode values.

5.2.3 Type Declarations

5.2.3.1 CON_STAT_BLK

The console driver has a status call which returns the current status of the driver. This record defines this status information:

```
CON_STAT_BLK = PACKED RECORD
                 CV: BYTE;
                 CH: BYTE;
                 WNDTOP: BYTE;
                 WNDBOT: BYTE;
                 WNDLFT: BYTE;
                 WNDRGT: BYTE;
                 WNDWTH: BYTE;
                 WNDLEN: BYTE;
                 CONWRAP: BYTE;
                 CONADV: BYTE;
                 CONLFD: BYTE;
                 CONSCRL: BYTE;
                 CONVID: BYTE;
                 DLEFLAG: BYTE;
                 CONFILL: BYTE;
                 MOUSE: BYTE;
               END;
```

See the Apple // Console Driver E.R.S. for a complete description of these fields.

The unit has a procedure (GET_CON_STATUS) which will perform the status call.

5.2.3.2 POSITION

This record defines the data structure by which the current cursor position can be read via a status call to the console driver:

```
POSITION = RECORD
             XPOS: INTEGER;
             YPOS: INTEGER;
           END;
```

where

XPOS is the absolute x-position of the cursor

YPOS is the absolute y-position of the cursor

5.2.3.3 MENU_ITEM

The record MENU_ITEM defines a single item in one menu:

```
MENU_ITEM = PACKED RECORD
              DO_POSITION: BYTE;
              XPOS: BYTE;
              YPOS: BYTE;
              STATE: BYTE;
              DSPLY_TEXT: STR60;
            END;
```

where

DO_POSITION is the control code for absolute position

XPOS is the x-position

YPOS is the y-position

STATE denotes whether the menu item is normal or inverse text (14 = normal, 15 = inverse)

DSPLY_TEXT is a string up to 60 characters in length that is the text for the menu item

5.2.3.4 A_MENU

This record defines a complete menu for a single "filecard". Its format is:

```
A_MENU = PACKED RECORD
```



```
        NUM_ITEMS: 1..9;
        CURRENT_ITEM: INTEGER;
        LIST: ARRAY[1..9] OF MENU_ITEM;
      END;
```

where

NUM_ITEMS is the number of menu items to be displayed (from 1 to 9)

CURRENT_ITEM is the number of the most recently selected item

LIST is the list of menu items (from 1 to 9) for this particular menu

5.2.3.5 A_CARD

This record defines a "filecard". Its format is:

```
A_CARD = PACKED RECORD
        MENU_NUMBER: INTEGER;
        MENU_LEVEL: 0..4;
        P_CARD: INTEGER;
        MENU_TITLE: STR22;
      END;
```

where

MENU_NUMBER is the number assigned to this "filecard" and its menu

MENU_LEVEL is the level assigned to this card (see above for a description of levels)

P_CARD is the menu_number of the previous card in the "menu tree"

MENU_TITLE is the title for this card, a string no greater than 22 characters in length

5.2.3.6 SCREEN_BUFFER

This type defines a buffer that can store one screen's worth of data. It is used to temporarily store the contents of the screen or a viewport when using a Save Viewport or Restore Viewport control call to the console driver.

```
SCREEN_BUFFER = PACKED ARRAY[1..1920] OF BYTE;
```

5.2.3.7 OUTPUT_BUFFER

This type defines an output buffer where data is placed prior to writing it out to the console driver.

OUTPUT_BUFFER = PACKED ARRAY[0..1023] OF BYTE;

5.2.3.8 ERROR_BUFFER

This type defines a smaller buffer where the screen contents behind an error box are stored so that the screen can be "re-painted" after an error box is removed from the screen.

ERROR_BUFFER = PACKED ARRAY[1..310] OF BYTE;

5.2.3.9 STR22 and STR60

For the strings used in some of the data strucutures defined in the unit, the following special string definitions are used:

STR22 = STRING[22]

STR60 = STRING[60]

5.2.4 Variable Declarations

5.2.4.1 SAVE_BUFFER

This is the buffer that is used by all saves and restores of the viewport . It is large enough to store a full screen (80 columns by 24 lines) of data.

SAVE_BUFFER: SCREEN_BUFFER;

5.2.4.2 ERR_BUFF

This is the buffer used by the errorbox routines to store the information overwritten by the errorbox on the screen so that it can be restored.

ERR_BUFF: ERROR_BUFFER;

5.2.4.3 BUFFER

This is the buffer used to collect data prior to writing it out to the console driver. All the routines in the unit that write to the console use this buffer.

BUFFER: OUTPUT_BUFFER;

5.2.4.4 BUFF_P

This is the global pointer into the output buffer, BUFFER. It is used as an index into the array. When data is placed into the array directly, this pointer must be incremented the appropriate number of times to reflect the number of bytes entered. There is a procedure, RESET_BUFFP, that will set it to 0, to reset the buffer for new input.

BUFF_P: INTEGER;

5.2.4.5 STATUS_BLK

This is a record to hold the status information for the console driver. Its format is described above and in the Apple // Console Driver E.R.S.

STATUS_BLK: CON_STAT_BLK;

5.2.4.6 MODE

All data output to the console driver is done via a UNITWRITE statement. This call requires a "mode expression" to control automatic DLE-expansion and/or automatic linefeeds. Normally, this value is 0 but it can be set for the procedure WRITE_BUFFER (described below) by setting this integer value, MODE. The values are their meanings are:

value	DLE-expansion	Auto linefeed
0	TRUE	TRUE
2	FALSE	TRUE
8	TRUE	FALSE
12	FALSE	FALSE

Any other values will result in undefined states. Changing this value while using the unit's functions can result in poor performance by the unit! If you change it for your own purposes, set it back to zero before calling the unit.

The unit supplies a procedure SET_MODE (5.2.5.9) that will properly set this value. The default setting used by the unit is DLE true and Auto-linefeed true.

5.2.5 Functions Available

5.2.5.1 PUT_CONTROL

CALL FORMAT:

PUT_CONTROL(CONTROL);

where CONTROL is an integer value that represents a control code for the console driver.

This procedure will place a console driver control code in the output buffer, BUFFER, and will increment BUFF_P. For example, to set up an absolute position control sequence, a program would have the following calls:

```
PUT_CONTROL(APOS);
PUT_CONTROL(NEW_X);
PUT_CONTROL(NEW_Y);
```

where NEW_X and NEW_Y are integer values corresponding to the x and y coordinates that the program wishes to move the cursor

5.2.5.2 PUT_STRING

CALL FORMAT:

PUT_STRING(A_STRING);

where A_STRING is a string (0 to 80 characters in length.

This procedure places a string in the output buffer, BUFFER, and increments the pointer BUFF_P the length of the string. For example,

```
PUT_STRING('This is a string to display!');
```

5.2.5.3 RESET_BUFFP

CALL FORMAT:

RESET_BUFFP;

This procedure resets the value of BUFF_P to 0 which effectively clears the output buffer, BUFFER, of data. Before setting up a new buffer-full of data, this procedure should be called.

5.2.5.4 WRITE_BUFFER

CALL FORMAT:

WRITE_BUFFER;

This procedure will write the current contents of the output buffer, BUFFER, to the console driver. The number of bytes written is equal to the current

value of the pointer BUFF_P. After the buffer is written, BUFF_P is set to 0.

5.2.5.5 GET_CON_STATUS

CALL FORMAT:

GET_CON_STATUS;

This procedure will make a status call to the console driver and return the current status information in the data structure, STATUS_BLK, where the calling program can inspect it.

5.2.5.6 VP_SAVE

CALL FORMAT:

VP_SAVE(SCR_BUF);

where SCR_BUF is a byte array large enough to hold the number of characters contained in the current viewport.

This procedure will save off the contents of the current viewport into a buffer. It is critical that the buffer be large enough to hold the number of characters in the viewport. This number can be calculated via a GET_CON_STATUS call and then multiplying the values of WNDLEN and WNDWTH.

5.2.5.7 VP_RESTORE

CALL FORMAT:

VP_RESTORE(SCR_BUF);

where SCR_BUF is a byte array large enough to hold the number of characters contained in the current viewport.

This procedure will restore the contents of the current viewport from the buffer where they were previously saved via a VP_SAVE call. It is critical that the buffer have the same number of bytes of data as the size of the current viewport. It is not important that the viewport occupy the same absolute position on the screen.

5.2.5.8 GET_POSITION

CALL FORMAT:

GET_POSITION(CUR_POS);

where CUR_POS is a record of the type POSITION

This procedure will return the current absolute position of the cursor in the console driver.

5.2.5.9 SET_MODE

CALL FORMAT:

SET_MODE(DLE, LFD);

where DLE and LFD are Boolean values

This procedure will set the MODE variable to the appropriate value given the settings of the DLE and LFD parameters. If DLE is TRUE then DLE-expansion will be set, otherwise it will be reset. If LFD is TRUE than Auto-linefeed will be set, otherwise it will be reset.

5.2.5.10 MIDDLE_UPDATE

CALL FORMAT:

MIDDLE_UPDATE(STR);

where STR is a string

This procedure will update the middle portion of the top display. It is used in conjuction with RIGHT_UPDATE (see below) to update the top portion of the screen during the selection process as "filecards" are displayed. This procedure is used to place the title of the current "filecard" on the screen. This procedure will clear out the right portion of the top, requiring it to be updated also.

This procedure makes a save-viewport control call to the console driver. The calling program should not assume that any viewports it has saved prior to this call will remain "restorable". Before exiting, this procedure will restore the previous viewport setting. The console driver only supports one level of save for viewports. Internal to the unit, this does not matter. Any program using the unit and is also making its own calls to the console driver (Save and Reset Viewport, Restore Viewport) should be aware of this.

5.2.5.11 RIGHT_UPDATE

CALL FORMAT:

RIGHT_UPDATE(STR);

where STR is a string

This procedure will update the right hand portion of the top display. It has no effect on the remaining part of the top display.

This procedure makes a save-viewport control call to the console driver. The calling program should not assume that any viewports it has saved prior to this call will remain "restorable". Before exiting, this procedure will restore the previous viewport setting. The console driver only supports one level of save for viewports. Internal to the unit, this does not matter. Any program using the unit and is also making its own calls to the console driver (Save and Reset Viewport, Restore Viewport) should be aware of this.

5.2.5.12 MAKE_CARD

CALL FORMAT:

MAKE_CARD(CURRENT_CARD, PREVIOUS_CARD);

where CURRENT_CARD and PREVIOUS_CARD are A_CARD's

This procedure will display a "filecard" on the screen. The card displayed will be CURRENT_CARD. The level of this card will determine its placement on the screen. As well as displaying the "filecard" (only the outline and title, the menu is not displayed at this time) this procedure will update middle potion of the top with the title of the current card. Using the PREVIOUS_CARD record it will also update the right hand side of the top with the "escape path".

Upon exiting this procedure, the viewport will be set to the inside of the "filecard" on the screen.

5.2.5.13 REMOVE_CARD

CALL FORMAT:

REMOVE_CARD(LEVEL, PREVIOUS_CARD, PRE_ESCAPE_CARD);

where LEVEL is a value between 1..4, PREVIOUS_CARD and PRE_ESCAPE_CARD are A_CARD's

This procedure will remove the current card from the display, and then display the previous

card (to the current card in the "menu tree") on the screen. LEVEL is the level of the current card. PREVIOUS_CARD is the card record of the card previous to the current card. PRE_ESCAPE_CARD is the card previous to the previous card! The description below of the sample program will make clear the use of this procedure. This procedure is used when the user types an escape during the selection process to go back to the previous "filecard".

Upon exiting this procedure, the viewport will be set to the inside of the new "filecard" on the screen.

5.2.5.14 MAKE_TOP

CALL FORMAT:

MAKE_TOP(LEFT, MIDDLE, RIGHT);

where LEFT, MIDDLE, RIGHT are strings

This procedure will put the top display on the screen, placing the LEFT string left-justified on the first line, centering the MIDDLE string, and right-justifying the RIGHT string. The second line is left blank, and a line of "_" is then drawn.

This procedure makes a save-viewport control call to the console driver. The calling program should not assume that any viewports it has saved prior to this call will remain "restorable". Before exiting, this procedure will restore the previous viewport setting. The console driver only supports one level of save for viewports. Internal to the unit, this does not matter. Any program using the unit and is also making its own calls to the console driver (Save and Reset Viewport, Restore Viewport) should be aware of this.

5.2.5.15 MAKE_BOTTOM

CALL FORMAT:

MAKE_BOTTOM(LEFT, RIGHT);

where LEFT and RIGHT are strings

This procedure contructs the bottom portion of the display. The LEFT string is left-justified and the RIGHT string is right-jusitified on line 22. Line 21 is a line of "_" and line 23 is left blank.

This procedure makes a save-viewport control

call to the console driver. The calling program should not assume that any viewports it has saved prior to this call will remain "restorable". Before exiting, this procedure will restore the previous viewport setting. The console driver only supports one level of save for viewports. Internal to the unit, this does not matter. Any program using the unit and is also making its own calls to the console driver (Save and Reset Viewport, Restore Viewport) should be aware of this.

5.2.5.16 CLEAR_SCREEN

CALL FORMAT:

CLEAR_SCREEN;

This is a general procedure to clear the entire screen. The viewport is left set to the entire screen. This is used primarily to clear the screen at the beginning of a program and at the end.

5.2.5.17 INIT_A_MENU

CALL FORMAT:

INIT_A_MENU(MENU_LIST);

where MENU_LIST is A_MENU

This procedure will set up the initial values of the following fields in a menu record. Those fields are:

DO_POSITION - set to control code for absolute position

XPOS - set to 1 (second column in viewport)

YPOS - set from 1 to 9 depending on the number of menu items in the list

STATE - for item 1 set to INVERSE, for the other items set to NORMAL

The unit actually uses these data structures to paint the menu items in the "filecard". The absolute position control code and the XPOS and YPOS values determine where the text is placed. The STATE value determines whether or not the text is in INVERSE or normal text. This procedure defaults to displaying the menu as a single-spaced list in the "filecard". For example,

1. First menu item
2. Second menu item
3. Third menu item
.
.
.
9. Last menu item

A program can modify the values of XPOS and YPOS to control the positioning of the menu items in the "filecard".

The procedure also sets the field CURRENT_ITEM to 1.

5.2.5.18 GET_SELECTION

CALL FORMAT:

```
SELECTED := GET_SELECTION(MENU_LIST, SELECT_NUM,
              SHOW_MENU);
```

where MENU_LIST is A_MENU, SELECT_NUM is a VAR parameter to return the number of the item selected, and SELECTED is a program supplied BOOLEAN variable; SHOW_MENU is a BOOLEAN that specifies whether or not the menu display needs to be updated, if TRUE update the display, FALSE don't update the display

This is the main procedure to handle the complete selection process for a menu displayed in a "filecard". Once a card has been displayed via a MAKE_CARD call, GET_SELECTION is then called with the MENU_LIST for the current card. This call will paint the menu list in the "filecard" on the screen, with the first menu item in inverse. If the variable SHOW_MENU is FALSE, the menu will not be displayed. This assumes that the menu is already present on the screen.

At this point, the user can type one of the following things:

UP-ARROW - will move to the next item above in the list

DOWN-ARROW - will move to the next item below in the list

a number - typing a number will move to the item with that number in the list

RETURN - will return the number of the item

currently in INVERSE (selected) in the variable SELECT_NUM and GET_SELECTION will return true

ESCAPE - will return 0 in SELECT_NUM and GET_SELECTION will return false

typing anything else (or a number not included in the list) will cause a beep

A menu item displayed in inverse is considered to be the "chosen" item. To select that item requires the user to type RETURN. Moving to an item either with the arrow-keys or typing a number constitutes choosing an item.

After GET_SELECTION returns it is up to the calling program to act on the choice. The description of the SAMPLE program below will illustrate how this is done.

5.2.5.19 ERROR_BOX

CALL FORMAT:

ERROR_BOX;

This procedure will place an error box on the screen, saving the screen contents behind the box. See section 2.5 for a description of an error box.

This procedure will do a save viewport control code. Any previously saved viewport specifiction will be lost. Internal to the unit this does not matter. Any viewport set by the calling program will have to be managed by that program.

It is up to the calling program to place any text in an error box. This has to be done with calls to the console driver (Pascal has no knowledge of the current screen state!) The current viewport is set to the inside of the error box, so all display will "automatically" take place there.

5.2.5.20 GO_AWAY_ERROR

CALL FORMAT:

GO_AWAY_ERROR;

This procedure must be used in conjunction with ERROR_BOX. Once an error box has been

displayed, along with an error message (supplied by the calling program), the box can be removed from the screen by calling this procedure. It will restore the original screen contents and will reset the viewport to the values saved at the time of the call to ERROR_BOX.

Any text in an error box is removed by this call. The calling program does not need to remove it itself.

5.2.5.21 RESET_CARD_VP

CALL FORMAT:

RESET_CARD_VP(CARD_REC);

where CARD_REC is A_CARD

This procedure is used to set the viewport back to the inside of a "filecard" (CARD_REC) on the screen. It is used to update the screen during the selection process as cards are removed and replaced on the screen. See below for details on its use.

5.2.6 Using the Unit with Your Program

To use the unit requires a USES statement in your program of the form:

USES {$U library} FILECARD;

where library is the name of the library file where the unit is located.

5.3 A Sample Application that Uses the Unit

As an appendix, there is a listing of a sample program that utilizes the unit as the primary human interface code. This program has 9 different "filecards" arranged in the "menu tree" in Figure 3.3. Each menu has between two to five items, some of which point to other menus and others which branch off to "pseudo" activities. This program illustrates the type of data structures that are used to create the "menu tree", how to initialize the data, and how to organize the "main loop" in the program which controls the selection process.

5.3.1 Setting up the "Filecards"

The program defines an array of A_CARD which designates the "filecards" used by the program.

CARD: ARRAY[0..9] OF A_CARD;

The zeroth element of the array is used only to store a string (the menu_title) that is displayed for the topmost card's escape path. Elements 1 through 9 are the actual "filecards" that are displayed. For any program there should be an array, like that above, with 0..number_of_filecards.

The following fields in each A_CARD need to be initialized as follows:

MENU_NUMBER - the number assigned by the programmer

MENU_LEVEL - the level 1..4 in the "menu_tree"

P_CARD - the menu_number of the previous card in the menu_tree

MENU_TITLE - the title displayed in the upper left corner of the "filecard"

The procedure SET_CARDS shows such an initialization process. Most of the details used should have already been worked out in the initial design of the "menu tree".

5.3.2 Setting up the Menus

The program defines another array to hold the information about each of the menus associated with the "filecards".

```
MENU: ARRAY[1..9] OF A_MENU;
```

Each element 1 to 9 corresponds directly with each element 1 to 9 of the array CARD.

The first step is to initialize the fields that control the display of the menu items (DO_POSITION, XPOS, YPOS, and STATE). This is done through a call to the procedure INIT_A_MENU in the unit. Since there are nine menus to set up, a simple FOR-LOOP does the trick:

```
FOR I := 1 TO 9 DO INIT_A_MENU(MENU[I]);
```

This is found in the procedure INIT_THE_MENUS in the sample program.

The next step is to specify the number of items for each menu and the text to be displayed for each item in the menu. The procedures P1_SET_MENU_ITFMS and P2_SET_MENU_ITFMS show this process. (There are two procedures because of the size limitiations for the amount of code generated in a procedure!)

The standard set up used here will result in a single

spaced list displayed left-justified in each "filecard". You can modify this by changing the values of XPOS and YPOS in each MENU_ITEM. When changing these values, remember that the x and y values are treated relative to the viewport coordinates.

5.3.3 The Main Body of the Program

Other than the calls to the initialization procedures, the main body of the program consists of a large REPEAT statement that contains a large CASE statement. This CASE statement controls the flow of action during the selection process.

5.3.3.1 The Selection Process

There are three types of action that occur during the selection process:

Display the next menu in the "menu tree" selected by a menu item.

Return to the previous "filecard" in the "menu tree".

Branch off to an activity that requires a different display than the "filecard" display.

The first occurs when a user selects a menu item using either the arrow keys or a number key and then types RETURN. If this menu item selects another menu, then the new "filecard" and its menu must be displayed.

The second happens when the user types ESCAPE while a "filecard" is displayed. For a "filecard" with a level greater than 1, this will result in "moving back" to the previous "filecard" in the "menu tree" (the card directly underneath the current "filecard" on the display.) For the topmost "filecard" (level 1) the result of typing escape is up to the program to determine. For the sample program, it terminates the execution of the program.

The third happens when a user selects an item as in the first case, but here the item selects an activity and not another menu. In this case, the "filecard" display will more than likely be removed from the screen and a different display will replace it. When the activity is completed, the display should return to where it was left (the "filecard" where the user selected the activity.)

The display of the menu (not the "filecard")

and the handling of the selection process is done via a call to the function GET_SELECTION. The function will return FALSE if the user typed ESCAPE. It will return TRUE if the user selected an item and typed RETURN and it will return the number of the item selected.

5.3.3.2 Going through the Menu Tree

When the program begins, it needs to put up the first display. This consists of the top and bottom portions of the screen and the first "filecard". In the program, the procedure FIRST_SCREEN does this. At all times, there is a variable CURRENT_CARD which has the number of the currently selected "filecard". FIRST_SCREEN sets this to 1, the number of the topmost (level 1) card. This variable controls the flow of the display during the selection process. Another variable, OLD_CARD is used to store the number of the last selected "filecard". This value is used to determine whether or not the menu requires updating when the "filecard" is redisplayed. This occurs either because of an error box or when the program branches off to an activity.

Once the initial screen has been displayed, the selection process begins. This is found in the REPEAT loop in the main body of the sample program. The general structure of the REPEAT loop is:

```
REPEAT
  IF OLD_CARD <> CURRENT_CARD THEN
    {redisplay menu in "filecard"}
    SELECTED := GET_SELECTION(MENU[CURRENT_CARD],
                SEL_NUM, TRUE)
  ELSE
    {don't redisplay menu}
    SELECTED := GET_SELECTION(MENU[CURRENT_CARD],
                SEL_NUM, FALSE);
  CASE CURRENT_CARD OF
    1: {case for each "filecard"}
    2: {case for each "filecard"}
    .
    .
    .
    9: {case for each "filecard"}
  END;
UNTIL FALSE;
```

As CURRENT_CARD has been set to 1 as the program enters this loop, GET_SELECTION will display the menu for the "filecard" on the screen and wait

for the user's input. Each case in the CASE statement corresponds to the menu_number of the CURRENT_CARD. Thus once something has been selected, the CASE statement will process the selection given the CURRENT_CARD.

Each case (of CURRENT_CARD) reacts to the type of selection the user has made. The basic format for each case is:

```
BEGIN
  IF NOT(SELECTED) THEN BACK_UP
  ELSE
    BEGIN
      CASE SEL_NUM OF
        1: {case for each menu item}
        2: {case for each menu item}
        .
        .
        .
        9: {case for each menu item}
      END;
      GO_FORWARD;
    END;
END;
```

The case statement here corresponds to the menu items in the menu displayed. For menu items that select another menu ("menu selectors") the entry in the case simply sets CURRENT_CARD to the value of the menu now selected. For example, if menu item 3 selects "filecard" 4 then in the case statement the entry for 3 would be:

```
3: CURRENT_CARD := 4;
```

When a "menu selector" has been selected, a new "filecard" must be displayed. The CASE statement will set CURRENT_CARD to the value of the new "filecard" number. At the bottom of the CASE statement there is a call to a procedure called GO_FORWARD. This procedure will display the new "filecard" selected. The sample program includes this procedure. Its content is shown below:

```
PROCEDURE GO_FORWARD;

VAR PREVIOUS_CARD, OLD_ITEM: INTEGER;

  BEGIN
    IF OLD_CARD <> CURRENT_CARD THEN
      BEGIN
        PREVIOUS_CARD := CARD[CURRENT_CARD].P_CARD;
        MAKE_CARD(CARD[CURRENT_CARD],
```



```
          CARD[PREVIOUS_CARD]);
        OLD_ITEM := MENU[CURRENT_CARD].CURRENT_ITEM;
        MENU[CURRENT_CARD].LIST[OLD_ITEM].STATE :=
          NORMAL;
        MENU[CURRENT_CARD].LIST[1].STATE := INVERSE;
        MENU[CURRENT_CARD].CURRENT_ITEM := 1;
      END;
  END;
```

The procedure determines the previous card to the new current card and then calls MAKE_CARD to display the new "filecard". It also sets the new menu values for CURRENT_ITEM.

This procedure only works when the program is moving to a new "filecard". When an activity is selected, the program will return to the original screen, which does not require that a new "filecard" be displayed. However, the case will fall through this procedure call. The test at the beginning handles this event.

If the user types ESCAPE, it is necessary to go back to the previous card in the display. This requires that the current card be removed from the display. The sample program has a procedure, BACK_UP which handles this. For each case in the case of CURRENT_CARD (except the first level 1 card) there is a test for SELECTED. If it is FALSE, the user has typed ESCAPE, so BACK_UP. The content of this procedure is shown below:

```
PROCEDURE BACK_UP;

VAR PREVIOUS_CARD,
    ESCAPE_CARD:    INTEGER;

  BEGIN
    PREVIOUS_CARD := CARD[CURRENT_CARD].P_CARD;
    ESCAPE_CARD := CARD[PREVIOUS_CARD].P_CARD;
    REMOVE_CARD(CARD[CURRENT_CARD].MENU_LEVEL,
                CARD[PREVIOUS_CARD],
                CARD[ESCAPE_CARD]);
    CURRENT_CARD := PREVIOUS_CARD;
  END;
```

The procedure determines the previous card (to back up to...) and the escape card to update the top of the display. It then calls REMOVE_CARD to clean up the display. Finally it sets CURRENT_CARD to PREVIOUS_CARD, thus the current card to be displayed is the "previous" card in the display.

5.3.3.3 Branching Off to an Operation

When the user selects a menu item that is an "action selector", the program must now branch off to perform that action. This requires in most cases a new display on the screen, removing either the top, bottom, or "filecard" area, or all three from the screen. Removing an area requires:

1. Setting the viewport to that area of the screen that is to be cleared.

2. Saving the contents of that portion of the screen so that it can be restored.

3. Clearing that area with a Clear Viewport command.

This will now set up that area to be used for activity-specific display.

The sample program has a procedure called SET_UP_ACTIVITY, which sets the viewport to the entire screen, saves the screen contents, and then clears the screen.

5.3.3.4 Coming Back from an Operation

When an activity is complete the user is to return to the "filecard" display at the point at which it was left. This means that the screen should be restored back to its original contents prior to branching off to the activity.

When returning, first set the viewport back to that area which was removed. Then restore the contents back to the screen. This will put the display back to its original form. Since, CURRENT_CARD has not changed, GO_FORWARD will restore the original "filecard" and not a new one. Falling through the end of the CASE statement will then bring us back to GET_SELECTION which will display the original menu. Thus the selection process begins anew at the place left when the user selected a "action selector".

The sample program has a procedure called RETURN_FROM_ACTIVITY which sets the viewport to the entire screen and then restores the screen contents, thus returning back to the "filecard" display at the point it was left.

5.3.3.5 Performing an Activity

In the case (of CURRENT_CARD), for each

card, there is a CASE statement corresponding to each item in the menu. For those items that are "activity selectors", the case has the form:

```
CASE SEL_NUM OF
.
n: BEGIN {branch off to an activity}
     SET_UP_ACTIVITY;
     DO_ACTIVITY;
     RETURN_FROM_ACTIVITY;
   END;
.
END;
```

If SET_UP_ACTIVITY and RETURN_FROM_ACTIVITY are properly done (see above) the code for the activity itself does not have to worry about maintaining the integrity of the "filecard" display itself.

5.3.3.6 Reporting an Error

If there is an error to report, the error box procedures supplied by the unit facilitate the reporting process. The sample program has a procedure which presents a simple error, DO_AN_ERROR.

```
PROCEDURE DO_AN_ERROR;

  BEGIN
    ERROR_BOX;
    PUT_ERROR_MSG;
    PAUSE;
    GO_AWAY_ERROR;
  END;
```

The basic format is to display an error box and then to display a message in the error box. PAUSE waits for the user to read the message and then to type something to exit. GO_AWAY_ERROR then cleans the error box from the screen.

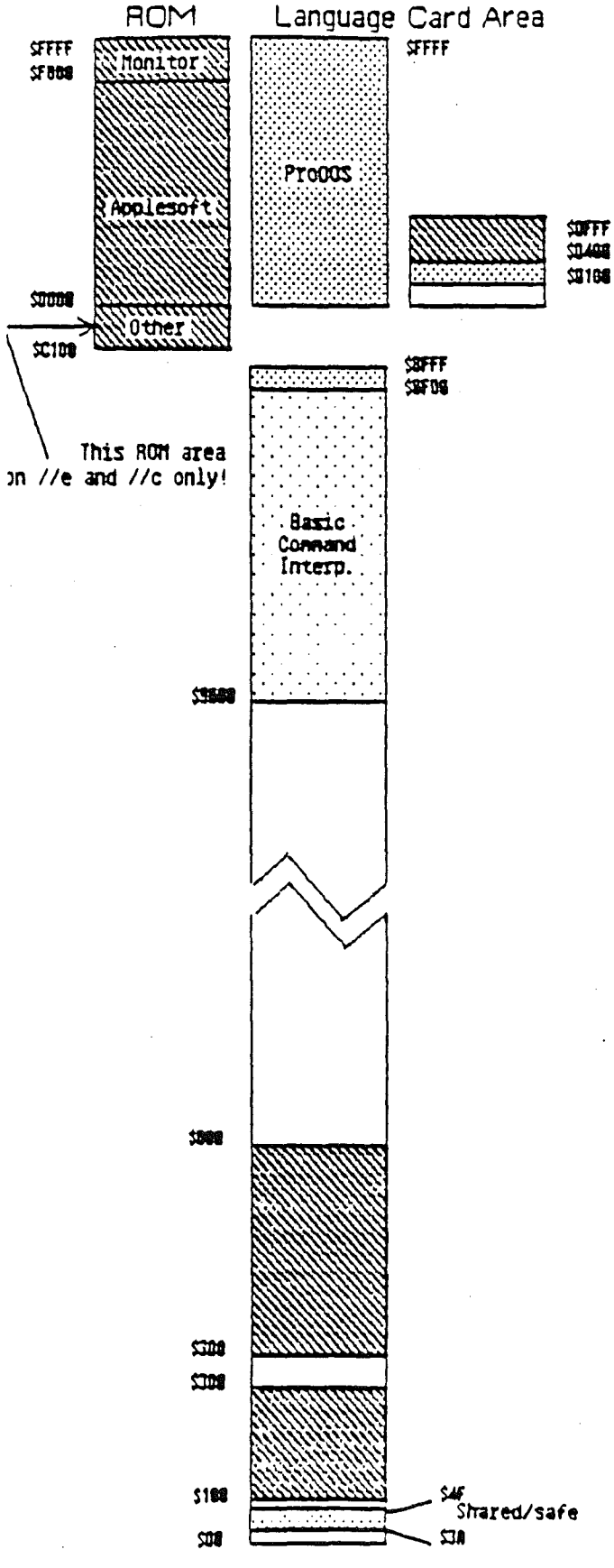
ProDOS Technical Notes

Revised May 08, 1984

For further information contact:
PCS Developer Technical Support
M/S 22-W. Phone (408) 996-1010

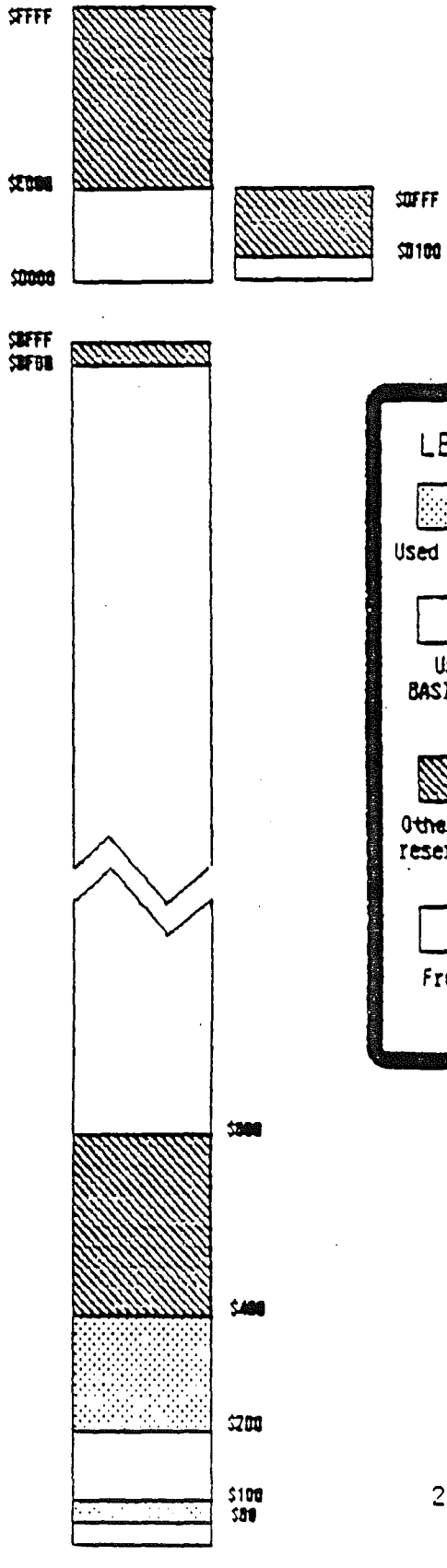
ProDOS Memory Map

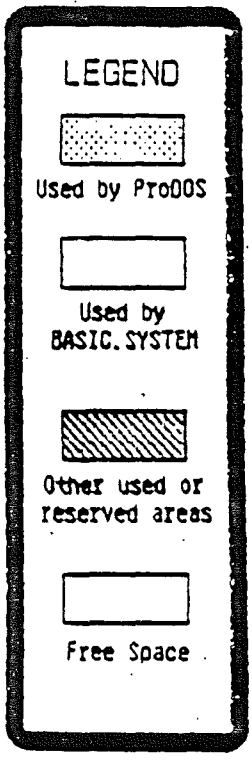
Main Memory



Auxiliary Memory

(//c or 128K //e only)





28 June 1984

ProDOS TECHNICAL NOTE #1

The GETLN Input Buffer and the ThunderClock

(14 July 1983)

The ThunderClock is automatically supported by ProDOS when ever it is identified as installed in the system. When programming under ProDOS, always consider the ThunderClock's impact on the GETLN input buffer ($200 - $2FF). ProDOS can support other clocks which may also use this space.

When ever the ThunderClock receives a call from ProDOS, it deposits an ASCII string in the GETLN input buffer of the form:

07,04,14,22,46,57

which translates as:

07 = The month, JULY (01=JAN,...,12=DEC)
04 = The day-of-the-week, THURSDAY (00=SUN,...,06=SAT)
14 = The date, 14th (00 to 31)
22 = The hour, 10PM (00 to 23)
46 = The minute (00 to 59)
57 = The second (00 to 59)

ProDOS calls the ThunderClock as part of many of its routines. Anything in the first 17 bytes of the GETLN input buffer is subject to loss if a ThunderClock is installed and gets called.

It has been the practice of programmers, in general, to use the GETLN input buffer for every conceivable purpose. Therefore, an application should never store anything there. If your application has future need to know about the contents of the $200-$2FF space, it should be transferred to some other location to guarantee it will remain intact, particularly under ProDOS where a ThunderClock may regularly be overwriting the first 17 bytes.

APPLE COMPUTER, Inc., PCS Developer Technical Support
20525 Mariani Avenue, M/S 22-W
Cupertino, CA 95014
Phone (408) 554-5213 or (408) 996-1010

ProDOS TECHNICAL NOTE #2

Notes on Transporting DOS Assembly Language Programs to ProDOS (Passing Disk Commands Under BASIC.SYSTEM to ProDOS from Machine Code.)

(Revised August 7, 1984)

Under DOS, commands were executed by a direct call to the proper address in DOS or by sending a string to COUT ($FDED) consisting of [CTRL-D] <command> [RETURN].

The practice that became very common under DOS of making direct calls to the desired routines within DOS cannot be carried over to ProDOS. Apple Computer will not support any entries into the BASIC Command Interpreter or the ProDOS kernel that are not published by Apple. If you use any undocumented entries, your application will almost certainly not operate under future releases of PRODOS and BASIC.SYSTEM.

Passing disk commands as ASCII strings to COUT is not supported under ProDOS.

If you wish to issue a ProDOS command from a machine language module operating with Applesoft or if your application can permit the ProDOS BASIC Command Interpreter (BASIC.SYSTEM) to be co-resident in memory, you can still use an ASCII string. All that is necessary is to move the string, ending with a RETURN ($8D) to the GETLN buffer ($200) and execute a JSR DOSCMD ($BE03) to execute the instruction at $200.

*** It is necessary that the JSR DOSCMD be performed in deferred mode (inside a program) and not in immediate mode. This also applies to the monitor program; while in the monitor you cannot do a $xxxxG to execute the code that contains the JSR DOSCMD. The reason for this is that BASIC.SYSTEM checks certain state flags. These flags are set correctly for the DOSCMD routine only while in deferred mode. DOSCMD was intended only to be used via a CALL inside a BASIC program.

There are certain commands that will NOT work correctly or as expected when initiated via DOSCMD. The following table lists those commands which work properly and those that do not.

PLEASE NOTE that some of the commands listed as not working properly may work well enough to suit your individual purposes. Also some commands will function (albeit precariously) in immediate mode. IF YOU DECIDE TO USE THE COMMANDS IN THIS MANNER YOU ARE ON YOUR OWN.

Attached is an example BASIC program that will BLOAD an assembly routine that will exercise the DOSCMD routine. The BASIC program is first LISTed and then RUN. A listing of the assembly routine follows. Please review it before writing your own routine.

DOSCMD is merely a means of performing some BASIC.SYSTFM commands from assembly language and is not a substitute for performing the commands in BASIC. Keep in mind all the conseqences of the command you are executing; EG. When doing a BRUN or BLOAD make sure the code is loaded at suitable addresses.

Error Handling

Right after you call DOSCMD the carry bit will tell you whether or not an error had occurred. The carry will be set if an error had occurred. The accumulator will always have the error number.

DOSCMD error handling can be handled in one of three ways:

1. Do a JSR ERROUT ($BE09). This will return control to your BASIC ONERR routine where you can then handle the error.

2. Do a JSR PRINTERR ($BE0C). This will print out the error and will return control to the point after the JSR (as usual).

3. You can handle the error yourself completely. If choose to go this route make sure you clear the carry (CLC) before you return control back to BASIC.SYSTEM. If you don't it will be assumed some error has occurred and will do awful and unpredictable things to you.

Works Correctly and Returns Control to Calling Routine	Works Incorrectly and/or does not Return Control to Calling Routine
Filing Commands:	
Catalog, Cat	
Prefix, Prefix /pn	
Create	
Rename	
Delete	
Lock	
Unlock	
Program Commands:	
	- (Dash)
	Run
	Load
Save	
Programming Commands:	
	Chain
Store	
Restore	
Pr#	
In#	
Fre	
Text File Commands:	
Open	
Close	
	Read
	Write
	Append
Flush	
Position	
EXEC Command:	
	Exec
Binary Commands:	
Brun	
Bload	
Bsave	

```
 10   REM  YOU MUST CALL THE ROUTINE FROM INSIDE A BASIC PROGRAM!!
      REM
 12   REM
 20   PRINT  CHR$ (4)"BLOAD/P/PROGRAMS/CMD.0"
 30   CALL 4096
 40   PRINT "BACK TO THE WONDERFUL WORLD OF BASIC!"
 50   END

]RUN


ENTER BASIC.SYSTEM COMMAND --> PREFIX

/P/


ENTER BASIC.SYSTEM COMMAND --> PREFIX/P/BUGS

ENTER BASIC.SYSTEM COMMAND --> PREFIX

/P/BUGS/


ENTER BASIC.SYSTEM COMMAND --> CATALOG

BUGS

 NAME            TYPE  BLOCKS  MODIFIED          CREATED           ENDFILE SUBTYPE

*SEQTEST          DIR       1  23-APR-84 16:12  23-APR-84 16:12       512
 WRITEFIELDS      BAS       1  27-MAR-84 15:00  23-APR-84 16:13       182
 R                BAS       1  27-MAR-84 15:29  23-APR-84 16:13       193
 READFIELDS       BAS       1  27-MAR-84 15:17  23-APR-84 16:13       185
 DUMPFILE         BAS       1  27-MAR-84 11:01  23-APR-84 16:13       191
 POSTEST          BAS       1  27-MAR-84 16:50  23-APR-84 16:13       174
 MAKEJUNK         BAS       1  29-MAR-84 14:10  23-APR-84 16:14        82
 P1               BAS       1   3-AUG-84 17:53  23-APR-84 16:15       416

BLOCKS FREE: 6215     BLOCKS USED: 3513     TOTAL BLOCKS: 9728


ENTER BASIC.SYSTEM COMMAND --> DO DA, DO DA

SYNTAX ERROR
BACK TO THE WONDERFUL WORLD OF BASIC!
```

```
SOURCE   FILE #01 =>/P/PROGRAMS/CMD
------ NEXT OBJECT FILE NAME IS /P/PROGRAMS/CMD.0
1000:        1000    1          ORG   $1000
1000:        FD6F    2 GETLN1   EQU   $FD6F        ; MONITORS INPUT ROUTINE
1000:        BE03    3 DOSCMD   EQU   $BE03        ; BASIC.SYSTEMS GLBL PG DOS CMD ENTRY
1000:        FDED    4 COUT     EQU   $FDED        ; MONITORS CHAR OUT ROUTINE
1000:        BE0C    5 PRERR    EQU   $BE0C        ; PRINT THE ERROR
1000:                6 *
1000:                7 *
1000:                8 *
1000:A2 00           9 START    LDX   #0           ; DISPLAY PROMPT...
1002:BD 1F 10       10 L1       LDA   PROMPT,X     ;
1005:F0 06   100D   11          BEQ   CONT         ; BRANCH IF END OF STRING
1007:20 ED FD       12          JSR   COUT         ;
100A:E8             13          INX                ;
100B:D0 F5   1002   14          BNE   L1           ; LOOP UNTIL NULL TERMINATOR IS HIT...
100D:               15 *
100D:20 6F FD       16 CONT     JSR   GETLN1       ; NOW ACCEPT USER COMMAND FROM KB
1010:20 03 BE       17          JSR   DOSCMD       ; AND EXECUTE THE COMMAND
1013:2C 10 C0       18          BIT   $C010        ; CLEAR STROBE SO KEY WON'T HANG AROUND..
1016:B0 02   101A   19          BCS   ERROR        ; BRANCH IF ERROR DETECTED
1018:90 E6   1000   20          BCC   START        ; OTHERWISE RESTART....
101A:               21 *
101                 22 *
101                 23 * NOTE: AFTER HANDLING YOUR ERROR YOU MUST CLEAR THE CARRY
101A:               24 *       BEFORE RETURNING TO BASIC OR ELSE BASIC WILL DO
101A:               25 *       STRANGE THINGS TO YOU.
101A:               26 *
101A:20 0C BE       27 ERROR    JSR   PRERR        ; PRINT 'ERR'
101D:18             28          CLC                ;
101E:60             29          RTS                ; RETURN TO BASIC
101F:               30 *
101F:               31          MSB   ON
101F:               32 *
101F:8D             33 PROMPT   DB    $8D          ; OUTPUT A RETURN FIRST
1020:C5 CE D4 C5    34          ASC   'ENTER       BASIC.SYSTEM COMMAND —> '
103F:00             35          DB    0
```

```
 100D CONT          FDED COUT          BE03 DOSCMD        101A ERROR
 FD6F GETLN1        1002 L1            BE0C PRERR         101F PROMPT
 1    START
** SUCCESSFUL ASSEMBLY := NO ERRORS
** ASSEMBLER CREATED ON 15-JAN-84 21:28
** TOTAL LINES ASSEMBLED     35
** FREE SPACE PAGE COUNT   89
```

ProDOS TECHNICAL NOTE #3

ProDOS Device Search and Identification Procedure
Disk Driver Conventions

(Revised 20 December 1983)

During boot-up, ProDOS does a device search looking for block storage devices. As described in the ProDOS Technical Reference Manual, all disk drives must "look and act just like one of our drives".

ProDOS looks for the following:

$Cn01 = $20 $Cn03=$00 $Cn05=$03

where n = the slot number. Having found these three bytes in the ROM of a particular slot, ProDOS assumes it has found a disk drive.

If $CnFF=$00 ProDOS assumes it has found a Disk][with 16-sector ROMs and marks the device driver table in the ProDOS global page with the address of the Disk][driver routines. The Disk][driver routines will support any drive that "looks and acts like a Disk][" (280 blocks, single volume, etc.).

If $CnFF=$FF, ProDOS assumes it has found a Disk][with 13-sector ROMs and makes no attempt to support the device 13-sector ROMs since it may not operate properly under ProDOS.

If ProDOS finds a value other than $00 or $FF at $CnFF, it assumes it has found an "intelligent" disk controller. If the STATUS BYTE at $CnFE indicates that the device supports READ and STATUS requests, ProDOS marks the global page with a device driver address whose high-byte is equal to $Cn and whose low-byte is equal to the value found at $CnFF. Intelligent controller cards CANNOT be auto-bootable due to a conflict with Pascal which believes all auto-boot devices are Disk][floppy drives. (Therefore, the byte at $Cn07 must not be $3C.)

The only calls to the disk driver are STATUS, READ, WRITE, and FORMAT. The STATUS call should perform a check to verify that the device is ready for a READ or WRITE. If it is not, the carry should be set and the appropriate error code returned in the accumulator. If the device is ready for a READ or WRITE, then the driver should clear the carry, place a zero in the accumulator, and return the number of blocks on the device in the X-register (lo-byte) and Y-register (hi-byte).

If you wish to interface a disk controller card with more than two drives (or a device with more than two volumes), additional device driver vectors for disk controllers plugged into slot 5 or 6 may be installed in slot 1 or 2 locations. There will be no conflict with character devices physically present in these slots. Device numbers for four drives in slot five or slot six are listed below.

Physical	S5,D1 = $50	Physical	S6,D1 = $60
Slot	S5,D2 = $D0	Slot	S6,D2 = $E0
Five	S1,D1 = $10	Six	S2,D1 = $20
	S1,D2 = $90		S2,D2 = $A0

The special locations in the ROM code are:

$CnFC-$CnFD = The total number of blocks on the device. Used for writing the disk's bit-map and directory header after formatting. (If this location is $0000, it indicates that the number of blocks must be obtained by making a STATUS request.)

$CnFE = The status byte (bit 0 and 1 must be set for ProDOS to install the driver vector!)

Bit 7	- Medium is removable	
Bit 6	- Device is interruptable	
Bit 5-4	- Number of volumes on the device (0-3)	
Bit 3	- The device supports formatting	
Bit 2	- The device can be written to	
Bit 1	- The device can be read from	(Must be on)
Bit 0	- The device's status can be read	(Must be on)

$CnFF = The lo-byte of entry to the driver routines...ProDOS will place $Cn + this byte in the global page.

The locations where the call parameters are passed to the driver are:

$42 - COMMAND: 0 = STATUS request 1 = READ request
2 = WRITE request 3 = FORMAT request

NOTE: The FORMAT code in the driver need only lay down address marks if required...the calling routine should write the "virgin directory and bit-map".

$43 - UNIT NUMBER:

7	6	5	4	3	2	1	0
,DR		,SLOT		not		used	

NOTE: The UNIT_NUMBER that appears in the device list (DEVLST) in the system globals will include the hi-nybble of the status byte ($CnFE) as an I.D. in it's lo-nybble.

$44-$45 - BUFFER POINTER: Indicates the start of a 512-byte memory buffer for data transfer.

$46-$47 - BLOCK NUMBER: Indicates the block on the disk for data transfer.

The device driver should report errors by setting the carry flag and loading the error code into the accumulator. The error codes that should be implemented are:

$27 - I/O error $28 - No device connected $2B - Write Protected

APPLE COMPUTER, Inc., PCS Developer Technical Support
20525 Mariani Avenue, M/S 22-W
Cupertino, CA 95014
Phone (408) 554-5213 or (408) 996-1010

ProDOS TECHNICAL NOTE #4

Notes on Transporting DOS Assembly Language Programs to ProDOS (Redirecting I/O and converting "JSR $3EA")

(26 July 1983)

When programming under DOS 3.3, if you wished to change the I/O hooks, all that was necessary was to install your I/O routine addresses in the character-out vector ($36-$37) and/or key-in vector ($38-$39) and notify DOS (JSR $3EA) to take your addresses and swap it's intercept routine addresses in.

Under ProDOS, there is no instruction installed at $3EA at all. So what do you do?

Just leave the ProDOS Basic Command Interpreter's intercept addresses installed in $36-$39 and install your I/O addresses in the global page at $BE30-$BE33. $BE30-$BE31 should contain the output address (normally $FDF0, the monitor COUT1 routine), and $BE32-$BE33 should contain the input address (normally $FD1B, the monitor KEYIN routine).

By keeping these vectors in a global page, a special routine for moving the vectors is no longer needed, thus, no $3EA instruction. Just install the addresses at their destination yourself.

APPLE COMPUTER, Inc., PCS Developer Technical Support
20525 Mariani Avenue, M/S 22-W
Cupertino, CA 95014
Phone (408) 554-5213 or (408) 996-1010

ProDOS TECHNICAL NOTE #5

ProDOS Disk Formatting Routines

(11 January 1984)

The ProDOS Disk][FORMATTER and ProDOS BUILDDISK Routines are supplied as text files of source code. They can be assembled with the ProDOS version of EDASM, Apple's editor/assembler.

The source code for the FORMATTER was prepared with no labels so that you can "INCLUDE" it with your application at assembly time. Since disk I/O core routines MUST include critical, time dependent code, the FORMATTER source file MUST be assembled with the "ORG" on a page boundary. (Many instruction times change when page boundaries are crossed.)

The formatter routine uses zero page locations $D0 thru $DD. If your application also uses these locations, you must save the contents prior to calling the formatter and restore them upon return.

When the routine is called, the ProDOS device number (DEVNUM) must be in the accumulator. DEVNUM in this case is defined as containing zeros in the low nibble, the slot number in bits 4, 5, 6, and the hi-bit set to zero for drive 1 or set to 1 for drive 2. Upon exit, if the carry flag is clear, no error has been detected and the accumulator will be zeroed.

If an error has been detected, the routine will exit with the carry flag set and the accumulator will hold the error code. Error codes that may be returned are: $27-unable to format, $28-write protected, $33-drive too slow, $34-drive too fast.

The FORMATTER routine ONLY writes zeros to each sector on a Disk][floppy. To install boot code, a directory and bit map, on any previously formatted storage device, you need the BUILDDISK routine.

Upon entry to the BUILDDISK routines the accumulator must contain the DEVNUM, X and Y must have the address of a 512 byte buffer (X-lo, Y-hi), and DUMMYNAM and DUMSIZE must be filled in with the desired volume name and name length if a name other than DEFAULT.NAME is desired.

BUILDDISK treats all devices the same, with two exceptions. These exceptions are identified by examining the low nibble of the DEVNUM. (Remember, the low nibble of the DEVNUM is derived from the high nibble of the device status byte at $CnFE in the ROM code.)

If all four bits of the i.d. nibble are set, BUILDDISK will assume that the device has unusual characteristics and that the driver has taken care of the bit map, directory and boot code during the format request. If all four bits are clear, BUILDDISK will recognize the device as a Disk][or Disk][emulator and assume the device has 280 blocks.

BUILDDISK leaves zero-page intact, with the exception of the bytes from $42 thru $47 which are defined for use when making requests to device drivers and standard ProDOS error codes will be returned.

APPLE COMPUTER, Inc., PCS Developer Technical Support
20525 Mariani Avenue, M/S 22-W
Cupertino, CA 95014
Phone (408) 554-5213 or (408) 996-1010

ProDOS TECHNICAL NOTE #6

Attaching External Commands to BASIC.SYSTEM

(Revised 19 September 1983)

Whenever BASIC.SYSTEM receives a command, it first checks it's command list, then sends it out to any external command handler and finally passes it on to Applesoft. If you find regular need for an additional command, you can write your own command handler and attach it to BASIC.SYSTEM through the EXTRNCMD jump vector. Just install the address of your routine in EXTRNCMD+1 and +2 (lo-byte first) and you're linked in. There are essentially three functions that your routine must perform.

(1) It must check for the presence of your command(s).

(2) If it is your command, it must let BASIC.SYSTEM know.

(3) It must execute the desired instructions expected of the command.

The first step (1) is quite straight forward, just inspect the GETLN input buffer. If it is not your command, a simple SEC and a RTS will return control to BASIC.SYSTEM to continue the search.

The second step (2) is more involved. It is your command, so you must zero XCNUM ($BE53) to indicate an external command and set XLEN ($BE52) equal to the length of your command string minus one.

If there are no associated parameters (such as slot, drive, A$, etc.) to parse, you must set all 16 parameter bits in PBITS ($BE54,$BE55) to zero. And, if you're going to handle everything yourself before returning control to BASIC.SYSTEM you must point XTRNADDR ($BE50, $BE51) at an RTS instruction...XRETURN ($BE9E) is a good location. Now just "fall through" to your execution routines (3).

If there are parameters to parse, it is easiest to let BASIC.SYSTEM parse them for you (unless you want to use some undefined parameters). By setting up the bits in PBITS ($BE54,$BE55), and setting XTRNADDR ($BE50,$BE51) equal to the location where execution of your command begins, you can return control to BASIC.SYSTEM, with an RTS, and let it parse and verify the parameters and return them to you in the global page.

The final step (3) is up to you and should RTS with the carry cleared.

Attached are two example routines, BEEP and BEEPSLOT. BEEP handles everything itself and BEEPSLOT will let you pass a slot & drive parameter (,S#,D#), where the drive is ignored.

APPLE COMPUTER, Inc., PCS Developer Technical Support
20525 Mariani Avenue, M/S 22-W
Cupertino, CA 95014
Phone (408) 554-5213 or (408) 996-1010

BRUN BEEP.0 to install the routine's address in EXTRNCMD. Then type BEEP as immediate command or use PRINT CHR$(4);"BEEP" in a program.

```
0300:          0300   1            ORG   $300
0300:          0200   2 INBUF      EQU   $200          ;GETLN input buffer
0300:          FCA8   3 WAIT       EQU   $FCA8         ;Monitor wait routine
0300:          FF3A   4 BELL       EQU   $FF3A         ;Monitor bell routine
0300:          BE06   5 EXTRNCMD   EQU   $BE06         ;External cmd JMP vector
0300:          BE50   6 XTRNADDR   EQU   $BE50         ;Ex cmd implementation addr
0300:          BE52   7 XLEN       EQU   $BE52         ;Length of command string-1
0300:          BE53   8 XCNUM      EQU   $BE53         ;CI cmd no. (ex cmd = 0)
0300:          BE54   9 PBITS      EQU   $BE54         ;Command parameter bits
0300:          BE9E  10 XRETURN    EQU   $BE9E         ;Known RTS instruction
0300:                11            MSB   ON            ;Set hi-bit on ASCII
0300:                12 ;
0300:A9 0B           13            LDA   #>BEEP        ;Install the address of our
0302:8D 07 BE        14            STA   EXTRNCMD+1    ;  command handler in the
0305:A9 03           15            LDA   #<BEEP        ;  external command JMP
0307:8D 08 BE        16            STA   EXTRNCMD+2    ;  vector
030A:60              17            RTS
030B:                18 ;
030B:A2 00           19 BEEP       LDX   #0            ;Check for our command
030D:BD 00 02        20 NXTCHR     LDA   INBUF,X       ;Get first char
0310:DD 43 03        21            CMP   CMD,X         ;Does it match?
0313:D0 2E    0343   22            BNE   RETURN        ;Nope, back to CI
0315:E8              23            INX                 ;Next character
0316:E0 04           24            CPX   #CMDLEN       ;All characters yet?
0318:D0 F3    030D   25            BNE   NXTCHR        ;No, read next one
031A:                26 ;
031A:A9 03           27            LDA   #CMDLEN-1     ;Our cmd!  Put cmd length
031C:8D 52 BE        28            STA   XLEN          ;  -1 in CI global XLEN
031F:A9 9E           29            LDA   #>XRETURN     ;Point XTRNADDR to a known
0321:8D 50 BE        30            STA   XTRNADDR      ;  RTS since we'll handle
0324:A9 BE           31            LDA   #<XRETURN     ;  at the time we inter-
0326:8D 51 BE        32            STA   XTRNADDR+1    ;  cept our command
0329:A9 00           33            LDA   #0            ;Mark the cmd number as
032B:8D 53 BE        34            STA   XCNUM         ;  zero (external)
032E:8D 54 BE        35            STA   PBITS         ;And indicate no paramet
0331:8D 55 BE        36            STA   PBITS+1       ;  to be parsed
0334:                37 ;
0334:A2 05           38            LDX   #5            ;Number of desired beeps
0336:20 3A FF        39 NXTBEEP    JSR   BELL          ;Else, beep once
0339:A9 80           40            LDA   #$80          ;Set-up the delay
033B:20 A8 FC        41            JSR   WAIT          ;  and wait
033E:CA              42            DEX                 ;Decrement index and
033F:D0 F5    0336   43            BNE   NXTBEEP       ;  repeat til X = 0
0341:18              44            CLC                 ;All done successfully
0342:60              45            RTS
0343:                46 ;
0343:38              47 RETURN     SEC                 ;Notify BASIC.SYSTEM it
0344:60              48            RTS                 ;  it wasn't our command
0345:                49 ;
0345:C2 C5 C5 D0     50 CMD        ASC   "BEEP"        ;Our command
0349:          0004  51 CMDLEN     EQU   *-CMD         ;Our Command length
```

BRUN BEEPSLOT.O to install the routine's address in EXTRNCMD. Then enter BEEPSLOT,S(n),D(n). Only a legal slot and drive numbers are acceptable. If no slot number, it will use the default slot number. Any drive number is simply ignored. The command may also be used in a program PRINT CHR$(4) statement.

```
0300:          0300   1            ORG   $300
0300:          0200   2 INBUF      EQU   $200          ;GETLN input buffer
0300:          FCA8   3 WAIT       EQU   $FCA8         ;Monitor wait routine
0300:          FF3A   4 BELL       EQU   $FF3A         ;Monitor bell routine
0300:          BE06   5 EXTRNCMD   EQU   $BE06         ;External cmd JMP vector
0300:          BE50   6 XTRNADDR   EQU   $BE50         ;Ex cmd implementation addr
0300:          BE52   7 XLEN       EQU   $BE52         ;Length of command string-1
0300:          BE53   8 XCNUM      EQU   $BE53         ;CI cmd no. (ex cmd = 0)
0300:          BE54   9 PBITS      EQU   $BE54         ;Command parameter bits
0300:          BE61  10 VSLOT      EQU   $BE61         ;Verified slot parameter
0300:                11            MSB   ON            ;Set hi-bit on ASCII
0300:                12 ;
0300:A9 0B           13            LDA   #>BEEPSLOT    ;Install the address of our
0302:8D 07 BE        14            STA   EXTRNCMD+1    ;  command handler in the
0305:A9 03           15            LDA   #<BEEPSLOT    ;  external command JMP
0307:8D 08 BE        16            STA   EXTRNCMD+2    ;  vector
030A:60              17            RTS
030B:                18 ;
030B:A2 00           19 BEEPSLOT   LDX   #0            ;Check for our command
030D:BD 00 02        20 NXTCHR     LDA   INBUF,X       ;Get first char
0310:DD 4B 03        21            CMP   CMD,X         ;Does it match?
0313:D0 36     034B  22            BNE   RETURN        ;Nope, back to CI
0315:E8              23            INX                 ;Next character
0316:E0 08           24            CPX   #CMDLEN       ;All characters yet?
0318:D0 F3     030D  25            BNE   NXTCHR        ;No, read next one
031A:                26 ;
031A:A9 07           27            LDA   #CMDLEN-1     ;Our cmd!  Put cmd length
031C:8D 52 BE        28            STA   XLEN          ;  -1 in CI global XLEN
031F:A9 38           29            LDA   #>EXECUTE     ;Point XTRNADDR to our
0321:8D 50 BE        30            STA   XTRNADDR      ;  command execution
0324:A9 03           31            LDA   #<EXECUTE     ;  routine
0326:8D 51 BE        32            STA   XTRNADDR+1
0329:A9 00           33            LDA   #0            ;Mark the cmd number as
032B:8D 53 BE        34            STA   XCNUM         ;  zero (external)
032E:8D 54 BE        35            STA   PBITS         ;And indicate that slot and
0331:A9 04           36            LDA   #%00000100    ;  drive parameter may be
0333:8D 54 BE        37            STA   PBITS         ;  accepted
0336:18              38            CLC                 ;Everything if OK
0337:60              39            RTS                 ;Return to BASIC.SYSTEM
0338:                40 ;
0338:AD 61 BE        41 EXECUTE    LDA   VSLOT         ;Get slot parameter
033B:29 0F           42            AND   #%00001111    ;Zero the hi-bits
033D:AA              43            TAX                 ;Transfer to index reg.
033E:20 3A FF        44 NXTBEEP    JSR   BELL          ;Else, beep once
0341:A9 80           45            LDA   #$80          ;Set-up the delay
0343:20 A8 FC        46            JSR   WAIT          ;  and wait
0346:CA              47            DEX                 ;Decrement index and
0347:D0 F5     033E  48            BNE   NXTBEEP       ;  repeat til X = 0
0349:18              49            CLC                 ;All done successfully
034A:60              50            RTS
034B:                51 ;
034B:38              52 RETURN     SEC                 ;Notify BASIC.SYSTEM, it
034A:60              53            RTS                 ;  wasn't our command
034B:                54 ;
034B:C2 C5 C5 D0     55 CMD        ASC   "BEEPSLOT"    ;Our command
0353:          0008  56 CMDLEN     EQU   *-CMD         ;Our Command length
```

ProDOS TECHNICAL NOTE #7

Starting and Quitting
Interpreter Conventions

(revised 09 March 1984)

It is absolutely essential that all interpreters (system programs) use a standard way of starting and quitting.

In order to provide a uniform method for starting and quitting, the following procedures are established and SUPERCEDE section 5.1.5 of the ProDOS Technical Reference Manual:

Starting:

System Programs are started by one of two ways:

1. The disk containing the ProDOS operating system and the system program is booted; ProDOS loads and runs the first XXX.SYSTEM file of type SYS($FF). The order of search is determined by the file entries in the boot volume directory.

2. The program is loaded by another program (like the ProDOS filer or the Basic Command Interpreter), or a program dispatcher (like the one that is part of ProDOS or a more sophisticated program selector).

The system program is loaded and jumped to at $2000. The complete or partial pathname of the system program is stored at $280 starting with a length byte. The string is a full pathname if it starts with a slash (/); it is a partial pathname if it starts with a letter.

The purpose of this pathname is to allow a system program to determine the directory where other needed files may reside. The program should NEVER assume that the files are in a specific directory or subdirectory.

Additionally, we establish a mechanism to pass a second pathname to interpreters which like to run STARTUP programs. An example of this is a language interpreter. The ProDOS dispatcher does not support this mechanism but other more sophisticated program selectors may.

The mechanism requires that the interpreter start a certain way:

- o $2000 is a jump instruction.
- o $2003 and $2004 are $EE.

If the interpreter starts this way, byte $2005 is assumed to be an indicator of the length of a buffer which starts at $2006 and holds the pathname (starting with a length byte) of the startup file.

Interpreters which support this mechanism should supply their own default string which should be a standard choice for a startup program or a flag not to run a startup program.

Once gaining control, the system program sets the reset vector and fixes the power-up byte. Never assume the state of the machine to be anything that is not clearly documented.

Note: If your interpreter makes use of the dispatcher/selector area (addresses $D100-$D3FF in the second 4K-byte bank of RAM), be sure that this area is initially saved and then restored on exit.

Quitting:

1. Do normal housekeeping... close files, reinstall /RAM if you have had it disconnected, etc.

2. Trash the power-up byte at $3F4. The simplest way to do this is either to increment or decrement it, which will always make it an invalid check of the $3F2 vector.

3. Execute a ProDOS system call number $65 as follows:

```
EXIT      JSR  PRODOS      ; Call the MLI ($BF00)
          DFB  $65         ; CALL TYPE = QUIT
          DW   PARMTABLE   ; Pointer to parameter table

PARMTABLE DFB  4           ; Number of parameters is 4
          DFB  0           ; 0 is the only quit type
          DW   0000        ; Pointer reserved for
;                            future use
          DFB  0           ; Byte reserved for future
;                            use
          DW   0000        ; Pointer reserved for
;                            future use.
```

It is most important to note that even though most of the parameter table is reserved for future use, it must all be present! It must consist of seven bytes... a $04 followed by six nulls ($00).

For more information on Dispatcher/Selector Conventions please see ProDOS Technical Note #14.

APPLE COMPUTER INC., PCS Developer Technical Support
20525 Mariani Avenue, M/S 22-W
Cupertino, CA 95014
Phone: (408) 996-1010

ProDOS Technical Note #8

August 13, 1984

This technical note explains:

1. How to protect auxiliary bank graphics pages from /RAM,
2. How to disconnect and reinstall /RAM (or some other device)

For further information contact:
PCS Developer Technical Support
M/S 22-W. Phone (408) 996-1010

ProDOS TECHNICAL NOTE #8

- Protecting Auxiliary Bank Hi-Res Graphics Pages -
- Disconnecting and Re-installing /RAM -
- Convention on How to Treat Ram Disk's with >64K -

(Revised August 13, 1984)

When ProDOS is booted a check is made of the environment. If a 128K Apple // system is found, the auxiliary 64K bank of memory is configured as a ram disk named /RAM that will appear as slot 3 drive 2 (since it is memory on the 80 column card which appears in slot 3). /RAM's unit number as entered in the ProDOS global page's device list will be $BF.

If you are going to use the auxiliary memory for any other purpose, you must protect yourself from /RAM.

If your use involves hi-res graphics, you may protect those areas of auxiliary memory. If you will save a "dummy" 8K file as the first entry in /RAM it will always be saved at $2000 to $3FFF. If you then immediately save a second "dummy" 8K file to /RAM it will be saved at $4000 to $5FFF. This technique provides a mechanism for protecting the hi-res pages in auxiliary memory while still maintaining /RAM as an online storage device.

There is no formula for determining where the blocks of /RAM physically reside in memory. Further, the logical blocks are not physically contiguous. There is no guaranteed way to protect any other fixed portions of auxiliary memory by the "dummy" file method.

If you wish to protect all of the auxiliary memory that has not been reserved for use by Apple, you must disconnect /RAM. To do this there are three areas of the system global page of interest:

$BF10-$BF2F contains the disk device driver addresses.

$BF31 contains the number of devices minus one.

$BF32-$BF3F contains the list of disk device numbers.

Here are the steps to be followed to disconnect /RAM:

0.) Suggested - Read block two on /RAM and take a peek at the file count field in the directory. If there are any files on /RAM, prompt the user to continue with the disconnect or abort the process.

1.) Check the MACHID byte at $BF96 to see if you are operating in a 128K environment. If not, there will be no /RAM to disconnect.

2.) The slot 0, drive 1 disk driver vector ($BF10) will point to the "No Device Connected" routine. The slot zero vectors $BF10 and $BF20 ARE RESERVED FOR OUR OWN USE. YOU CANNOT THEREFORE USE THESE VECTORS IF THIS CONVENTION IS TO WORK! If the slot 3 drive 2 vector also points to the same address, then /RAM is already disconnected.

3.) If we have determined that /RAM is on line, we are ready to remove it.

NOTE: If ProDOS has just been booted, /RAM is the last "disk" device installed. However, if the user has "manually" installed another device(s) the device number for /RAM will not be the last entry in the device list (DEVLST).

Also note that the following steps can be generically followed if you wish to disconnect ANY device.

a.) Retrieve the slot 3, drive 2 device number you find in DEVLST and save it.

b.) Move any remaining device numbers forward in the DEVLST.

c.) Retrieve the slot 3 drive 2 driver vector and save it for later re-installation.

d.) Replicate the "No Device Connected" vector in slot 0 drive 1 into slot 3 drive 2.

e.) Decrement the device count (DEVCNT).

/RAM is now disconnected and you are free to use the unreserved areas of auxiliary memory.

A convention has now been established for those ram disks with a capacity greater than 64K and wish not to be disconnected by programs that would not realize excess memory could still be utilized by the ram disk driver.

Here is what the routine might look like:

```
S    E   FILE #01 =>/P/INSTALLRAM
----- NEXT OBJECT FILE NAME IS /P/INSTALLRAM.0
1000:        1000    1           ORG   $1000
1000:        BF31    2 DEVCNT    EQU   $BF31          ; GLOBAL PAGE DEVICE COUNT
1000:        BF32    3 DEVLST    EQU   $BF32          ; GLOBAL PAGE DEVICE LIST
1000:        BF98    4 MACHID    EQU   $BF98          ; GLOBAL PAGE MACHINE ID BYTE
1000:        BF26    5 RAMSLOT   EQU   $BF26          ; SLOT 3, DRIVE 2 IS /RAM'S DRIVER VECTOR
1000:                6 *
1000:                7 * NODEV IS THE GLOBAL PAGE SLOT ZERO, DRIVE 1 DISK DRIVE VECTOR.
1000:                8 * IT IS RESERVED FOR USE AS THE "NO DEVICE CONNECTED" VECTOR.
1000:                9 *
1000:        BF10   10 NODEV     EQU   $BF10          ;
1000:               11 *
1000:               12 * FIRST THING TO DO IS TO SEE IF THERE IS A /RAM TO DISCONNECT!
1000:               13 *
1000:AD 98 BF       14           LDA   MACHID         ; LOAD THE MACHINE ID BYTE
1003:29 30          15           AND   #$30           ; TO CHECK FOR A 128K SYSTEM
1005:C9 30          16           CMP   #$30           ; IS IT 128K?
1007:D0 4D   1056   17           BNE   DONE           ; IF NOT, THEN BRANCH SINCE NO /RAM!
1009:               18 *
1009:AD 26 BF       19           LDA   RAMSLOT        ; IT IS 128K; IS A DEVICE THERE?
100C:CD 10 BF       20           CMP   NODEV          ; COMPARE WITH LOW BYTE OF NODEV
100F:D0 08   1019   21           BNE   CONT           ; BRANCH IF NOT EQUAL, DEVICE IS CONNECTED
1011:AD 27 BF       22           LDA   RAMSLOT+1      ; CHECK HI BYTE FOR MATCH
1014:CD 11 BF       23           CMP   NODEV+1        ; ARE WE CONNECTED?
1017:F0 3D   1056   24           BEQ   DONE           ; BRANCH, NO WORK TO DO; DEVICE NOT THERE!
1019:               25 *
1019:               26 * AT THIS POINT /RAM (OR SOME OTHER DEVICE) IS CONNECTED IN
1019:               27 * THE SLOT 3, DRIVE 2 VECTOR.  NOW WE MUST GO THRU THE DEVICE
1019:               28 * LIST AND FIND THE SLOT 3, DRIVE 2 UNIT NUMBER OF /RAM ($BF).
1019:               29 * THE ACTUAL UNIT NUMBERS, (THAT IS TO SAY 'DEVICES') THAT WILL
1019:               30 * BE REMOVED WILL BE $BF, $BB, $B7, $B3.  /RAM'S DEVICE NUMBER
1019:               31 * IS $BF.  THUS THIS CONVENTION WILL ALLOW OTHER DEVICES THAT
1019:               32 * DO NOT NECESSARILY RESEMBLE (OR IN FACT, ARE COMPLETELY DIFFERENT
1019:               33 * FROM) /RAM TO REMAIN INTACT IN THE SYSTEM.
1019:               34 *
1019:               35 *
1019:AC 31 BF       36 CONT      LDY   DEVCNT         ; GET THE NUMBER OF DEVICES ONLINE
101C:B9 32 BF       37 LOOP      LDA   DEVLST,Y       ; START LOOKING FOR /RAM OR FACSIMILE
101F:29 F3          38           AND   #$F3           ; LOOKING FOR $BF, $BB, $B7, $B3
1021:C9 B3          39           CMP   #$B3           ; IS DEVICE NUMBER IN ($BF,$BB,$B7,$B3)?
1023:F0 05   102A   40           BEQ   FOUND          ; BRANCH IF FOUND..
1025:88             41           DEY                   ; OTHERWISE CHECK OUT THE NEXT UNIT #.
1026:10 F4   101C   42           BPL   LOOP           ; BRANCH UNLESS YOU'VE RUN OUT OF UNITS.
1028:30 2C   1056   43           BMI   DONE           ; SINCE YOU HAVE RUN OUT OF UNITS TO
102A:B9 32 BF       44 FOUND     LDA   DEVLST,Y       ; GET THE ORIGINAL UNIT NUMBER BACK
102D:8D 59 10       45           STA   RAMUNITID      ; AND SAVE IT OFF FOR LATER RESTORATION.
1030:               46 *
1030:               47 * NOW WE MUST REMOVE THE UNIT FROM THE DEVICE LIST BY BUBBLING
1030:               48 * UP THE TRAILING UNITS.
1030:               49 *
1030:B9 33 BF       50 GETLOOP   LDA   DEVLST+1,Y     ; GET THE NEXT UNIT NUMBER
1033:99 32 BF       51           STA   DEVLST,Y       ; AND MOVE IT UP.
```

```
1036:F0 03   103B   52          BEQ   EXIT          ; BRANCH WHEN DONE(ZEROS TRAIL THE DEVLST)
1038:C8             53          INY                  ; CONTINUE TO THE NEXT UNIT NUMBER...
1039:D0 F5   1030   54          BNE   GETLOOP       ; BRANCH ALWAYS.
103B:               55 *
103B:AD 26 BF       56 EXIT     LDA   RAMSLOT       ; SAVE SLOT 3, DRIVE 2 DEVICE ADDRESS.
103E:8D 57 10       57          STA   ADDRESS       ; SAVE OFF LOW BYTE OF /RAM DRIVER ADDRESS
1041:AD 27 BF       58          LDA   RAMSLOT+1     ; SAVE OFF HI BYTE
1044:8D 58 10       59          STA   ADDRESS+1     ;
1047:               60 *
1047:AD 10 BF       61          LDA   NODEV         ; FINALLY COPY THE 'NO DEVICE CONNECTED'
104A:8D 26 BF       62          STA   RAMSLOT       ; INTO THE SLOT 3, DRIVE 2 VECTOR AND
104D:AD 11 BF       63          LDA   NODEV+1       ;
1050:8D 27 BF       64          STA   RAMSLOT+1     ;
1053:CE 31 BF       65          DEC   DEVCNT        ; DECREMENT THE DEVICE COUNT.
1056:60             66 DONE     RTS                  ; AND RETURN
1057:               67 *
1057:00 00          68 ADDRESS  DW    $0000         ; STORE THE DEVICE DRIVER ADDRESS HERE
1059:00             69 RAMUNITID DFB  $00           ; STORE THE DEVICE'S UNIT NUMBER HERE
105A:               70 *
```

Part of your exit procedure should include code to re-install /RAM so that it is available to the next application. Don't blindly reinstall /RAM...be sure it is off-line first. Applications should not begin by re-installing /RAM since this would preclude passing files from one application to the next in /RAM.

Here is the way to reinstall /RAM (or any general device):

a.) Re-install the device driver address you retrieved and saved as the slot 3 drive 2 vector.

b.) Increment the device count (DEVCNT).

c.) Re-install the device number in the device list (DEVLST).

NOTE: It may be best to re-install the device number as the first entry in the list. If the user has "manually" installed a disk driver, he may assume that since it was the last thing installed that it is still the last one in the list. Therefore, we recommend that you move all the entries in the list down one and re-install the /RAM device number as the first entry.

d.) Finally, set up the parameters for a format request and JSR to the device driver address you have re-installed. The /RAM driver will set up a "virgin" directory and bit map.

Here is what the reinstallation code might look like:

```
105A:                 *
105A:                 * THIS IS THE EXAMPLE /RAM INSTALL ROUTINE
105A:                 *
105A:AC 31 BF                LDY   DEVCNT        ; GET THE NUMBER OF DEVICES - 1.
105D:B9 32 BF         LOOP1  LDA   DEVLST,Y      ; LOAD THE UNIT NUMBER
1060:29 B0                   AND   #$B0          ; CHECK FOR SLOT 3, DRIVE 2 UNIT.
1062:C9 B0                   CMP   #$B0          ; IS IT THE SLOT 3, DRIVE 2 UNIT?
1064:F0 40   10A6            BEQ   DONE1         ; IF SO BRANCH.
1066:88                      DEY                  ; OTHERWISE SEARCH ON...
1067:10 F4   105D            BPL   LOOP1         ; LOOP UNTIL DEVLST SEARCH IS COMPLETED
1069:AD 57 10                LDA   ADDRESS       ; RESTORE THE DEVICE DRIVER ADDRESS
106C:8D 26 BF                STA   RAMSLOT       ; LOW BYTE..
106F:AD 58 10                LDA   ADDRESS+1     ; NOW THE
1072:8D 27 BF                STA   RAMSLOT+1     ; HI BYTE.
1075:EE 31 BF                INC   DEVCNT        ; AFTER INSTALLING DEVICE,INC DEVICE COUNT
1078:AC 31 BF                LDY   DEVCNT        ; USE Y FOR LOOP COUNTER..
107B:B9 31 BF         LOOP2  LDA   DEVLST-1,Y    ; BUBBLE DOWN THE ENTRIES IN DEVICE LIST
107E:99 32 BF                STA   DEVLST,Y      ;
1081:88                      DEY                  ; NEXT
1082:D0 F7   107B            BNE   LOOP2         ; LOOP UNTIL ALL ENTRIES MOVED DOWN.
1084:                 *
1084:                 * NOW SET UP A /RAM FORMAT REQUEST
1084:                 *
1084:A9 03                   LDA   #3            ; LOAD ACC WITH FORMAT REQUEST NUMBER.
1086:85 42                   STA   $42           ; STORE REQUEST NUMBER IN PROPER PLACE.
1088:                 *
1088:AD 59 10                LDA   RAMUNITID     ; RESTORE THE DEVICE
108B:8D 32 BF                STA   DEVLST        ; UNIT NUMBER IN THE DEVICE LIST
108E:29 F0                   AND   #$F0          ; STRIP THE DEVICE ID (ZERO LOW NIBBLE)
1090:85 43                   STA   $43           ; AND STORE THE UNIT NUMBER IN $43.
1092:                 *
1092:A9 00                   LDA   #$00          ; LOAD LOW BYTE OF BUFFER POINTER
1094:85 44                   STA   $44           ; AND STORE IT.
1096:A9 20                   LDA   #$20          ; LOAD HI BYTE OF BUFFER POINTER
1098:85 45                   STA   $45           ; AND STORE IT.
109A:                 *
109A:AD 8B C0                LDA   $C08B         ; READ & WRITE ENABLE
109D:AD 8B C0                LDA   $C08B         ; THE LANGUAGE CARD WITH BANK 1 ON.
10A0:                 *
10A0:                 * NOTE HOW THE DRIVER IS CALLED.  YOU JSR TO AN INDIRECT JMP SO
10A0:                 * CONTROL IS RETURNED BY THE DRIVER TO THE INSTRUCTION AFTER THE JSR.
10A0:                 *
10A0:20 A7 10                JSR   DRIVER        ; NOW LET DRIVER CARRY OUT CALL.
10A3:AD 82 C0                LDA   $C082         ; NOW PUT ROM BACK ON LINE.
10A6:60               DONE1  RTS                  ; THAT'S ALL.
10A7:                 *
10A7:6C 26 BF         DRIVER JMP   (RAMSLOT)     ; CALL THE /RAM DRIVER
```

The above routines address the specific case of /RAM. However, with a little massaging, they can easily be adapted to install or remove any disk driver routines.

The routines described in this document are examples only. No guarantee is made regarding their performance or suitability for any particular use.

ProDOS TECHNICAL NOTE #9

Buffer Management using BASIC.SYSTEM

(31 August 1983)

BASIC.SYSTEM provides buffer management for file I/O. Those facilities can also be utilized from machine language modules operating in the ProDOS/AppleSoft environment to provide protected areas for code, data, etc.

BASIC.SYSTEM resides from $9A00 upward with a general purpose buffer from $9600 (himem) to $99FF. When a file is opened, BASIC.SYSTEM does garbage collection, if needed, moves the general purpose buffer down to $9200 and installs a file I/O buffer at $9600. When a second file is opened, the general purpose buffer is moved down to $8E00 and a second file I/O buffer is installed at $9200. If an EXEC file is opened, it is always installed as the highest file I/O buffer at $9600, and all the other buffers are moved down. Additional regular file I/O buffers are installed by moving the general purpose buffer down and installing it below the lowest file I/O buffer. All file I/O buffers, including the general purpose buffer, are 1K (1024 bytes) and begin on a page boundary.

BASIC.SYSTEM may be called from machine language to allocate any number of pages (256 bytes) as a buffer, located above himem and protected from AppleSoft Basic programs. The ProDOS bit-map is not altered so that files may be BLOADed into the area without an error from the ProDOS kernel. If you subsequently alter the bit-map to protect the area, it is your responsibility to mark the area as free when you are finished...BASIC.SYSTEM will not do it for you.

To allocate a buffer, simply place the number of desired pages in the accumulator and JSR GETBUFR ($BEF5). If the carry flag returns clear, the allocation was successful and the accumulator will return the high byte of the buffer address. If the carry flag returns set, an error has occurred and the accumulator will return the error code. Note that the X and Y registers are not preserved.

The first buffer is installed as the highest buffer, just below BASIC.SYSTEM, from $99FF downward, regardless of the number and type of file I/O buffers that are open. If a second allocation is requested, it will be installed immediately below the first. Thus, it is possible to assemble code to run at known addresses...relocatable modules are not needed.

To deallocate the buffers created by the above call, it is only necessary to JSR FREEBUFR ($BEF8) and all of the buffers will be deallocated and the file buffers will be moved back up. It is important to note that although more than one buffer may be allocated by this call, they may not be selectively deallocated.

APPLE COMPUTER, Inc. PCS Developer Technical Support
20525 Mariani Avenue, M/S 22-W
Cupertino, CA 95014
Phone (408) 554-5213 or (408) 996-1010

ProDOS TECHNICAL NOTE #10

Installing Clock Driver Routines in ProDOS

(Revised 8 November 1983)

In you wish to support clock cards other than the ThunderClock, there are a number of possible places to locate your code. The "cleanest" place is to replace the ThunderClock routines located in ProDOS with your routines, if your code will fit.

When the PRODOS system file is executed, it installs the address of the ThunderClock routines at $BF07,$BF08 whether a card is present or not. The address is preceeded with a $4C (JMP) if a ThunderClock card is found or a $60 (RTS) if it was not.

The ThunderClock card is identified by looking at the $Cn00 ROM for:

$Cn00 = $08 $Cn02 = $28 $Cn04 = $58 $Cn06 = $70

If you look at $BF07,$BF08 you will find the location to put your code. There is room for 125 bytes.

To install your code, simply write enable the "language card" area, and move your code. Don't forget that your relocation code must justify the absolute addresses as part of the relocation procedure. Finally, restore any soft-switches you have changed. (There is no guarantee as to the absolute location of the clock driver code on future revisions of ProDOS, only that it's location may be found by examining the global page, as mentioned above.)

All that your code need do is get the time from the clock card, convert it to the ProDOS format and store it in the date and time locations in the global page.

Your installation routine can be called from an application or as part of the STARTUP program.

APPLE COMPUTER, Inc., PCS Developer Technical Support
20525 Mariani Avenue, M/S 22-W
Cupertino, CA 95014
Phone (408) 554-5213 or (408) 996-1010

ProDOS TECHNICAL NOTE #11

The ProDOS Machine Identification Byte

```
*   THIS NOTE SUPERCEDES THE INFORMATION   *
*  FOUND IN SECTIONS 5.2.3 & 5.3.1 OF THE  *
*    ProDOS TECHNICAL REFERENCE MANUAL     *
```

(revised 08 May 1984)

The Machine Identification byte (MACHID) in the ProDOS system global page has been redefined to permit identification of future products from Apple Computer, Inc. that may use the ProDOS operating system. The change does not impact any checking for existing systems that your application may now be doing.

The definition of MACHID at $BF98 is:

Bits 7-6	If bit 3 = 0 then
	00 =][
	01 =][+
	10 = //e
	11 = /// emulation
		If bit 3 = 1 then
		00 = reserved
		01 = reserved
		10 = //c
		11 = reserved

Bits 5-4	00 = reserved, 01 = 48K, 10 = 64K, 11 = 128K	
Bit 3	The value of bit 3 determines how bits 7-6 will be interpreted. See Bits 7-6 definition.	
Bit 2	Reserved for future definition	
Bit 1	0 = No 80-column card 1 = 80-column card installed	
Bit 0	0 = No ThunderClock or equivalent 1 = ThunderClock or equivalent installed	

APPLE COMPUTER, Inc., PCS Developer Technical Support
20525 Mariani Avenue, M/S 22-W
Cupertino, CA 95014
Phone (408) 996-1010

ProDOS TECHNICAL NOTE #12

Interrupt Handling

(1 December 1983)

This technical note expands upon the information found in the ProDOS Technical Reference Manual. It is assumed that the reader has already read and understands the sections regarding interrupts.

This tech note includes a superior example of an interrupt handler for use with ProDOS. The example in the book works properly, however, it will always claim every interrupt whether it came from the clock or not. Additionally, it does not conform to one protocol which will be required in future revisions of ProDOS, nor does it incorporate some common examples of good programming practice.

Vectors for interrupt handlers must be installed and removed with ALLOC_INTERRUPT and DEALLOC_INTERRUPT calls to ProDOS. Even though the vectors appear in the system global page, you must always use only the systems calls...never change the global page entries yourself.

All interrupt routines must commence with a CLD instruction. Although not checked in the initial release of ProDOS, this first byte will be checked in future revisions to verify the validity of the interrupt handler.

Good programming practice dictates that an interrupt handler should preserve the status register (PHP) and mask interrupts (SEI). The code should restore the status register (PLP) before exit, and before setting or clearing the carry flag as required by ProDOS.

If your application includes an interrupt handler, before you exit:

(1) Turn off the interrupts...remember, an unclaimed interrupt will cause system death.

(2) Make a DEALLOC_INTERRUPT call before exiting from your application. Don't leave a vector installed that will point to a routine that is gone.

Within your interrupt handler routines, you MUST leave ALL memory banks in the same configuration you found them. DON'T FORGET ANYTHING...main language card, main alternate $D000, main motherboard ROM...and, on an Apple //e...auxiliary language card, auxiliary alternate $D000, alternate zero page and stack, etc., etc... This is important! The ProDOS interrupt receiver assumes the environment is absolutely unaltered when your handler relinguishes control.

If your handler recognizes the interrupt and services it, the carry should be cleared (CLC) immediately before returning (RTS). If it was not your interrupt, the carry should be set (SFC) immediately before returning (RTS). Do not use a return from interrupt (RTI) to exit...the ProDOS interrupt receiver still has some housekeeping to perform before it issues the RTI instruction.

Here is a sample routine which will turn on interrupts on a ThunderClock card and print the date and time to the upper right corner of the screen.

```
0300:          0300   1            ORG $300
0300:          C20B   2 WTTCP      EQU $C20B     ; Clock write entry point (Slot 2)
0300:          C208   3 RDTCP      EQU $C208     ; Clock read entry point (Slot 2)
0300:          C080   4 TCICR      EQU $C080     ; Interrupt cont. register (Slot 2)
0300:          C088   5 TCMR       EQU $C088     ; Mystery register (Slot 2)
0300:                 6 *
0300:          0200   7 IN         EQU $200      ; Where the clock leaves the time
0300:                 8 *
0300:          0412   9 UPRIGHT    EQU $412      ; The upper right of the screen
0300:          047A  10 INTON1     EQU $47A      ; Leave interrupts on (Slot 2)
0300:          07FA  11 INTON2     EQU $7FA      ; Leave interrupts on (Slot 2)
0300:                12 *
0300:          BF00  13 MLI        EQU $BF00     ; Entry point to the ProDOS MLI
0300:                14 *
0300:                15 * CALLING INTERRUPTS, CALLING INTERRUPTS
0300:                16 *
0300:20 7E 03        17            JSR ALLOC.INT ; Install interrupt routine
0303:60              18            RTS           ; That's all forks
0304:                19 *
0304:                20 *
0304:          0304  21 SHOWTIME   EQU *
0304:D8              22            CLD
0305:08              23            PHP
0306:78              24            SEI           ; Disable Interrupts
0307:A0 20           25            LDY #$20      ; For slot 2
0309:B9 80 C0        26            LDA TCICR,Y   ; Get Interrupt Control Reg value
030C:29 20           27            AND #$20      ; Bit 5 indicates INT is clock
030E:F0 3C     034C  28            BEQ NOTCLK    ; If bit 5 is off, not from clock
0310:B9 88 C0        29            LDA TCMR,Y    ; Clear mystery register
0313:B9 80 C0        30            LDA TCICR,Y   ; Clear interrupt on hardware
0316:CE 4F 03        31            DEC COUNTER   ; Only print time every second
0319:D0 2E     0349  32            BNE EXITCLK   ; Not time to print yet
031B:                33 *
031B:A2 27           34            LDX #39       ; Save the input buffer
031D:BD 00 02        35 DOIN       LDA IN,X      ; Since the clock writes over it
0320:9D 56 03        36            STA INBUF,X   ; When it is called
0323:CA              37            DEX
0324:10 F7     031D  38            BPL DOIN
0326:                39 *
0326:A9 A5           40            LDA #$A5      ; Set Applesoft string input mode
0328:20 0B C2        41            JSR WTTCP     ; and send it to the card
032B:20 08 C2        42            JSR RDTCP     ; Read time into input buffer
032E:                43 *
032E:A2 15           44            LDX #21
0330:BD 01 02        45 GETNEXT    LDA IN+1,X    ; Print time to screen
0333:9D 12 04        46            STA UPRIGHT,X ; Chars 0-22 of input buffer
0336:CA              47            DEX
0337:10 F7     0330  48            BPL GETNEXT
0339:                49 *
0339:A9 40           50 SETCNTR    LDA #64       ; Set up counter for next time
033B:8D 4F 03        51            STA COUNTER
033E:                52 *
033E:A2 27           53            LDX #39       ; Restore the input buffer
0340:BD 56 03        54 DOIN2      LDA INBUF,X
0343:9D 00 02        55            STA IN,X
0346:CA              56            DEX
0347:10 F7     0340  57            BPL DOIN2
```

```
0349:                 *
0349:28               EXITCLK   PLP            ; Tell MLI we processed the INT
034A:18                         CLC
034B:60                         RTS
034C:28               NOTCLK    PLP
034D:38                         SEC            ; Tell MLI it isn't ours
034E:60                         RTS
034F:                 *
034F:         0001    COUNTER   DS  1,0
0350:                 *
0350:02 00            AIPARMS   DFB 2,0        ; Put allocate and deallocate
0352:04 03                      DW  SHOWTIME   ; Interrupt parameters here
0354:                 *
0354:01 00            DIPARMS   DFB 1,0        ; so both routines can use them
0356:                 *
0356:         0028    INBUF     DS  40,0       ; Save 40 bytes of IN here
037E:                 *                        ; for input buffer save/restore
037E:                 *
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
037E:20 00 BF         ALLOC.INT JSR MLI        ; Call the MLI
0381:40                         DFB $40        ; to allocate the interrupt
0382:50 03                      DW  AIPARMS
0384:D0 19    039F              BNE OOPS       ; Break on error
0386:                 *
0386:A0 20                      LDY #$20
0388:A9 AC                      LDA #$AC       ; Set 64hz interrupt rate
038A:20 0B C2                   JSR WTTCP      ; by writing a ',' to clock
038D:A9 40                      LDA #$40       ; Now enable the software
038F:8D 7A 04                   STA INTON1     ; and tell it not to disable
0392:8D FA 07                   STA INTON2     ; interrupts after reads
0395:99 80 C0                   STA TCICR,Y
0398:A9 01                      LDA #1         ; Print time immediately
039A:8D 4F 03                   STA COUNTER    ; Once per second later
039D:58                         CLI            ; Allow the 6502 to see the
039E:60                         RTS            ; interrupts
039F:                 *
039F:00               OOPS      BRK            ; Break on error
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
03A0:A9 00            DEALLOC.INT LDA #0       ; Disable interrupts
03A2:8D 7A 04                   STA INTON1     ; in the thunder clock
03A5:8D FA 07                   STA INTON2
03A8:A0 20                      LDY #$20
03AA:99 80 C0                   STA TCICR,Y
03AD:                 *
03AD:AD 51 03                   LDA AIPARMS+1  ; GET INT_NUM
03B0:8D 55 03                   STA DIPARMS+1  ; FOR DEALLOCATION
03B3:20 00 BF                   JSR MLI        ; CALL THE MLI
03B6:41                         DFB $41        ; TO DEALLOCATE THE INTERRUPT
03B7:54 03                      DW  DIPARMS    ; POINTER TO PARAMETER LIST
03B9:D0 01    03BC              BNE OOPS2      ; BREAK ON ERROR
03BB:60                         RTS            ; DONE
03BC:                 *
03BC:00               OOPS2     BRK            ; BREAK ON ERROR
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```


APPLE COMPUTER, Inc., PCS Developer Technical Support
20525 Mariani Avenue, M/S 22-W
Cupertino, CA 95014
Phone (408) 554-5213 or (408) 996-1010

ProDOS TECHNICAL NOTE #13

Double High Resolution Graphics Files

(6 January 1984)

The 128K Apple //e supports a graphics mode known as Double Hi-Res Graphics in which both main and auxiliary memory hi-res graphics pages are used to produce pictures with twice as many dot positions horizontally.

Apple /// graphics has a similar mode and a FOTOFILE file type ($08) has been defined under SOS to contain the screen image. All 16K double hi-res files under ProDOS should be of this file type.

The format of the file is as shown at the right. The "graphics mode" is stored in the 121st byte of the file (Location $78 in the file). The modes for both 1st and 2nd page of double hi-res are:

		Pg 1	Pg 2
280 X 192	Limited Color =	1	5
560 X 192	Black and White =	2	6
140 X 192	Full Color =	3	7

```
                    End-of-file
$3FFF|-------------------|
     |   Main Memory     |
     |  portion of file  |
$2000|                   |
     |-------------------|
$1FFF|                   |
     | Auxiliary  Memory |
     |  portion of file  |
$0000|-------------------|
     |  Beginning of file |
```

The normal Apple][hi-res 280 X 192 screen may be BSAVEd as usual. If you desire, for Apple /// SOS compatibility, you may also save these screens as an 8K type $08 FOTOFILE and mark the graphics mode as zero (page 1) or four (page 2), (Apple /// 280 X 192 Black and White mode).

APPLE COMPUTER, Inc., PCS Developer Technical Support
20525 Mariani Avenue, M/S 22-W
Cupertino, CA 95014
Phone (408) 554-5213 or (408) 996-1010

ProDOS TECHNICAL NOTE #14

Selector/Dispatcher Conventions

(revised 09 March 1984)

ProDOS MLI call $65, the QUIT call moves addresses $D100 - $D3FF from the second 4K-byte bank of RAM of the language card to $1000 and executes a JMP to $1000. What initially resides in that area is OUR dispatcher code.

The dispatcher once executed does the following:

1. Interactively allows you to enter a prefix and file name of the system program (interpreter) that you wish to execute.

2. Stores the system program name at $280 starting with a length byte. This is done so once the system program executes, it can find from where is was started and locate any files it could need for processing.

3. Closes any open files.

4. Clears the bit map and protects the zero, stack, text and ProDOS Global pages.

5. Reads in the system file at $2000 and executes a JMP to $2000.

If you wish, you can install your own QUIT code which may load in your own full blown selector program. If you choose to do this, you must at some point:

1. Follow steps 2 - 4 above.

2. THE $D100 BYTE MUST BE A CLD ($D8) INSTRUCTION. This convention is established so programs will be able to tell whether it is selector code or the ProDOS dispatcher code that is resident.

In addition to just leaving the pathname at $280 for the interpreters own use, a method to enable a selector program to specify an accomanying 'STARTUP' program has been defined. Once active, an interpreter can immediately run that program.

The procedure will be to reserve an area in the system file which will be overwritten by a selector program with the 'STARTUP' programs name. The interpreter would then load and execute that specified program.

The actual nuts and bolts of this procedure are as follows:

The selector program will look at the first byte of the interpreter at $2000. If it is a JMP ($4C) instruction, and bytes $2003 and $2004 are both $EE's, then byte $2005 will be interpreted as a buffer size indicator with the buffer starting at $2006. The string at $2006 would be the normal ProDOS pathname or partial pathname starting with a length byte.

JMP CONT	$2000-$2002
$EE | $EE	$2003-$2004
$41 (eg.)	$2005
$07	$2006
STARTUP	$2007-$200D
:	
CONT: (eg.)	$2047

The two $EE's serve as a marker to the selector program to let it know that this particular interpreter can run a startup program. The interpreters that will support this feature will of course supply their own default string which may be a startup program or a flag of your own choice.

For more information on Interpreter Conventions please see ProDOS Technical Note #7.

APPLE COMPUTER INC., PCS Developer Technical Support
20525 Mariani Avenue, M/S 22-W
Cupertino, CA 95014
Phone: (408) 996-1010

APPLE COMPUTER, INC.
20525 Mariani Avenue
Cupertino, CA 95014

PASCAL TECHNICAL NOTE #10

Configuration and Use of The Apple II Pascal 1.2 Runtime Systems

(December 1983)

For further information contact:
PCS Developer Technical Support
M/S 22-W, Phone (408) 996-1010

Disclaimer of All Warranties and Liabilities

Notice

APPLE COMPUTER, INC.
20525 Mariani Avenue
Cupertino, CA 95014

PASCAL TECHNICAL NOTE #10

Configuration and Use of The Apple II Pascal 1.2 Runtime Systems

(December 1983)

For further information contact:
PCS Developer Technical Support
M/S 22-W, Phone (408) 996-1010

I. INTRODUCTION

The Apple II Pascal 1.2 Runtime Systems permit the "turnkey" execution of application software that has been developed using Apple Pascal. This Technical Note is intended to aid Vendors and applications developers who are familiar with the Apple II Pascal 1.2 Development System. Those who are not should read carefully the following documents:

* Apple Pascal Operating System Reference Manual (with addendum)
* Apple Pascal Language Reference Manual (with addendum)
* Apple II Pascal 1.2 Update Manual

II. SYSTEM OVERVIEW

The Runtime Systems support only the execution of an application package. Unlike the Pascal Development System, the Runtime Systems do not contain the Assembler, Compiler, Editor, Filer or Linker, nor even an error reporting mechanism at the system level. System operations such as transferring files, disk compacting ("Krunching"), and the reporting of and recovery from errors, are all left to the application program. Clearly, it is the software developer's responsibility to design and implement "friendly," entirely self-contained packages for use with the Runtime Systems. The safest assumption to make when developing such packages is that the end-user is not only unfamiliar with the facilities of the Pascal Development System, but may also be ignorant of computer operation and use in general.

The three runtime systems currently available are :

* The 48K Runtime System (standard and stripped versions)
* The 64K Runtime System (standard version only)
* The 128K Runtime System (standard version only)

The name of each runtime system indicates the minimum amount of RAM necessary for proper operation. Any additional RAM available above the minimum will not be used by the Runtime Systems.

There are two versions of the 48K Runtime System available, one of which provides more free memory for the application package's programs and data than does the other. Except as noted later, the "standard" configuration of the Runtime System supports all features of the Pascal Development System that are relevant to turnkey execution of applications software. The "stripped" configuration lacks set operations and floating-point arithmetic.

III. CONTENTS OF APPLE II PASCAL 1.2 RUNTIME DISKETTES

The following files are contained on "RT48:", the Apple II Pascal 1.2 48K Runtime System diskette:

* RTSTND.APPLE (29 blocks) — 48K Runtime "standard" P-machine.
* RTSTRP.APPLE (24 blocks) — 48K Runtime "stripped" P-machine.
* SYSTEM.PASCAL (28 blocks) — 48K Runtime operating system.
* SYSTEM.LIBRARY (39 blocks) — Contains the same Intrinsic Units as described in the Apple Pascal Language Reference Manual. However, these Units are for use only with the Runtime System, and will not execute properly in the development environment. Conversely, only Units in this library, NOT those on the 1.2 Development System diskettes, should be used when executing programs in the Runtime environment. Note that the developer is, however, free to add his own Intrinsic Units to the Runtime SYSTEM.LIBRARY.
* SYSTEM.ATTACH (9 blocks) — A runtime version of the dynamic driver-attachment program described in the Apple II Pascal Attach Tools manual. This version may only be used with the Runtime Systems.
* RTSETMODE.CODE (4 blocks) — Utility program that permits Vendor to arm or disarm any or all of four configuration options: "Filehandler Overlay", "Single Drive System", "Ignore External Terminal" and "Get/Put and Filehandler Overlay".
* RTBOOTLOAD.CODE (4 blocks) — Utility program to load 48K Runtime bootstrap code onto blocks 0 and 1 of Vendor Product Diskette.
* RTBSTND.BOOT (4 blocks) — Contains bootstrap code for RTSTND.APPLE.
* RTBSTRP.BOOT (4 blocks) — Contains bootstrap code for RTSTRP.APPLE.
* II40.MISCINFO (1 block) — Miscinfo file optomized for a 40-column Apple II or Apple II Plus. Identical to that supplied with the Development System.
* II80.MISCINFO (1 block) — Miscinfo file optomized for an 80-column Apple II or Apple II Plus . Identical to that supplied with the Development System.
* IIE40.MISCINFO (1 block) — Miscinfo file optomized for a 40-column Apple //e. Identical to that supplied with the Development System.
* SYSTEM.MISCINFO (1 block) — Miscinfo file optomized for an 80-column Apple //e. Identical to that supplied with the Development System.
* SYSTEM.CHARSET (2 blocks) — Identical to that supplied with the Development System, it is included here only for redundancy's sake. SYSTEM.CHARSET is needed on the Vendor Product Diskette only if TURTLEGRAPHICS is used.

The following files are contained on "RT64:", the Apple II Pascal 1.2 64K Runtime System diskette:

* SYSTEM.APPLE (32 blocks) — 64K Runtime "standard" P-machine.
* SYSTEM.PASCAL (29 blocks) — 64K Runtime operating system.

* SYSTEM.LIBRARY
* SYSTEM.ATTACH
* RTSETMODE.CODE
* II40.MISCINFO
* II80.MISCINFO
* IIE40.MISCINFO
* SYSTEM.MISCINFO
* SYSTEM.CHARSET

——> same files as 48K Runtime System

The following files are contained on "RT128:", the Apple II Pascal 1.2 128K Runtime System diskette:

* SYSTEM.APPLE (32 blocks) — 128K Runtime "standard" P-machine.
* SYSTEM.PASCAL (29 blocks) — 128K Runtime operating system.

* SYSTEM.LIBRARY
* SYSTEM.ATTACH
* RTSETMODE.CODE
* SYSTEM.MISCINFO
* SYSTEM.CHARSET

——> same files as 48K Runtime System

Of these files, the final Vendor Product Diskette should contain only the Runtime P-machine (RTSTND.APPLE, RTSTRP.APPLE, or SYSTEM.APPLE), SYSTEM.PASCAL, SYSTEM.LIBRARY, the appropriate miscinfo file renamed to SYSTEM.MISCINFO, and, optionally, SYSTEM.CHARSET. Information on the different miscinfo files is contained in the Apple II Pascal 1.2 Update Manual. SYSTEM.ATTACH, with its attendant data files as described in the Apple II Pascal Attach Tools manual, should be included on the Vendor Product Diskette if and only if special device drivers, written in machine-code, must be bound into the system for use by the Applications Package. All other files on the Runtime System diskettes are used in creating and configuring the Vendor Product Diskette.

IV. OPERATION

The term "Vendor Product Diskette," as used throughout this Technical Note, refers to the primary (boot) diskette in a turnkey application package, which is assumed to contain the following software: the Runtime P-machine , the Runtime Operating system, a SYSTEM.LIBRARY file, a SYSTEM.MISCINFO file, and the files comprising the applications package's programs (and any necessary data). In most instances, the Vendor Product Diskette will be the only software diskette in the package. Larger systems, however, may also include other diskettes that contain additional software and data which will not fit on the bootstrap diskette.

Note that the main application program must be named SYSTEM.STARTUP, so that the Runtime System can find it at bootstrap-load time.

A two-stage boot process can be used with the 64K and 128K Runtime Systems if the necessary boot files listed above cannot fit on a single diskette. In this case, the primary boot diskette would contain only the Runtime P-machine. A second-stage boot diskette would contain the remainder of the files. A two-stage boot process cannot be used with the 48K Runtime System.

A. The Bootstrapping Process

In a machine equipped with an auto-start ROM, the bootstrap loading process occurs automatically, as soon as the Apple's main power switch is turned "ON." As a result, the end-user is greeted by the applications package. In a machine that lacks an auto-start ROM, the end-user first encounters the Apple MONITOR, or BASIC, and must initiate the bootstrapping process by issuing a 6-CTRL-P command (in the case of the MONITOR) or a PR#6 command (for BASIC).

The bootstrap loader checks for the P-machine file and loads it into RAM. The P-machine, in turn, brings in and initializes the Runtime operating system. (In the case of a two-stage boot, the message "Insert boot diskette with SYSTEM.PASCAL on it, then press RETURN" appears after the P-machine has been loaded. The end-user should then insert the second-stage boot diskette and press RETURN which results in the Runtime operating system being loaded and initialized.) The first noteworthy action taken by the operating system is to execute SYSTEM.ATTACH, if that utility program is available on the Vendor Product Diskette. Remember that SYSTEM.ATTACH must not be present on the Vendor Product Diskette unless special, low-level I/O drivers must be bound into the system. As explained more fully in the Apple II Pascal Attach Tools manual, SYSTEM.ATTACH uses two special data files, and will fail if these files are not present on the bootstrap diskette. A vendor who puts SYSTEM.ATTACH on his Vendor Product Diskette without also providing the data files required by that program insures consistent failure of the system bootstrap process. The vendor may include the SYSTEM.ATTACH software on the Vendor Product Diskette, while defeating the automatic execution of that utility at bootstrap load time, by changing its name in the diskette directory.

The bootstrap process culminates when the main applications program, SYSTEM.STARTUP, is loaded and executed. Any failure during the bootstrap process is fatal. Whenever possible, a failure will leave displayed the message

SYSTEM FAILURE NUMBER nn. PLEASE REFER TO PRODUCT MANUAL.

Here, "nn" refers to the actual number reported when the failure occurs. This number will correspond to one of the following failures:

01 Unable to load specified program
02 Specified program file not available
03 Specified program file is not code file
04 Unable to read block zero of specified file
05 Specified code file is un-linked
06 Conflict between user and intrinsic segments
07 UNASSIGNED ERROR CODE
08 Required intrinsics not available
09 System internal inconsistency
10 Can't load required intrinsics/Can't open library file
11 Specified code file must be run under the 128K system
12 Original disk not in boot drive

Clearly, these messages are useful as debugging tools as well as in mechanisms for field failure-reporting. The "PRODUCT MANUAL" mentioned in the bootstrap failure message is, of course, the vendor's own product manual. It is the responsibility of the vendor to ennumerate and explain for the end-user the situations in which bootstrap failures may occur, as well as suggest remedies for these failures.

B. General Considerations

Once the program is loaded and running, operation proceeds normally, and may even include removal of the system disk. (It is, however, the responsibility of the application package to protect itself against the possibility that the system disk will not be on-line when a segment must be overlaid, or a specific subprogram must be chained to. At such times, the application software should first determine whether or not the required disk is on-line, and, if not, suspend operation, after giving a suitable prompt, until the user has inserted the disk in the appropriate drive.) Any errors that occur during execution of the applications package cause the system to transfer program control to a specific procedure in the currently-executing application program, where code intended to respond to errors is assumed to exist. If any program in the applications system terminates without chaining to another one, the Runtime system re-boots into SYSTEM.STARTUP.

VI. SPECIFICATIONS

A. Available Configurations

The memory requirements of different applications impose the need for different Runtime Systems. The applications developer should choose one of the systems as the target environment, and keep its limitations and capabilities in mind during design and implementation of the applications package. Apple currently supports the following Runtime Systems:

* 48K Runtime System (standard and stripped versions)
* 64K Runtime System (standard version only)
* 128K Runtime System (standard version only)

The difference between the standard and stripped versions of the 48K

Runtime System is that the stripped version does not support set operations or floating point arithmetic thereby making more memory available for the application.

The chart below summarizes the amount of free memory that is available under the different Runtime Systems for use by the application package. Note that when swapping is set to level 1 the amount of memory available to the application package is increased by 3668 bytes.

FREE MEMORY IN APPLE II PASCAL 1.2 RUNTIME SYSTEMS

	NO SWAPPING	SWAPPING ON LEVEL 1
48K STANDARD	23372 bytes	27040 bytes
48K STRIPPED	25676 bytes	29344 bytes
64K	40322 bytes	43990 bytes
128K (CODE)	41227 bytes	44879 bytes
128K (DATA)	44502 bytes	44526 bytes

NOTE - the amount of free memory available with the 64K Runtime System is reduced by 1024 bytes if it is operating in 40-column mode.

There is another level of swapping (level 2) which provides an additional 822 bytes of usable memory, however, application writers should not depend on the extra memory being available in the future. Certain planned enhancements to the Pascal system will reduce the memory available to applications by approximately 1000 bytes. Swapping level 2 will help programs currently running at the limit of available memory to run under the enhanced system.

NOTE - using GET or PUT to disk will be slow if swapping level 2 is selected since these routines will have to be loaded repeatedly. READ and WRITE to disk will also be slow since they use GET and PUT. BLOCKREAD, BLOCKWRITE, UNITREAD, and UNITWRITE will be unaffected.

Swapping can be set to the desired level by using RTSETMODE (described later) or by calling a procedure in CHAINSTUFF before chaining to another subprogram. See the Apple II Pascal 1.2 Update Manual for further information on swapping.

B. Use Environment

The hardware environment must include the following:

48K Runtime System	- An Apple II or II Plus with 48K of RAM (minimum), or an Apple //e
64K Runtime System	- An Apple II or II Plus with 48K of RAM and an Apple Language Card, or an Apple //e
128K Runtime System	- An Apple //e with an Extended 80-column Text Card
All Runtime Systems	- At least one disk drive, set up for 16-sector operation.
All Runtime Systems	- Video screen or external terminal (video screen preferred).

Note that the Runtime Systems support all Apple peripheral cards. Other cards may not operate properly, especially if they include firmware that depends upon specific internal characteristics of the P-machine interpreter or operating system. SYSTEM.ATTACH must be used by those Vendors who wish to reconfigure the BIOS (Basic I/O Subsystem) to support non-standard peripheral devices. Through the ATTACH facility, it is possible to assign new physical devices to any of the existing logical I/O units in the Pascal system, as well as retain the standard device assignments while adding new devices to the system. Drivers prepared for use with SYSTEM.ATTACH are bound into the system dynamically, at each and every boostrap load. Note that the addition of special I/O drivers to the system will further restrict the amount of free memory available for use by the applications code, since drivers are loaded on the Pascal system heap. For more information, see the Apple II Pascal Attach Tools manual.

C. Restrictions and Considerations

1. SYSTEM.ATTACH and the CHAINSTUFF, LONGINTIO, and PASCALIO units in SYSTEM.LIBRARY make assumptions about the internal structure of the Pascal operating system. Because the internals of the Runtime operating systems are different from those in the Development System, only the versions of CHAINSTUFF, LONGINTIO, PASCALIO and SYSTEM.ATTACH that are supplied on the Runtime System diskettes should be used in the Runtime execution environment. (Furthermore, these special versions should never be used in the Development environment!)

2. The units TRANSCEND and TURTLEGRAPHICS employ floating-point operations, so software intended to be executed under the 48K Stripped Runtime System should not use them. For software that employs the TURTLEGRAPHICS procedure TURNTO, note that turns through right-angles and null-angles are treated as special cases, and the TURTLEGRAPHICS unit uses only integer arithmetic in calculating the trigonometric values needed to execute them. So, TURTLEGRAPHICS may be used under the 48K Stripped Runtime System if and only if the turtle is allowed to make only right-angle turns (as in the HILBERT demonstration program on APPLE3:, for example). Attempts to draw arbitrary curves, as demonstrated in the GRAFDEMO program on APPLE3:, will produce execution errors in the 48K Stripped Runtime environment.

3. Pascal's special function keys retain their meanings in the Runtime Systems. The following keys have special meaning:

   * Freeze (Stop) screen display - CTRL-S
   * Flush screen display - CTRL-F
   * Switch to alternate half of screen - CTRL-A
   * Toggle display to switch screen halves to follow cursor - CTRL-Z
   * Left square bracket - CTRL-K
   * Right square bracket - SHIFT-M
   * Break - CTRL- @
   * Upper/lower case activation toggles - CTRL-W, CTRL-E

   NOTE - Some of these special function keys are ignored by Pascal if it is running on a //e. See the Apple II Pascal 1.2 Update Manual for more information. It is possible to disable some of these special key functions. See the Apple II Pascal Attach Tools manual for complete details.

4. The Runtime System will operate correctly only with programs that have been prepared, using Apple's Pascal compiler and/or Pascal-system assembler on either an Apple II or an Apple ///, for execution in the Apple II Pascal environment.

5. The Runtime System is optimized for operation with the Apple's built-in video output screen. There is no easy way for a turnkey package to reconfigure its host Runtime System to use the random-cursor facilities of any arbitrary external terminal. Therefore, it is expected that users of the system will be operating with the standard Apple video screen, and not an external terminal. Any program that makes use of screen control, such as clearscreen, random cursor addressing, or backspacing, is not likely to work properly on an external terminal. To avoid this problem, the Runtime System contains a switch which can be set through the RTSETMODE program (explained below). When set, this switch causes the system to ignore an external terminal, if one is connected. Simple programs that do not make use of any screen control may leave the external terminal switched in without any adverse consequences.

D. Runtime System Configuration Utilities

1. RTSETMODE (provided with all Runtime Systems)

Flags which note the state of four system options are contained within a special part of the directory of any Runtime System bootstrap diskette. (These flags will not normally be present on diskettes prepared for or used with the Pascal Development System.) When a flag is set (TRUE), the corresponding system option is enabled. The option is disabled when the corresponding flag is reset (FALSE). At bootstrap time, the option-flags are retrieved and are used during a dynamic configuration process which occurs before the applications software is executed.

The RTSETMODE utility is used by the applications developer to set or reset the option-flags, according to the requirements of the applications package. In operating RTSETMODE, the developer first selects the Pascal volume to be affected, then answers four yes-or-no questions by pressing the "Y" or "N" keys, respectively. Responding to any prompt for input by pressing only the RETURN key causes immediate termination of the program.

Answering "Y" to any of the following questions ARMS the indicated option (setting the corresponding flag), while answering "N" DISARMS the option (and resets the corresponding flag).

* ARM Filehandler Overlay Option? - Arming this option sets swapping to level 1. System primitives related to disk file opening and closing are overlaid as needed by the application software, thus freeing 3668 bytes of RAM for use by the application.

* ARM Single-Drive System Option? - With this option armed, once the initial bootstrap process is finished at the beginning of any turnkey software run, the system itself will not assume the availability of any disk drives other than the bootstrap device. Specifically, "volume searches" will be limited to the single drive. The application may still use Apple Pascal's UNITREAD and UNITWRITE procedures to access any other drives which may be connected to the system.

* ARM Ignore External Terminal Option? - Arming this option insures that the system CONSOLE: device will always be the Apple's built-in video screen, whether or not an external terminal interface or 80-column card is available in slot 3.

* ARM Get/Put and Filehandler Overlay Option? - Arming this option sets swapping to level 2. System primitives related to disk file opening and closing, as well as GET and PUT to disk are overlaid as needed. (See section A for more information on swapping level 2.)

After the four-question sequence, RTSETMODE asks the user to confirm that all information input to that point is correct and should be used to update the Vendor Product Diskette. If so, an attempt is made to update the diskette's directory with the new set of option flags, and RTSETMODE finishes by reporting the success or failure of the update operation.

Developers should note that only exact copies of a Runtime bootstrap diskette will retain its option-flags. Transferring the Runtime System and applications software from diskette to diskette on a file-by-file basis will not also transfer the option-flags between the diskettes. For this reason, it is recommended that RTSETMODE be applied to the product master of any Runtime-based package immediately prior to releasing that master to production, in order to insure the correct status of the option-flags.

If a two-stage boot will be used for a runtime application, RTSETMODE must be run on both boot diskettes since some of the flags are checked by the P-machine while others are checked by the operating system.

2. RTBOOTLOAD (48K Runtime System only)

This program is used to transfer to the Vendor Product Diskette the proper bootstrap code for the chosen 48K Runtime configuration (STND or STRP). Responding to any prompt for input by pressing only the RETURN key results in immediate termination of the program. RTBOOTLOAD first asks for the name of the file which contains the appropriate bootstrap code (either RTBSTND.BOOT or RTBSTRP.BOOT). The file name must be entered exactly as it appears in the directory (including a volume prefix if the file is not on the default volume), or the program will not be able to find the file, and will repeat its request for a file name. Once it has fetched the bootstrap code, RTBOOTLOAD asks for the volume name of the Vendor Product Diskette, then waits for the user to press the SPACE-BAR (thus providing the user with an opportunity to mount the selected volume, if necessary) before attempting to transfer the bootstrap information. The success or failure of the transfer is reported before RTBOOTLOAD terminates. This program is only supplied on the 48K Runtime System diskette and should never be used to transfer bootstrap information to a diskette which contains the 64K or 128K Runtime Systems, as doing so will prevent the systems from booting correctly.

E. Error Handling

If an error in execution or I/O occurs during program operation, the Runtime System attempts to let the application package itself acknowledge, and if possible, recover from the error condition. Just as he may in the Pascal Development environment, the application developer is free to use the $I- and $R- compiler options to assume localized, programmatic control of the corresponding error situations.

When the Runtime System detects an error, it stores the error number in IORESULT and calls "PROCEDURE NUMBER TWO" of the currently-executing program. This is the procedure in segment number 1 that has been given the procedure number 2 by the compiler. In other words, it is the first one declared after the program heading that is not itself a unit or segment procedure, or within a unit or segment procedure. In a compiler listing, "PROCEDURE NUMBER TWO" may be identified as those lines whose "S" (segment) number is 1, and whose "P" (procedure) number is 2.

"PROCEDURE NUMBER TWO" may be declared as a forward procedure since the procedure number is assigned at the forward declaration.

From now on, "PROCEDURE NUMBER TWO" will usually be called the "Error Handler," since it must always be reserved by the applications programmer for the sole purpose of handling errors. The Error Handler may not have any parameters, and must always be declared as a PROCEDURE, never as a

FUNCTION.

The Error Handler can determine what kind of error has occurred by checking the value of the IORESULT function. In the Development System, this function is restricted to containing the codes for any I/O errors that might occur during execution. In the Runtime Systems, IORESULT has been extended to report all system errors, as well as the usual I/O errors.

Here are all the values IORESULT can assume during Runtime execution:

00 No error	100 Unknown Runtime error
01 Bad block, parity error	101 Value range error
02 Bad I/O unit number	102 No procedure in segment table (*)
03 Illegal I/O request	103 Exit from uncalled procedure (*)
04 Data-com timeout	104 Stack overflow (*)
05 Volume went off-line	105 Integer overflow
06 File lost in directory	106 Divide by zero
07 Bad file name	107 Nil pointer reference
08 No room on volume	108 Program interrupted by user
09 Volume not found	109 System I/O error
10 File not found	110 User I/O error
11 Duplicate directory entry	111 Unimplemented instruction
12 File already open	112 Floating point error
13 File not open	113 String overflow
14 Bad input format	114 Programmed HALT
16 Disk is write-protected	115 Programmed breakpoint
17 Illegal block number	116 Codespace overflow
18 Illegal buffer address	
19 Must read a multiple of 512 bytes	
20 Unknown ProFile error	
64 Device error (bad disk format)	

* = fatal error

It is recommended that a program's Error Handler should simply report "system error" for all cases except those which are relevant to the program. Global state variables in the program may be used to help determine the nature of the problem and report it to the user. Note that a system re-boot occurs if an attempt is made to exit the program (without chaining to another).

After the Error Handler finishes its operation, control returns to the caller of the procedure where the error occurred (unless the error was fatal). In this way, program operation may be continued, cleanly and simply, after an error is handled. The caller of a failure-prone procedure can set and test status flags to determine whether or not the called procedure completed its operation, and either repeat the procedure call, or perform an alternative action.

In developing particularly large systems where program chaining is used, the applications programmer should remember that each chained program must reserve "PROCEDURE NUMBER TWO" as an Error Handler.

Following are two programming examples. The first shows a typical

Error Handler routine, and the second is a program fragment that demonstrates an error recovery technique.

```
(* EXAMPLE #1 -- ERROR HANDLER *)

   (* THE FOLLOWING PROCEDURE IS ONLY *)
   (* CALLED BY THE OPERATING SYSTEM *)

   PROCEDURE ErrorHandler;

     PROCEDURE Message(Space: Boolean; S: String);
     VAR Ch : Char;
     BEGIN (* Message *)
       WriteLn;
       WriteLn('*** ',S);
       IF Space THEN
         BEGIN
           Write('*** Press SPACE-BAR to continue');
           REPEAT
             Read(Keyboard, Ch)
           UNTIL ((Ch = ' ') AND (NOT EoLn));
         END;
     END   (* Message *);

   BEGIN (* ErrorHandler *)
     IF (IOResult = 14) THEN
       Message(True,'That is not a legal integer!')
     ELSE IF (IOResult = 106) THEN
       Message(True,'Division by zero is impossible!')
     ELSE BEGIN
       Message(False,'System error.  Please reboot.');
       WHILE True DO (* Hang *);
     END;
   END   (* ErrorHandler *);

   (* END OF EXAMPLE #1 *)


   (* EXAMPLE #2 -- ERROR RECOVERY USING ERROR HANDLER OF EXAMPLE #1 *)

   PROCEDURE Calculator;
   (* Features recovery from input or arithmetic error. *)
   TYPE Order = (First, Second);
   VAR A,B : Integer;
       Flag : Boolean;

     PROCEDURE GetNumber(Which: Order; VAR Number: Integer);
     BEGIN
       Write('Input the');
       IF (Which = First) THEN
         Write(' first')
       ELSE Write(' second');
       Write(' number: ');
```

```
      Read(Number);  ReadLn;
      Flag := True;
    END   (* GetNumber *);

    PROCEDURE Answer;
    VAR R : Real;
    BEGIN
      R := A / B; (* Bombs if B=0 *)
      WriteLn;
      WriteLn(A,' divided by ',B,' is ',R);
    END   (* Answer *);

  BEGIN (* Calculator *)
    REPEAT
      Flag := False;
      WriteLn;
      WriteLn;
      REPEAT
        GetNumber(First,A)
      UNTIL Flag;
      Flag := False;
      WriteLn;
      REPEAT
        GetNumber(Second,B)
      UNTIL Flag;
      Answer;
    UNTIL Eof;
  END   (* Calculator *);

  (* END EXAMPLE #2 *)
```

To illustrate the effect of the Runtime System's error handling mechanism, here is the interaction between user and machine during a typical run of the above "Calculator" program. User-input is terminated by a press of the <RETURN> key in all cases except the first and last. In the first case, the Error Handler is invoked during the erroneous numeric input. In the last case, the system accepts and acts upon a <CONTROL-C> signal before the user has a chance to press any other keys.

```
Input the first number: N

*** That is not a legal integer!

Input the first number: 16

Input the second number: 0

*** Division by zero is impossible!

Input the first number: 16

Input the second number: 2
```

16 divided by 2 is 8

Input the first number: <CONTROL-C>

As soon as the user presses <CONTROL-C>, the Runtime system detects the end of the standard input file (EOF), and re-boots (right back into "Calculator").

V. DIFFERENCES BETWEEN THE PASCAL DEVELOPMENT SYSTEM AND THE RUNTIME SYSTEMS

Although the Runtime Systems will run most Pascal code files exactly as does the Pascal Development System, the applications system developer must be aware of important differences between the two environments. As mentioned above, there is no "system-level" handling of any type of error that may occur, including stack overflow, arithmetic errors, or bad disk reads. It is left to the application package to respond to all error conditions. The typical user will not have access to (nor knowledge of) the Pascal Formatter or Filer.

Many programs which fit comfortably in the 64K Development System environment may fail to execute at all under the 48K Runtime System due to the difference in available user memory. Similarly, programs developed with the 128K Development System may fail to execute under the 64K Runtime System for the same reason. While large systems can be made to fit within the confines of a particular Runtime environment, this is possible only through use of Apple Pascal's program segmentation (overlay) and chaining facilities. It is suggested, however, that much thought and care be taken when using chaining and segmentation in software design, since these facilities, by their very nature, involve time-consuming disk accesses. Application software that abuses chaining and/or segmentation, or employs them in a careless fashion, may easily waste a large amount of time in "disk thrashing," especially if swapping is being used. Finally, an applications package runs the risk of massive failure unless calls to program overlays and chaining are preceded by checks that the expected diskette is in the appropriate drive. This is especially important when the target machine includes only one disk drive (as is frequently the case).

The following items are never present in the Runtime Systems:

* System HOMECURSOR, CLEARSCREEN, and CLEARLINE functions
* System prompt function
* Compiler, Assembler, Linker, Editor, Filer
* IDSEARCH and TREESEARCH procedures (which exist in the Development System only to benefit the Compiler).

Programs that make use of information stored in specific memory locations within the 1.2 Development System P-machine, or that make assumptions about static or dynamic memory allocation at the operating system level (e.g., for the purpose of accessing system data structures) are likely to function incorrectly when executed in the Runtime environment. This is due to the code reorganization, compaction, and optimization that was necessary to produce the Runtime Systems.

VII. CREATION OF VENDOR PRODUCT DISKETTE

The following steps can be used as a guide for creating a Vendor Product Diskette:

1 - Format a diskette using the Pascal Development System formatter.

2 - Transfer the files SYSTEM.APPLE (or RTSTND.APPLE or RTSTRP.APPLE), SYSTEM.PASCAL, SYSTEM.LIBRARY, SYSTEM.MISCINFO, and SYSTEM.CHARSET (if needed) from the Runtime System diskette to the Vendor Product diskette.

3 - Transfer the code file(s) for the application to the Vendor Product diskette. The main code file for the application must be named SYSTEM.STARTUP.

4 - Run the Pascal Development System library program to add any needed library units to SYSTEM.LIBRARY on the Vendor Product diskette.

5 - Run RTBOOTLOAD to load the appropriate bootstrap code from RT48: onto the Vendor Product diskette. (48K RUNTIME SYSTEMS ONLY)

6 - Run RTSETMODE if you wish to ARM the "Filehandler Overlay" option, the "Single-Drive System" option, the "Ignore External Terminal" option and/or the "Get/Put and Filehandler Overlay" option.

Vendor Product Diskettes, or other diskettes which contain 48K Runtime System software should be copied using only "whole volume" transfer mechanisms, such as that provided by the Pascal system Filer. A succession of "individual file" transfers, or a "Wildcard" transfer (such as transferring "#5:=" to "#5:$"), will only copy files from one disk to another. They will not copy the crucial 48K Runtime boostrap code between disks. Only "whole volume" transfers (such as "#4:" to "#5:", or "SOUP:" to "NUTS:") will result in complete copies, containing the proper bootstrap information.

Vendor Product Diskettes, or other diskettes which contain 64K or 128K Runtime System software can be copied using either whole volume or individual file transfers since they do not contain special bootstrap information.

VIII. APPLE FORTRAN AND THE RUNTIME SYSTEMS

Apple FORTRAN programs will execute correctly under the Apple II Pascal 1.2 Runtime Systems (48K and 64K only), so long as no execution errors or untrapped I/O errors occur. Using only FORTRAN, it is impossible to produce object code that contains the specially-placed error-handling procedure to which control is transferred in the event of an untrapped error during Runtime execution. Furthermore, the FORTRAN Run Time Support Library includes system-level code for handling FORTRAN I/O errors independently of the Apple Pascal system's own error-handling facilities. Execution of this special code will always lead to a system re-boot in the Runtime environment.

Users who wish to provide turnkey packages based on FORTRAN object-code are advised to link the FORTRAN object-code to a Pascal host, as explained in the Apple FORTRAN Language Reference Manual. The only "live code" which the Pascal host must contain is the error-handling procedure that the Runtime Systems require for robust execution of turnkey software.

APPLE COMPUTER, INC.
20525 Mariani Avenue
Cupertino, CA 95014

PASCAL TECHNICAL NOTE #11

Apple Pascal 1.1
BIOS Reconfiguration Using ATTACH

(02 April 1981)

For further information contact:
PCS Developer Technical Support
M/S 22-W, Phone (408) 996-1010

Disclaimer of All Warranties and Liabilities

Notice

ATTACH-BIOS document for Apple II Pascal 1.1

By Barry Haynes

Jan 12, 1980

This document is intended for Apple II Pascal internal applications writers, Vendors and Users who need to attach their own drivers to the system or who need more detailed information about the 1.1 BIOS. It is divided into two sections, one explaining how to use the ATTACH utility available through technical support and the other giving general information about the BIOS. It is a good idea to read this whole document before assuming something is missing or hasn't been completely explained. This document is intended for more advanced users who already know a fair amount about I/O devices and how to write device drivers. It is not intended to be a simple step by step description of how to write your first device driver, nor does it claim to be a complete description of all there is to know about the Pascal BIOS.

The Apple Pascal UCSD system has various levels of I/O that are each responsible for different types of actions. It was divided at UCSD into these levels to make it easy to bring up the system on various processors and also various configurations of the same processor and yet have things look the same to the Pascal level regardless of what was below that level. The levels are:

LEVEL	TYPES OF IO ACTIONS
Pascal	READ & WRITE BLOCKREAD & BLOCKWRITE UNITREAD & UNITWRITE UNITCLEAR UNITSTATUS
RSP (Runtime Support Package)	This is part of the interpreter and is the middle man between the above types of I/O and the below types of I/O. All the above types are translated by the compiler and operating system into UNITREAD, UNITWRITE, UNITCLEAR and UNITSTATUS if they are not already in that form in the Pascal program. The RSP checks the legality of the parameters passed and reformats these calls into calls to the BIOS routines below. The RSP also expands DLE (blank suppression) characters, adds line feeds to carriage returns, checks for end of file (CTRL C from CONSOLE:), monitors UNITRW control word commands, makes

	calls to attached devices if present, echoes to the CONSOLE:.
BIOS (Basic I/O Subsystem)	This is the lowest level device driver routines. This is the level at which you can attach new drivers to replace or work with the regular system drivers and also attach drivers for devices that will be completely defined by you.

I.RECONFIGURING THE BIOS TO ADD YOUR OWN DRIVERS USING THE ATTACH UTILITY.

INTRODUCTION

With the Apple Pascal 1.1 System (both regular and runtime 1.1), there is an automatic method for you to configuer your own drivers into the system. This method requires you to write the drivers following certain rules and to use the programs ATTACHUD.CODE and SYSTEM.ATTACH provided through Apple Technical Support. At boot time, the initialization part of SYSTEM.PASCAL looks for the program SYSTEM.ATTACH on the boot drive. If it finds SYSTEM.ATTACH, it Xecutes it before Xecuting SYSTEM.STARTUP. SYSTEM.ATTACH will use the files ATTACH.DATA and ATTACH.DRIVERS which must also be on the boot disk. ATTACH.DATA is a file the developer will make using the program ATTACHUD. It tells SYSTEM.ATTACH the needed information about the drivers it will be attaching. ATTACH.DRIVERS is a file containing all the drivers to be attached and is constructed by the developer using the standard LIBRARY program. The drivers are put on the Pascal Heap below the point that a regular program can access it. They do take away Stack-Heap (= to the size of the drivers attached) space from that available to Pascal code files but this should not be a problem unless the drivers are very large or the code files very hungry in their use of memory. Since these drivers are configured into the system after the operating system starts to run, this method will not work for configuring drivers for devices that the system must cold boot from. Some of supporting code in the RSP, boot and Bios may make the task of bringing up boot drivers easier though. The advantages to this kind of setup are:

1. Software Vendors can use the ATTACHUD program to put their own drivers into the system at boot time. This will be invisible to the user.

2. There can be no problems losing drivers due to improper heap management since the drivers are put on the heap by the operating system and before any user program can allocate heap space.

3. This method does not freeze parts of the system to special memory locations since it enforces the clean methodology of using relocatable drivers.

USING ATTACHUD

ATTACHUD.CODE will ask you questions about the drivers you want to attach to the system. It makes a file called ATTACH.DATA which tells SYSTEM.ATTACH which drivers to attach to the system, what unit numbers to attach them to and other information. The options covered by ATTACHUD are:

1. A driver can be attached to one of the system devices, then all I/O to this device (PRINTER: for example) will go to this new driver. In the case of a new driver for a disk device the user will have to specify which of the 6 standard disk units will go to this new driver. This will allow replacement of standard drivers with custom ones without having to restrict the I/O interface to UNITREAD and UNITWRITE as is the case with option 2.

2. A driver can be attached to one of 16 userdevices. I/O to these will be done with UNITREAD and UNITWRITE to device numbers 128-143.

3. A method will be included to allow the attached driver to start on an N byte boundry. The driver writer will be responsible for aligning his code from that point.

4. More than one unit can be attached to the same driver. This way only one copy of the driver resides in memory and I/O to all the attached units goes to this one driver. It is up to the driver to decide which unit's I/O it is doing. How this is done is explained below.

5. The initialize routine for any attached driver can be called by SYSTEM.ATTACH after it has attached the driver and before any programs can be Xecuted.

6. In case any of your programs use the Hires pages, you can specify in ATTACHUD that drivers must not be put on the heap over these areas. Your drivers would have to be quite large before they could possibly overlap the Hires pages.

Follow through this example of a session with ATTACHUD where the options available are completely described. First Xecute ATTACHUD:

You will be given the prompt:

```
Apple Pascal Attachud [1.1]

Enter name of attach data file:
```

This is asking for what you want the output file from this session with ATTACHUD to be called. You could call it ATTACH.DATA or some other name and then rename it to ATTACH.DATA when you put it on the boot disk with SYSTEM.ATTACH.

If you ever get a message of the form:

ERROR => some error
Try again (RETURN to exit program):

then just retype what was requested on the previous prompt after deciding what mistake you made while typing it the first time.

The next prompt is:

> These next questions will determine if
> attached drivers can reside in the hires
> pages. It will be assumed they can for the
> page in question if you answer no to the
> prompt for that page.
> Will you ever use the (2000.3FFF hex)
> hires page?

Followed by:

> Will you ever use the (4000.5FFF hex)
> hires page?

You should answer yes to the question for a particular Hires page if you will ever be running a program that uses that Hires page while the drivers are Attached. You don't want the possibility of your driver residing in the Hires page if that page will be clobbered by one of your programs. After answering the Hires questions you will be asked the following questions once for each driver you will be attaching:

> What is the name of this driver? This
> must be the .PROC name in its assembly
> source (RETURN to exit program):

This must be the name of one of the drivers in the ATTACH.DRIVERS that will be used with this ATTACH.DATA. The length of this name must not be more than 8 characters. After entering the name you will be asked:

> Which unit numbers should refer to this
> device driver?
>
> Unit number (RETURN to abort program):

You must enter a unit number in the range 1,2,4..12,128..143 or will be given an error message. You cannot attach a character unit (CONSOLE:, PRINTER: or REMOTE:) to the same driver as a block structured unit and if you try you will be given the message:

> You can''t attach a character unit and
> a block unit to the same driver. I
> will remove the last unit# you entered.
> Type RETURN to continue:

If you don't get the above error, you will be asked:

> Do you want this unit to be
> initialized at boot time?

A yes response will put the unit number just entered on a list of units that SYSTEM.ATTACH will call UNITCLEAR on after attaching all

the drivers. This gives you a way to have the system make an initialize call on your attached unit at boot time. A no response will mean that no boot time init call will be made on this unit to the driver you just attached.

You will be eventually asked:

```
Do you want another unit number to refer
to this device driver?:
```

A yes response will get you to the Unit number prompt again and a no response will get you to the prompt:

```
Do you want this driver to start on a
certain byte boundary?
```

A yes here will give you more prompts:

```
The boundry can be between 0 and 256.
  0=>Driver can start anywhere.(default)
  8=>Driver starts on 8 byte boundary.
  N=>Driver starts on N byte boundary.
256=>Driver starts on 256 byte PAGE boundary.
Enter boundary (RETURN to exit program):
```

And the last line of the prompt will repeat until you enter a boundary in the correct range. The boundary refers to the memory location where the first byte of the driver is loaded. If your driver needs to be aligned on some N byte boundary you can assure it will be using this mechanism. if you know how the driver's origin is aligned, You can align internal parts of your driver however you want. Finally you will get to the prompt:

```
Do you want to attach another driver?
```

And if you answer Yes to this you will return to the 'What is the name of this driver' prompt and answering No will end the program, saving the data file you have made.

THE DRIVER

Drivers must be written in assembly using the Pascal Assembler. They must not use the .ABSOLUTE option, so the drivers can be relocated as they are brought in by the system. Each driver must be assembled separately with no external references. When all drivers are assembled, use the LIBRARY program (in the same way you would use it to put units into a library) to put all the drivers in one file. Name this file SYSTEM.DRIVERS. See further explanation of making SYSTEM.DRIVERS below.

Considerations for all drivers:

1. Study the examples below as certain information is only documented there.

2. Refer to the Apple II Pascal memory map below and you will see

that parts of the interpreter and BIOS reside in the same address range and are bank-switched. The system automatically folds in the BIOS for drivers added using ATTACH. Most of these drivers will have to make calls to CONCK if they want type ahead to continue to work properly. CONCK is the BIOS routine that monitors the keyboard. See the example drivers below to be sure you are doing this correctly. You cannot call CONCK through the CONCK vector at BF0A (see BIOS part of this document) because this call would go through the same mechanism used to get to your driver and the return address to Pascal would be lost.

3. All attached drivers must be written with one common entry point for read, write, init and status. The driver will use the Xreg contents to decide which type of I/O call this is and jump to the appropriate place within it's code. The Xreg is decoded as follows:

```
0 -->read (no bits set)
1 -->write (bit 0 set)
2 -->init (bit 1 set) § The Pascal statement
     UNITCLEAR(UNITNUMBER);  makes an init call for
     unit UNITNUMBER †
4 -->status (bit 2 set)
```

4. The drivers must also pop a return address off the stack, save it and later push it to do a RTS when the driver is finished. All other parameters must be removed from the stack by the driver. For all calls, the return address will be the top word on the stack.

5. SYSTEM.ATTACH will make a copy of the normal system jump vector (the vector after the fold) and put this on the heap. There will be a pointer to this vector at 0E2. Your drivers can use this vector to get to the normal system drivers for device numbers 1..12. See example below.

6. All drivers must pass back a completion code in the X register corresponding to the table on page 280 of the 1.1 "Apple II Apple Pascal Operating System Reference Manual".

7. In references below to parameters passed on the stack, all parameters are one word parameters so they require two bytes to be popped from the stack by the driver.

8. Control word format for Unitread & Unitwrite

bits	15..13	12..6	5	4	3	2	1..0
	user defined functions	reserved for future expansion	type B chars	type A chars	nocrlf	nospec	reserved for future expansion

```
type B =0 ==>System will check for CTRL S & F from CONSOLE:
             during the time of this Unitio call.
      =1 ==>System will not check for CTRL S & F during this
             Unitio.
type A =0 ==>If using Apple Keyboard, system will check for
             CTRL A,Z,K,W & E from CONSOLE: during the period
             of this Unitio.
```

```
                =1 ==>System will not check for the chars during
                     this Unitio.
        nocrlf =0 ==>line feeds are added to carriage returns by the
                     Interpreter.
                =1 ==>no line feeds are added ...
        nospec =0 ==>DLE's (blank suppression code) are expanded on
                     output and the EOF character is detected on input
                =1 ==>nothing special is done to DLE's on output and
                     EOF on input.

        default setting for all control word bits = 0.
```

9. Control word format for UNITSTATUS

```
bits  15..13    12..2      1          0
      user      reserved   for        direction
      defined   for future purpose

      direction =0 ==>Status of output channel is requested
                =1 ==>Status of input  ...
      purpose   =0 ==>Call is for unit status
                =1 ==>Call is for unit control
```

10. These are the new vectors and routines added to the BIOS to make attach work. The RSP, bootstrap, and readseg were also modified to allow for attaches.

```
UDJMPVEC    ;Jump vector for user devices, offset=0 => unattached device.
            ;The correct addresses are initialized by SYSTEM.ATTACH
            ;See locations section of BIOS part below for pointers to
            ;this vector.
            JMP     0               ;Unit 128
            JMP     0               ;Unit 129
            .
            .
            .
            JMP     0               ,Unit 143

DISKNUM     ;If high byte=FF then
            ;  device is not a disk drive
            ;else
            ;  if high byte=0 then
            ;    device is a regular disk drive and low byte=drive #
            ;  else
            ;    driver for this disk drive has been attached by SYSTEM.ATTACH
            ;    and the driver address is stored in this word.
            ;    (Driver address has to be the address-1 for RTS in PSUBDR
            ;     to work correctly, remember this for ATTACH.  PSUBDR is
            ;     listed below.)
            ;See locations section of BIOS part below for pointers to
            ;this vector.
            .WORD   OFFFF           ;Unit #1
            .WORD   OFFFF           ;Unit #2  (ATTACH would modify the words
            .WORD   OFFFF           ;Unit #3   for units 4,5,9..12 if a
            .WORD   0               ;Unit #4   different disk driver were
            .WORD   1               ;Unit #5   attached to any of them)
            .WORD   OFFFF           ;Unit #6
```

```
        .WORD   0FFFF           ;Unit #7
        .WORD   0FFFF           ;Unit #8
        .WORD   4               ;Unit #9
        .WORD   5               ;Unit #10
        .WORD   2               ;Unit #11
        .WORD   3               ;Unit #12

UDRWIS  ;Routine to get to an attached driver through UDJMPVEC
        ;Assume unit# in Areg & operation to be performed in Xreg.
        ;See the jump vector in the BIOS sections to see how you
        ;get to this routine.
        STA     TT1
        AND     #7F             ;Clear top bit of unit#
        STA     TT2             ;Make address in UDJMPVEC table
        ASL     A               ;Address=Areg*3 + base of table
        CLC
        ADC     TT2             ;Now we have (Areg*3).
        ADC     #JVECTRS        ;Add in low byte of base of table having
        STA     TT2             ;no carry problem with only 16 UD's.
        LDA     #0
        ADC     JVECTRS+1       ;JVECTRS is a word pointing to the base
                                ;of UDJMPVEC.
        STA     TT2+1
        LDA     TT1
        JMP     @TT2

PSUBDR  ;Routine to get to an attached driver through DISKNUM
        ;We assume on entry, Areg=unit#, Yreg=DISKNUM
        ;offset & Xreg=the command to be performed by the substituted
        ;disk driver.
        ;See the jump vector in the BIOS sections to see how you
        ;get to this routine.
        STA     TT1         ,Save unit#.
        LDA     DISKNUM-1,Y ;Store MSB of driver address.
        PHA
        LDA     DISKNUM-2,Y ;Store LSB of driver address.
        PHA
        LDA     TT1         ;Restore unit# to Areg.
        RTS                 ;Jump to substituted driver.  This assumes
                            ;the driver address in DISKNUM =
                            ;(ADDRESS OF DRIVER)-1 for the RTS to work
```

Special considerations when attaching drivers for the system devices, unitnumbers 1..12.

A. Character Oriented Devices (Pass the character to be read-written in the A-register and make Bios calls one character at a time from RSP level. On entry, the unit number will be in the Y register in case you wanted to attach all character oriented devices to the same driver). If you attach REMOTE: & or PRINTER: to the same driver as CONSOLE:, all will have their jump vectors pointing to the start of the driver+3 bytes. See further discussion on this below.

   Units 1 & 2 (CONSOLE: and SYSTERM:)
   1. These must both go to the same driver.

2. The system CONCK routine will be patched to jump to the start of the driver. The CONCK routine gets characters entered at the keyboard and fills the type ahead buffer. See the example CONSOLE: driver below.
3. Because of item 2, the entry point for normal calls (not CONCK calls) to the attached driver will be 3 bytes beyond the start of the driver.
4. The interpreter takes care of expanding blank suppression codes (DLE's), echo to the screen, EOF (the end of file character), and adding line feeds to every carriage return. Your driver doesn't need to do this.
5. CONSOLE: read and write have only the return address on the stack. The stack for CONSOLE: init looks like:

```
POINTER TO BREAK VECTOR (This should be stored at
                         location BF16..BF17 by CONSOLE:
                         init.)
POINTER TO SYSCOM       (This should be stored at
                         location F8..F9 by CONSOLE:
                         init.)
                        (Also at init time, the Flush
                         and Start/Stop conditions
                         should be set to normal and
                         the type-ahead queue should
                         be emptied.)
RETURN ADDRESS         <--TOS (top of stack)
```

The stack for CONSOLE: status looks like:

```
POINTER TO STATUS RECORD
CONTROL WORD
RETURN ADDRESS         <--TOS
```

6. A status request should return, in the first word of the status record, the number of characters currently queued in the direction asked for. This is the number of characters in the type-ahead buffer. If no type-ahead is being used then output status should always return a 0 and input status a 1 if a char is waiting to be read, otherwise a 0.
7. Since we are using 7 bit ASCII codes, the CONSOLE: read routine should zero the high order bit of all characters it reads from the keyboard and passes back to Pascal (to the RSP). The CONSOLE: write routine should transfer all 8 bits as received from the RSP since many devices use 8 bit control codes.
8. The RSP will send both upper and lower case chars to the CONSOLE: write routine. The write routine should map the lower to upper if the device cannot handle lower case.
9. CONSOLE: Output Requirements:
   A. CR (0D hex) A carriage return should move the cursor to the beginning of the current line.
   B. LF (0A hex) A line feed should move the cursor to the next line but not change the column position. If the cursor is on the last line on the screen when a line feed is sent, the rest of the screen should scroll up one line and the bottom line be cleared.
   C. BELL (07 hex) A sound should be made if possible when the CONSOLE: gets 07. If making a sound is not possible then ignore the 07.
   D. SP (20 hex) Place a space at the current cursor position overwriting whatever is there. Move the cursor to the next column. If the cursor is on the last column of a line, it is best if the cursor stays where it is after the space fills that

position. If the cursor is on the last column of the last line on the screen, it is also best if the cursor remains in that position and the screen does not scroll. These are the prefered actions of the cursor at end of line & end of screen; in the strict sense, the actions of the cursor in these circumstances are undefined.

E. NUL (00 hex) When a Null is sent to the CONSOLE: from the RSP, the CONSOLE: should delay for the ammount of time required to write one character but the state of the screen should not change.

F. All printable characters should be written to the screen and the cursor should move in the same way it does for SP.

G. See the discussion on pages 199-215 in the 1.1 Operating System Reference Manual for further requirements and information.

10. CONSOLE: Input Requirements:

A. The RSP takes care of echoing characters to the screen typed from the CONSOLE: keyboard.
(below items optional The Start/Stop, Flush & Break chars are redefinable; see 9G above for more info.)

B. The Start/Stop character is detected by CONCK and is used to stop all processing until the character is received a second time. When the character is received (see 9G above for more info) one should loop in CONCK continuing to process other characters until:

1. the S/S char is received again
2. the Break char is received

In case 1, the suspended processing should continue as it was before the first S/S was typed. Action needed for the Break char is described below. The S/S char is never returned to the RSP and CONSOLE: type-ahead, if implemented, should continue during the suspended state. Offset from SYSCOM to this char is 85 decimal. (This and the next 2 chars are redefinable by the Setup program and SYSCOM is the system area that keeps track of this info. The pointer to the start of SYSCOM is passed to the CONSOLE: init routine and is stored at F8..F9 hex.)

C. The Flush character will stop all output and echoing to the CONSOLE: until it's second occurEnce (see 9G above). CONCK detects this and must set a flag to tell the CONSOLE: output routine to ignore characters while the flag is set. If the CONSOLE: is re-initialized or a Break-char is received, the flush state should be turned off. Flush is never returned to the RSP. Flush only stops CONSOLE: output, other processing continues. Offset from SYSCOM to this char is 83 decimal.

D. The Break char should cause CONCK to jump to the location stored at BF16. This location is also passed to the CONSOLE: init routine which stores it at BF16. The break char is never returned to the RSP and it should remove the system from Stop or Flush mode if it is in either mode. Offset from SYSCOM to this char is 84 decimal.

E. Type-ahead should be implemented in CONCK by storing characters typed at the keyboard in a queue until they are requested by a CONSOLE: read from Pascal. When the queue fills, further characters should be ignored and a bell sounded when they are typed. The length of the queue should be at least 20 characters.

11. For more information on CONSOLE: requirements, see pages 199-

Unit 6 (the PRINTER:)

1. The interpreter takes care of expanding blank suppression codes (DLE's), EOF (the end of file character), and adding line feeds to every carriage return.
2. PRINTER: read,write and init have only the return address on the stack. PRINTER: status has the same items on the stack as CONSOLE: status. PRINTER: init should cause the PRINTER: to do a carriage return and a line feed and throw away any characters buffered to be printed. No form feed should be done.
3. For status, return in the first word of the status record the number of bytes buffered in the direction asked for; if this cannot be determined by your PRINTER:, return a 0.
4. The PRINTER: write routine must buffer a line and send it all at once if your PRINTER: can only receive data that way.
5. Line Delimiter characters:
   A. CR (hex 0D) A carriage return should cause the PRINTER: to print the current line and return the carriage to the first column. An automatic line feed should not be done by the PRINTER: driver when it reads a CR.
   B. LF (hex 0A) The RSP will send line feeds to the PRINTER: driver after each carriage return. This should cause the PRINTER: to advance to the next line. If the PRINTER: must also do a carriage return when it is given a line feed, then this is O.K.
   C. FF (hex 0C) This should cause the PRINTER: to move the paper to top of form and do a carriage return. If top of form is not possible on your PRINTER:, do a carriage return followed by a line feed.
6. It is assumed that input cannot be received from the PRINTER:. See the BIOS section for a discussion of how to get input from the PRINTER:. Normally, trying to get input from the PRINTER: should return completion error code #3.

Units 7 (REMOTE: in) & 8 (REMOTE: out)

1. These must both go to the same driver.
2. The interpreter takes care of expanding blank suppression codes (DLE's), EOF and adding line feeds to every carriage return.
3. Same stack setup as the PRINTER:.
4. Status should return in first word of status vector the number of bytes buffered for the direction specified in the control word, 0 if you have no way to check.
5. This unit is supposed to be an RS-232 serial line for many different applications so it is necessary that it transfer the data without modifying it in any way. The transfer rate default is 9600 baud.
6. It would be nice if the input to REMOTE: could be buffered in the same way input to the CONSOLE: is but this is not an absolute requirement.
7. REMOTE: init should set up the REMOTE: device so it is ready to read and write.

B. Block Structured Devices

Units 4 (the boot unit),5,9,10,11,12.

1. These units are assumed to be block structured devices, the drivers for these units must do their own Pascal Block to Track-Sector conversions.
   The UCSD system assumes the disk device is a 0-based consecutive array of 512 byte logical blocks. All UCSD Pascal disks must have this logical structure no matter what their actual physical structure or size are. The physical allocation schemes for information on different types of disks are arranged with sectors that are of various sizes that depend on the hardware of the particular disk device used. The driver must convert the Pascal block # to the appropriate track & sector # of where that block is stored on it's disk device. This could be a floppy or hard disk or some other type of device. It doesn't really matter, so long as your driver maps the Pascal Block to the cOrrect place and continues to do so for the length (byte count) required for the UnitIO operation.
   The Pascal system uses logical blocks 0 & 1 for it's bootstrap code. These logical blocks should not be used for anything else and should therefore only be available to Pascal through direct UNITREAD & UNITWRITE operations and not accessable by the system through any other means. This document will not attempt to describe the boot sequence & does not attempt to give you enough information to attach another driver or device to unit #4: so you can cold boot from that unit.
   When a UNITWRITE is done to disk where the byte count MOD 512 is not equal to 0 (this means the last block included in the write would be partially written to according to the byte count), it is undefined whether garbage is written into the remaining part of this last block. So you may write a whole block anyhow if that is more efficient and the Pascal system will not suffer any bad consequences.
   When a UNITREAD is done from a disk you are not allowed to overwrite into the unused part of the last block (if there is an unused part due to byte count MOD 512 <> 0). You must only send the number of bytes asked for because you could clobber memory having some other valid use if you wrote extra bytes. You will have to buffer the last sector inside your disk read routine then transfer exactly the number of bytes from this last sector needed to add up to the total bytes requested.
2. The unit number will always be in the A register.
3. The stack setup for read and write is:

```
CONTROL WORD      (The MODE parameter mentioned in the
                   1.1 Language Ref Manual on page 41)
DRIVE NUMBER
BUFFER ADDRESS
BYTE COUNT
BLOCK NUMBER
RETURN ADDRESS  <--TOS
```

   For init there is only the return address on the stack and for status the setup is the same as for the CONSOLE:.
4. Status requests should return the following in the status record:
   word1:Number of bytes buffered in the direction asked for in the control word. Return 0 if you have no way of checking.
   word2:Number of bytes per sector.

word3:Number of sectors per track.
word4:Number of tracks per disk.

C. Other
Unit 3
1. This unit has no meaning for the Apple II system except that UNITCLEAR on this unit sets text mode.

Considerations when attaching drivers for user defined devices numbers 128-143.

These unit numbers are provided for you to do whatever you want with them. you can define what they do except for the following protocols.

1. Follow the considerations for all drivers listed above.
2. The unit number will always be in the A register.
3. The stack setup for read and write is:

```
CONTROL WORD
DRIVE NUMBER
BUFFER ADDRESS
BYTE COUNT
BLOCK NUMBER
RETURN ADDRESS  <--TOS
```

For init there is only the return address on the stack and for status the setup is the same as for the CONSOLE:.

This is a sample driver for a user defined device.

```
        ;Locations 0..35 hex may be used as pure temps. One should
        ;never assume these locations won't be clobbered if you leave
        ;the environment of the driver itself. ("leaving" includes
        ;calls to CONCK).

CONCKADR .EQU    02

        ;Only one .PROC may occur in a driver, each driver to be
        ;ATTACHED must be assembled separately using the Pasçal
        ;assembler and can have no external references.

        .PROC U128DR

        STA   TEMP1     ;Save Areg contents (unit#)
        PLA
        STA   RETURN
        PLA
        STA   RETURN+1
        TXA             ;Use the X reg to tell you what kind of
                        ;call this is.
        CMP   #2
        BEQ   INIT
```

```
        CMP   #4
        BEQ   STATUS
        CMP   #0
        BEQ   PMS
        CMP   #1
        BEQ   PMS

        ;Could have error code here
        JMP   RET

PMS     PLA             ;Get the parameters
        STA   BLKNUM
        PLA
        STA   BLKNUM+1
        PLA
        STA   BYTECNT
        PLA
        STA   BYTECNT+1
        PLA
        STA   BUFADR
        PLA
        STA   BUFADR+1
        PLA
        STA   UNITNUM   ;Also in TEMP1
        PLA
        STA   UNITNUM+1 ;Should always be 0
        PLA
        STA   CONTROL
        PLA
        STA   CONTROL+1
        TXA
        BNE   WRITE

READ    JSR   GOTOCK
        ;Your driver's code for a read
        (If more than one unit were attached to this driver, this
         code could jump to various places depending on the contents
         of the Areg stored in TEMP1)
        JMP   RET

WRITE   JSR   GOTOCK
        ;Your driver's code for a write
        JMP   RET

        ;If you wanted to call CONCK whenever your device did a read
        ;or write, you would use this routine:
CKR     .WORD CONCKRTN-1
GOTOCK  LDY   #55.      ;Offset to address of CONCK
        LDA   @0E2,Y
        STA   CONCKADR
        INY
        LDA   @0E2,Y
        STA   CONCKADR+1
        LDA   CKR+1     ;Set it up so you return to CONCKRTN after
        PHA             ;the CONCK call.
        LDA   CKR
        PHA
        JMP   @CONCKADR ,Jump to CONCK
```

```
CONCKRTN RTS            ;Return to caller.

INIT     ;Your driver's code for init
         JMP   RET

STATUS   PLA
         STA   CONTROL
         PLA
         STA   CONTROL+1
         PLA
         STA   BUFADR    ;Address of status record.
         PLA
         STA   BUFADR+1
         ;Your driver's code for status

RET      LDA   RETURN+1
         PHA
         LDA   RETURN
         PHA
         LDA   TEMP1
         RTS


RETURN   .WORD   0      ;Can't use 0 page for these since we leave
TEMP1    .WORD   0      ;our environment when going to CONCK.
CONTROL  .WORD   0
UNITNUM  .WORD   0
BUFADR   .WORD   0
BYTECNT  .WORD   0
BLKNUM   .WORD   0


         .END
```

This is a sample driver for a CONSOLE: driver replacement.

```
ROUTINE  .EQU  02
TEMP1    .EQU  04


         .PROC CKATCH

         JMP   CONCKHDL ;SYSTEM.ATTACH will patch the start of CONCK
                        ;to jump here when you attach a driver to the
                        ;CONSOLE:.

                        ;We are not popping the return address from
                        ;the stack cause we'll return from the system
                        ;routine we call from this driver.
         STA   TEMP1    ;All the read,write,init and stat calls will
                        ;jump here (the starting address of your
                        ;CONSOLE: driver+3).

         STY   TEMP1+1
         TXA
```

```
                        ;This example shows you how to have your
                        ;own code for the CONSOLE: as well as using
                        ;the system CONSOLE: routines.  If you want
                        ;to replace the system routines completely,
                        ;you need to pull the return address here.
        BEQ   READ
        CMP   #1
        BEQ   WRITE
        CMP   #2
        BEQ   INIT
        CMP   #4
        BEQ   STATUS

        ;Error code here

READ    ;Your driver's code for a read

        LDY   #1        ;offset to address of CONSOLE: read in
                        ;the copy of the jmp vector made by
                        ;SYSTEM.ATTACH.  See the jump vectors in the
                        ;BIOS section below to see how we get the
                        ;offsets.

        BNE   GET

        ;You would have a JMP RET here (see example for user defined
        ;device) if you were not using the system CONSOLE: routines
        ;as well.

WRITE   ;Your driver's code for a write
        LDY   #4
        BNE   GET

INIT    ;Your driver's code for init
        LDY   #7
        BNE   GET

STATUS  ;Your driver's code for status
        LDY   #43.

GET     LDA   @0E2,Y    ;At E2 is a pointer to the copy of the
                        ;jump vector made by SYSTEM.ATTACH before
                        ;it was modified to attach your drivers.

        STA   ROUTINE
        INY
        LDA   @0E2,Y
        STA   ROUTINE+1
        LDY   TEMP1+1   ;Restore registers
        LDA   TEMP1
        JMP   @ROUTINE  ;Go to the original CONSOLE: driver for this
                        ;I/O command.  You will return from there; the
                        ;BIOS is already folded in due to the way your
                        ;driver was attached by SYSTEM.ATTACH.


CONCKHDL PHP            ;Duplicate the 1st three instructions of CONCK
         PHA            ;as they were patched by SYSTEM.ATTACH to jump
```

```
        ;TXA below      ;to the 1st instruction of this driver.

        ;Here you can put the code for your own part of CONCK (you
        ;may want to check some additional device like a keypad or
        ;something or you may want to replace the system CONCK
        ;routine alltogether.  If you do this, you must save the rest
        ;of the machine state and return it when you are finished.
        ;See example below.

        TYA             ;Save Yreg contents for a second.
        PHA

        ;This code gets us to the system CONCK routine.
        CLC
        LDY   #55.      ;Offset to the address of system CONCK in the
                        ;copy of the original jmp vector.

        LDA   @0E2,Y
        ADC   #3        ;Add 3 so you enter right after the three
                        ;instructions you duplicated at CONCKHDL.
        STA   ROUTINE
        INY
        LDA   @0E2,Y
        ADC   #0
        STA   ROUTINE+1
        PLA             ;Restore Yreg.
        TAY
        TXA             ;Last of CONCK instructions SYSTEM.ATTACH
                        ;overwrote with the jmp to the start of this
                        ;driver.

        JMP   @ROUTINE ;Goto system CONCK and return from there.

        .END
```

Here is another alternative for the CONCKHDL part of the above program.

```
CKRTN    .WORD CONCKRTN-1
CONCKHDL ; 1.If you don't care about type-ahead, this could be
         ;   simply the following code (assuming your CONSOLE:
         ;   read gets a character directly from your CONSOLE:
         ;   device whenever it is called) :

-----------------------------------------------------------------
         PHP
         INC RANDL    ;RANDL is a permanent word at BF13 used in
                      ;the built in random function.
         BNE $1
         INC RANDH    ;RANDH
$1       PLP
         RTS
-----------------------------------------------------------------

         ; 2.If you want type-ahead, this code should check to see
         ;if there is a character available and stuff it into a type-
         ;ahead buffer.
```

```
        ; 3.If you are using this with the regular CONCK (extra keypad
        ;to check or statistics for example), then you can do it this
        ;way.

-----------------------------------------------------------------
        PHP         ;Save state of machine
        PHA
        TXA
        PHA
        TYA
        PHA

        ;Put your driver's part of CONCK here (gives your driver
        ;priority)

        LDA CKRTN+1 ;Set up things to return from reg CONCK
        PHA
        LDA CKRTN
        PHA
        PHA         ;Push garbage to account for other pushes done
        PHA         ;in first three bytes of CONCK

        CLC             ;Setup to call CONCK
        LDY   #55.      ;Offset to the address of system CONCK in the
                        ;copy of the original jmp vector.

        LDA   @0E2,Y
        ADC   #3        ;Add 3 so you enter right after the three
                        ;instructions you duplicated at CONCKHDL.
        STA   ROUTINE
        INY
        LDA   @0E2,Y
        ADC   #0
        STA   ROUTINE+1
                        ;In this example we don't have to worry about
                        ;the machine state here as we are restoring
                        ;it after we call CONCK

        JMP   @ROUTINE ;Goto system CONCK and return to CONCKRTN

CONCKRTN PLA            ;Restore state of machine
        TAY
        PLA
        TAX
        PLA
        PLP
        RTS             ;Return to the guy who called CONCK.
-----------------------------------------------------------------
```

MAKING ATTACH.DRIVERS

1. Xecute the standard 1.1 LIBRARY program.
2. The output code file should be ATTACH.DRIVERS or could be named somethine else and renamed ATTACH.DRIVERS when you put it on the boot disk.
3. For the Link code file use the code file of your first driver.

4. Copy its slot #1 into slot #0 of ATTACH.DRIVERS.
5. As long as you have more drivers to add, use N(EW to get another Link code file and copy it's slot #1 into slots #2,3,...15 of ATTACH.DRIVERS.
6. When done, type 'Q' then 'N' followed by a RETURN for the notice. See the 1.1 Operating System Reference Manual for further info on the LIBRARY program.

THE WORKINGS OF SYSTEM.ATTACH

If it is on the boot disk, SYSTEM.ATTACH is Xecuted by the operating system (both regular 1.1 and runtime 1.1) before SYSTEM.STARTUP. The 1.1 runtime system will use a runtime version of SYSTEM.ATTACH.

The error messages that can be generated by SYSTEM.ATTACH are:

1. ERROR =>No records in ATTACH.DATA
2. ERROR =>Reading segment dictionary of ATTACH.DRIVERS
3. ERROR =>reading driver
4. ERROR =>A needed driver is not in ATTACH.DRIVERS
5. ERROR =>ATTACH.DATA needed by SYSTEM.ATTACH
6. ERROR =>ATTACH.DRIVERS needed by SYSTEM.ATTACH

If all goes well attaching drivers, SYSTEM.ATTACH will display nothing unusual in the regular boot sequence except for extra disk accesses and anything done in the init calls to any of the attached devices.

II.BIOS

This section explains things in the BIOS area that are extensions and modifications that were added to Apple Pascal version 1.1 that were different or not there at all in Apple Pascal version 1.0 (UCSD version II.1).

1. The disk routines have been modified to handle interrupts (So interrupt driven devices could be attached to 1.1 Pascal) if they are being used. To use interrupts, one would have to attach an interrupt driver, then patch the IRQ vector (FFFE hex) to point to this driver. The Pascal system is defined to come up with interrupts turned off so, once the driver is brought in and the IRQ patched, interrupts must be turned on. The driver's init call could patch the IRQ and turn on interrupts. The disk routines save the current state of the system and turn interrupts off only during crucial time periods, the state of the system is returned during non crucial time periods so interrupts can be handled. This has not been tested at this time, so there is no data concerning the maximum interrupt response time delay.

2. The control word parameter in UNITREAD and UNITWRITE was not passed on to the BIOS level routines from the RSP level. This has been done

in 1.1 to allow the changes to the control word listed below under special character checking and also so user defined units or attached Pascal units can use the user defined bits of the control word.

3. IORESULTS 128-255 are available for user definition on user defined devices.

4. UNITSTATUS has been implemented in the Apple II Pascal 1.1 system. This works for the Pascal system units as described in the ATTACH part of this document. For user defined units, Unitstatus can be used for whatever necessary.

   Unitstatus is a procedure that can be called from the Pascal level in the same way Unitread can. It has three parameters:

   1. unit#.
   2. pointer to a buffer.
      (any size buffer you want of type Packed Array of Char)
   3. control word.

   When you make a Unitstatus call from Pascal, the call should look like:

   ```
   UNITSTATUS(UNITNUM,PAC,CONTROL);
   ```

   Where UNITNUM & CONTROL are integers and PAC is a Packed Array of CHAR or a STRING and may be subscripted to indicate a starting position to transfer data to or from. See further information on what Unitstatus is defined to do for the various devices in the ATTACH part of this document.

   The control word will tell the status procedure for a particular unit what information about the unit you want. Bit 0 of this word should equal 1 for input status and 0 for output status. Unitstatus is implemented with bit 1 of the control word =1 meaning the call is for unit control. When this bit =0 the call is for unitstatus. In all cases bits 2-12 are reserved for system use and bits 13-15 are available for user defined funtions.

   An entry in the jump vector has been made for each of the system Unitstatus calls, i.e. CONSOLESTAT,PRINTERSTAT,REMOTESTAT,etc.. Unitstatus calls to a user defined device (128-143) will all go through the same jump vector location.

5. The handling of CTRL-C by the Apple bios was non standard in 1.0. The UCSD BIOS definition specifies that a CTRL-C coming from REMOTE: or the PRINTER: should be placed in the input buffer and then no more characters should be received. Our bios did fill the buffer with nulls including the place where the CTRL-C was to go. Apple Pascal's BIOS now conforms to the standard definition, where the null filling of the buffer is done only when CTRL-C comes from the CONSOLE: (#1:).

6. The unitio routines can be accessed from assembly procedures by pushing the correct parameters on the stack and using the jump vector to get to the BIOS routine. A seperate document needs to be written describing how this is done and pointing out the problems doing it in the case of the CONSOLE:,SYSTERM:,PRINTER: & REMOTE: units.

These problems are concerned with the special character handling done in the RSP for these units. The assembly procedures calling the pascal drivers for these units would either have to repeat portions of the RSP code themselves or not get the special character handling provided by the RSP. Calling the CONSOLE: init routine requires pointers to syscom and the break routine to be passed on the stack. These pointers are now stored in a fixed location so assembly routines wanting to call coninit can get at them. See the locations section.

7. Suppression of Special Character Checking.

   Special characters in the Pascal system are of three types:

   A. Chars used to control the 40 character screen. These are ctrl-A,Z,W,E & K.

   B. Pascal system control chars for general CONSOLE: use. These are ctrl-S & F.

   C. Types A & B are checked for by the CONCK funtion in the bios. There are other special chars checked for in the RSP. These are ctrl-C, DLE, and CR (line feeds are automatically appended to CR). With UNITREAD and UNITWRITE the automatic handling done by the Pascal system of these characters can be turned off. To turn off DLE expansion and EOF checking give bit 2 of the control word a value of 1. The automatic adding of line feeds to carriage returns can be suppressed by setting bit 3 of the control word to 1.

      A way was needed to suppress special handling for types 'A'&'B'. This can now be done in two ways. First, the control word of UNITR/W will turn off checking for type 'A' control chars if bit 4 is set and will turn off checking for type 'B' chars if bit 5 is set. In this mode, the special char handling will only be turned off during that particular unitio. This will be be done for you in the RSP by setting bits in a byte 'SPCHAR' at location BF1C. The CONCK routine will look at bit 0 of SPCHAR and if set will not look for the type 'A' chars; if bit 1 is set, it will not look for the type 'B' chars. If you set these bits in the SPCHAR yourself instead of letting the RSP do it through the unitio control word, then the associated special character checking will be turned off until you reboot or reset the bits again. When special char checking is turned off, the chars are passed back to the Pascal level like all other chars would be. You can use these added features to redefine the system special chars in a particular application program or to just disable them.

8. The EOF char (ctrl-C) causes a lot of problems in the Pascal system. The cause of the problems is that the editor looks for this character to end many of it's editing modes. The editor has it's own getchar routine which reads each character the user enters from SYSTERM:. When reading from SYSTERM: instead of the CONSOLE:, the EOF char is passed back as any other character but it still ends the current call to unitread. The editor echoes each char to the CONSOLE: itself until it comes to ctrl-C. The operating system and the filer both use the getchar routine in the operating system. This routine is defined to re-init the system if it gets a ctrl-C from the CONSOLE: and it reads from the CONSOLE:, not SYSTERM:. You must be sure not to end responses with control-C except for the cases (in the editor only) that are

supposed to end with control-C. See the 1.1 Operating System Reference Manual.

9. The bios card recognizing section has been enhanced to recognize a new 'FIRMWARE' type card. This card will allow OEM's to have their drivers in their own firmware on the card. Routines have been added to allow for init,read,write & status calls to this new type card. This protocol has been documented and is attached as an appendix to this document.

10. As you can see, the Pascal system memory usage is scattered all over the 64k space. The Apple II was not designed with a stack machine, like the Pascal P-machine, in mind. We don't need any more constraints fixing certain pieces of the system to certain EXACT places. To make the best use of the space we have, we must have the ability to move things around. To achieve this goal, we intend the following:

    A. To stop people from writing things that peek here and poke there and expect things to stay exactly where they were for future versions.

    B. Various people need space for patch areas and other purposes. All programs have to be written so this space does not have to be in a permanent fixed location if this is at all possible. The areas reserved for system use are filling up fast, we need to avoid using them. You can get space dynamically using NEW but you must be careful that this space stays around for the whole time you need it. If you are attaching a driver, you can get buffer space in the driver by using .WORD or .BLOCK in the Assembler. This space can be accessed from outside the driver if you know the offset to the start of this space from the start of the driver. This method could even be used to get space below the heap by attaching a driver to one of the user defined devices that is a large .BLOCK and is only used as a buffer. You can get the address of this buffer (of a driver) from the jump vector that has a pointer to the driver. Pointers to all the jump vectors are in zero page, see the locations section below.

    C. The jump vector will have a fixed order for version 1.1 and future versions. The order is the same as in the old version 1.0 with the new entrys added to the bottom. The setup for the jump vector and getting into the BIOS is different than the old 1.0 system. Here is how the new system is set up with the fixed order for the jump vector:

```
;-----------------------------------------------
;
;  MAIN BIOS JUMP TABLE CALLED FROM INTERPRETER
;  (FOLLOWED BY REAL JUMP TABLE AT FIXED OFFSET)
;  RSP CALLS COME TO THIS JUMP VECTOR
;
;-----------------------------------------------
```

```
BIOS        JSR SAVERET         ;CONSOLE READ   ;Jump vector before fold.
            JSR SAVERET         ;CONSOLE WRITE
            JSR SAVERET         ;CONSOLE INIT
            JSR SAVERET         ,PRINTER WRITE
            JSR SAVERET         ;PRINTER INIT
            JSR SAVERET         ;DISK WRITE
            JSR SAVERET         ;DISK READ
            JSR SAVERET         ;DISK INIT
            JSR SAVERET         ;REMOTE READ
            JSR SAVERET         ;REMOTE WRITE
            JSR SAVERET         ;REMOTE INIT
            JSR SAVERET         ;GRAFIC WRITE
            JSR SAVERET         ;GRAFIC INIT
            JSR SAVERET         ;PRINTER READ
            JSR SAVERET         ;CONSOLE STAT
            JSR SAVERET         ;PRINTER STAT
            JSR SAVERET         ;DISK STAT
            JSR SAVERET         ;REMOTE STAT
KCONCK      JSR SAVERET         ;To get to CONCK from CONCKVEC
            JSR SAVERET         ;USER READ   For UDRWIS
                                ;USER WRITE
                                ;USER INIT
                                ;USER STAT
             JSR SAVERET        ;For PSUBDR
             JSR SAVERET        ;IDSEARCH
                .
                .
                .

;-------------------------------------
;
;  THIS JUMP TABLE MUST BE OFFSET
;  FROM BIOSTBL BY EXACTLY $5C.
;  SYSTEM.ATTACH MODIFYS THIS JUMP
;  VECTOR TO POINT TO ATTACHED DRIVERS
;  FOR THE STANDARD SYSTEM UNITS.
;
;-------------------------------------

BIOSAF      JMP CREAD           ;Jump vector after fold.
            JMP CWRITE
            JMP CINIT
            JMP PWRITE
            JMP PINIT
            JMP DWRITE
            JMP DREAD
            JMP DINIT
            JMP RREAD
            JMP RWRITE
            JMP RINIT
            JMP IORTS          ;Do nothing for GRAFWRITE.
            JMP GRAFINIT
            JMP IORTS          ;Do nothing for PRINTER: read.
            JMP CSTAT
            JMP ZEROSTAT       ;For PRINTER: stat, pop params & store 0
                               ;in 1st buffer word.
            JMP DSTATT
            JMP ZEROSTAT       ;For REMOTE: stat, pop params & store 0
```

```
                                ;in 1st buffer word.
            JMP CONCK
            JMP UDRWIS          ;Routine to get to user defined devices, see
                                ;ATTACH part of document for description of
                                ;this routine.
            JMP PSUBDR          ;Routine to get to drivers that are substituted
                                ;for the standard Pascal disk units 4,5,9..12.
                                ;See ATTACH part of document for description of
                                ;this routine.
            JMP IDS


;----------------------------------------
;
; STRIP LOCAL RETURN ADDR,
; STRIP PASCAL ADDR AND SAVE IN RETL,RETH
; PLACE 'GOBACK' ON RETURN STACK
; THEN RESTORE LOCAL RET ADDR & RETURN
; MEANWHILE UNFOLD BIOS INTO DXXX
;
;----------------------------------------
SAVERET     STA TT1             ;SAVE A REG
            PLA
            CLC
            ADC #05A            ;ADD OFFSET TO JUMP TABLE (BIOSAF)
            STA TT2             ;LOCAL RET ADDR
            PLA
            ADC #0
            STA TT3
            PLA
            STA RETL            ;PRESERVE PASCAL RETURN
            PLA
            STA RETH
            .IF RUNTIME=0
            LDA 0C083           ;UNFOLD BIOS INTO DXXX
            .ENDC
            LDA TT1             ;RESTORE A-REG
            JSR SAVRET2         ;PUTS 'GOBACK' ON STACK

;----------------------------------------
;
; FOLD INTERP INTO DXXX
; THEN RETURN TO PASCAL VIA
; RETURN ADDR SAVED IN RETL,RETH
;
;----------------------------------------
GOBACK      STA TT1             ;SAVE A-REG
            LDA RETH
            PHA
            LDA RETL
            PHA
            .IF RUNTIME=0
            LDA 0C08B           ;FOLD INTERP INTO DXXX
            .ENDC
            LDA TT1
            RTS                 ;AND BACK TO PASCAL

SAVRET2     JMP @TT2            ;JUMP INTO JUMP TABLE (BIOSAF)
```

D. In zero page are two words pointing to the base of the two jump vectors (before and after the fold). These are stored in PERMANENT locations that had a value of 0 in the old 1.0 release and were not used by the system (see locations section). Applications needing to patch the jump vectors can store the offset from the vector base in the Y reg and use indirect indexed addressing to do the patch. The application will need to have the vector base locations for the old release hardcoded in as the base pointer for the old 1.0 release will be 0. If you want to write an application that works with 1.0 and 1.1 and future versions, you know if the zero page vector pointers are 0 it's the 1.0 system otherwise it's 1.1 or a future version which will use the same protocols as 1.1 as described in this document.

   It is important that any application patching the jump vector temporarily then returning it to its original value get the original value from the vector itself before the patch and put it in a storage location. When the vector needs to be restored to it's original state, use this storage location for it's original value. The patches should be done in this manner so the applications doing the patches will always return the system to it's original state no matter what past, present or future Pascal version it is patching.

E. For CONSOLE: init to be used from assembly routines the locations of SYSCOM and the BREAK routine have to be available. The CONINIT routine requires these on the stack. Pointers to SYSCOM and BREAK will be stored by the interpreter boot in a PERMANENT location in the BF00 page (see locations section).

F. Since the old 1.0 release, the code to jump to the CONCK routine has been set up at location BF0A. Anyone wishing to get to the CONCK routine should do a JSR BF0A as this will always get them there no matter where the CONCK routine really is. The keypress function has been changed to conform to this new convention but it will use the old convention if it is working from within an old system. Do not try to get to CONCK in this way from within an ATTACHED driver as you will loose your return address to Pascal. See ATTACH part of this document for how to get to CONCK from an attached driver.

G. There is now a version byte so one can tell which version (1.0, 1.1, etc.) of Apple Pascal he is working with. There is also a flavor byte to tell one which flavor of this version he has (regular, runtime, runtime without sets, etc.). (see locations section)

11. Whenever SYSTEM.ATTACH is used, it will make a copy of the original BIOS jump vector (the after fold vector that has the actual driver addresses in it) and put this below the heap with the drivers that are attached. It will leave a pointer to this copy of the vector at location 00E2. You can use this vector in you drivers to get to the standard Apple drivers for any device. This way you can define a driver that does something above and

beyond the standard Apple driver yet this new driver can still make use of the standard Apple driver. See the ATTACH part of this document for more information.

12. In the RSP are two vectors that tell the RSP what is legal (input &-or output) for a particular character orientated device (CONSOLE:, REMOTE: & PRINTER:). For example it tells the RSP that it is illegal to read from the PRINTER:. If you wanted to ATTACH a PRINTER: driver so you could read from the PRINTER:, you would have to change this vector. 00E4 points to the READTBL vector and 00E6 to the WRITTBL vector. Let's take the READTBL for an example:

```
READTBL     ;table of routine addresses to be called when
            ;writing to that unit (disk I/O does not use
            ;this table).
            ;an entry=0 means that the operation is illegal
            ;for that unit.
            .WORD    BIOS+CONREAD    ;unit 1
            .WORD    BIOS+CONREAD    ;unit 2
            .WORD    0               ;unit 3
            .WORD    0               ;4 & 5 are disk units
            .WORD    0
            .WORD    0               ;6 is PRINTER:
            .WORD    BIOS+REMREAD    ;unit 7
            .WORD    0               ;8 is rem write which has
                                     ;an address in the WRITTBL
```

Here BIOS refers to the base of the jump vector before the fold and CONREAD is the offset off the base of that vector to get to the jump to the CONSOLE: read routine (for CONSOLE: read the offset is 0, for CONSOLE: write it's 3, etc). The value for BIOS is the pointer stored in location 00EC mentioned in the locations section below.

LOCATIONS.

These are the locations of new system permanents mentioned in this document, all pointers are set up by the system and are stored low byte first. Do not modify what is stored in these pointers (except for SPCHAR if you want to suppress special character checking) since the system uses this information too. These locations are defined to have the same function & remain in the same place for future versions of Apple II Pascal.

BF1C	SPCHAR	(To control special chars)
BF1D	IBREAK	(Set by boot in interp for assembly calls to CONINIT)
BF1F	ISYSCOM	('')
BF21	VERSION	(1 byte Version # of system, =2 for the new release, 0 for the old 1.0 release)
BF22	FLAVOR	(This byte tells which flavor [runtime,regular, etc.] of this VERSION you are dealing with) The encoding is: 1 -->regular system

runtime versions:
2 -->LC-ALL (LC- means no language card)
3 -->LC-no sets
4 -->LC-no floating point
5 -->LC-no sets or floating point
6 -->LC+ALL
7 -->LC+no sets
8 -->LC+no floating point
9 -->LC+no sets or floating point

This flavor byte is 0 in the old 1.0 release.

BFC0-BFFF	BDEVBUF	(Area for non Apple boot devices, like the CORVUS)
00E2	ACJVAFLD	(Pointer to ATTACH copy of the original Jump Vector after the fold)
00E4	RTPTR	(Pointer to READTBL)
00E6	WTPTR	(Pointer to WRITTBL)
00E8	UDJVP	(Pointer to user device jump vector)
00EA	DISKNUMP	(Pointer to disknum vector)
00EC	JVBFOLD	(Pointer to jump vector before fold)
00EE	JVAFOLD	(Pointer to jump vector after fold)
FFF6		(Version word which = 1 for version 1.0 and = 0 for version 1.1 This version word should not be used at runtime to tell which version you have. For that use the version byte mentioned above. This word should only be used by software that wants to see which SYSTEM.APPLE it is dealing with by looking at the contents of this word in the SYSTEM.APPLE file when it is not loaded in memory)
FFF8		(Start vector)
FFFA		(NMI non maskable interrupt vector)
FFFC		(RESET vector)
FFFE		(IRQ interrupt request vector)

The locations and code in the 1.0 'PRELIMINARY APPLE PASCAL GUIDE TO INTERFACING FOREIGN HARDWARE' BIOS document are not the same for Apple Pascal 1.1 and that document clearly stated we would not commit ourselves to keeping them the same.

Pascal 1.1 Firmware Card Protocol

One major problem with Apple Pascal 1.0 is the way it deals with peripheral cards. It was set up to work with the four peripheral cards that Apple supported at the time of its release (the disk,communciations,serial and parallel cards) and had no mechanism for interfacing any other devices. Since Apple as well as many other vendors continue to produce new peripherals for the Apple][, a new protocol was designed and implemented in the Pascal 1.1 BIOS which allows new peripheral cards to be introduced to the system in a consistent and transparent fashion. The new protocol is called the "firmware card" protocol since the BIOS deals with these cards by making calls to their firmware at entry points defined by a branch table on the card

itself. The new protocol fully supports the Pascal typeahead function and KEYPRESS will work with firmware cards used as CONSOLE devices. The following paragraphs describe the firmware card protocol in full detail.

A firmware card may be uniquely identified by a four byte sequence in the card's $CN00 ROM space. Location $CN05 must contain the value $38 and location $CN07 must contain $18. Note that these are identical to the Apple Serial Card. A firmware card is distinguished from a serial card by the further requirement that location $CN0B must contain the value $01. This value is called the "generic signature" since it is common to all firmware cards. The value at the next sequential location, $CN0C, is called the "device signature" since it uniquely identifies the device.

The device signature byte is encoded in a meaningful way. The high order 4 bits specify the class of the device while the low order four bits contain a unique number to distinguish between specific devices of the same class. The appendix to this document defines some device class numbers; in any case vendors should contact Apple Technical Support to make sure they use a unique number for their device signature. Although the device signature is ignored by the 1.1 BIOS, it may be used by applications programs to identify specific devices.

Following the 2 signature bytes is a list of four entry point offsets starting at address $CN0D. These four entry points must be supported by all firmware cards. They are the initialization, read, write and status calls. The BIOS takes care of disabling the $C800 ROM space of all other cards before calling the firmware routines.

The offset to the initialization routine is at location $CN0D. Thus, if $CN0D contains XX, the BIOS will call $CNXX to initialize the card. On entry, the X register contains $CN (where N is the slot number) and the Y register contains $N0. On exit, the X register should contain an error code, which should be 0 if there was no error. This error code is passed on to the higher levels of the system in the global variable "IORESULT". Registers do not have to be preserved.

The offset to the read routine is at location $CN0E. On entry, the X register will contain $CN and the Y register will contain $N0. On exit, the A register should contain the character that was read while the X register contains the IORESULT error code. The A and Y registers do not have to be preserved.

The offset to the write routine is at location $CN0F. On entry, the A register contains the character to be written while the X register contains $CN and the Y register contains $N0. On exit the X register should contain the IORESULT error code (which should be 0 for no error). The A and Y registers do not have to be preserved.

The offset to the status routine is at location $CN10. On entry, the X register contains $CN and the Y register contains $N0 while the A register contains a request code. If the A register contains 0, the request is "are you ready to accept output?". If the A register contains 1, the request is "do you have input ready for me?". On exit, the driver returns the IORESULT error code in the X register and the results of the status request in the carry bit. The carry clear means "false" (i.e., no, I don't have any input for you), while the carry set means true. Note that the status call must preserve the Y register but does not have to preserve the A register.

Thus, sample code for the first few bytes of a firmware card's $CN00 space should look something like:

```
BASICINIT       BIT     $FF58           ;set the v-flag
                BVS     BASICENTRY      ;always taken
IENTRY          SEC                     ;BASIC input entry point
                DFB     $90             ;opcode for BCC
OENTRY          CLC                     ;BASIC output entry point
                CLV
                BVC     BASICENTRY      ;Always taken
;
;  Here is the Pascal 1.1 Firmware Card Protocol Table
;
                DFB     $01             ;Generic signature byte
                DFB     $41             ;Device signature bye
;
PASCALINIT      DFB     >PINIT          ; > means low order byte
PASCALREAD      DFB     >PREAD          ;offset to read
PASCALWRITE     DFB     >PWRITE         ;offset to write
PASCALSTATUS    DFB     >PSTATUS        ;offset to status routine
```

The above code fulfils all the requirements for both the BASIC and Pascal 1.1 I/O protocols. The routines PINIT, PREAD, etc, are probably jumps into the card's $C800 space which is already properly enabled by the BIOS. The reason the $CN00 space was chosen for the protocol (as opposed to the $C800 space) is that the BASIC protocol requires that all cards have $CN00 ROM space while some smaller cards may not need any $C800 ROM space.

The firware card protocol includes 2 optional calls that do not have to be implemented but would be kind of nice. The BIOS checks location $CN11 to determine if the optional calls are present; if that location contains a $00 then the BIOS thinks the calls are implemented. Thus if your card does not implement the optional calls, you should ensure that $CN11 contains a non-zero value. The two optional calls are a control call pointed to by $CN12 and an interrupt handler call pointed to by $CN13.

The control call entry point is specified by the offset at $CN12. On entry, the X register contains $CN, the Y register contains $N0 and the A register contains the control request code. Control requests are defined by the device. On exit the X register should contain the IORESULT error code.

The interrupt poll entry point is specified by the offset at $CN13. On entry, the X register contains $CN and the Y register contains $N0. The interrupt poll routine should poll the card's hardware to determine if it has a pending interrupt; if it does not it should return with the carry clear. If it does, it should handle the interrupt (including disabling it) and return with the carry set. Also, the X register should contain the IORESULT error code which should be 0 if there was no error. An interrupt polling routine must be careful not to clobber any zero page or screen space temporaries.

The control and interrupt requests are not implemented in the Pascal 1.1 BIOS but it would be nice to support them if possible as they may be implemented in later versions of the Pascal BIOS as well as other forthcoming operating system environments for the Apple][.

Note that the firmware card signature is a superset of the Apple serial card signature as recognized by the Pascal 1.0 BIOS. This allows a firmware card to function with both Pascal 1.0 and Pascal 1.1. If a card wishes to work with Pascal 1.0 as a "fake" seral card, it must provide an input entry point at $C84D and an output entry point at $C9AA. Note that since Pascal 1.0 will think the card is a serial card, typeahead and KEYPRESS capabilities will be lost.

Additional Notes

1. The Pascal RSP expects the high order bit of every ASCII character it receives from the Console read routine to be clear. The RSP will not do this for you; you must ensure the high bit of all text your card passes to the RSP from the console read routine is clear.

2. Zero page locations $00 to $35 may be used as temporaries by your firmware, as are the slot 0 screen space locations ($478,$4F8, etc.). In general, peripheral card firmware should be as conservative as possible in their memory usage, preserving zero page contents whenever possible. An interrupt polling routine must not destroy these or any other memory locations.

3. Location $7F8 must be set up to contain the value $CN, where N is the slot number, if your card utilizes the $C800 expansion ROM space. The BIOS does not do this for you; his must be done if you want your card to function in an interrupting environment.

4. The firmware card status routine should be as quick as possible, as it may be called from within the I/O polling loops of many other peripherals if your card is being used as the console device. In no case should the status routine take longer than 100 milliseconds.

5. A firmware card in slot 1 is automatically recognized as the volume "PRINTER:". A firmware card in slot 2 is automatically recognized as the volumes "REMIN:" and "REMOUT:". A firmware card in slot 3 is automactically recognized as the volumes "CONSOLE:" and "SYSTERM:".

APPENDIX

The following numbers correspond to device classes used in the device signature code. Make sure you contact Apple Technical Support to ensure that you have a unique device signature code.

0 -- reserved
1 -- printer
2 -- joystick or other X-Y input device
3 -- I/O serial or parallel card
4 -- modem
5 -- sound or speech device
6 -- clock
7 -- mass storage device

8 -- 80 column card
9 -- Network or bus interface
10 -- Special purpose (none of the above)

11 through 15 are reserved for future expansion

Additional Information

1. The type ahead buffer is located at $03B1 hex and is $4E hex in length. It is implemented with a read pointer (RPTR at BF18 hex) and a write pointer (WPTR at $BF19 hex). At CONSOLE: init time, these should both be set to 0. When a character is detected by CONCK, the WPTR is incremented then compared with $4E. If it is equal to $4E, it is set to $0 (this is a circular buffer). Then the WPTR is compared with RPTR and if they are equal the buffer is full. If the buffer is not full, the character is stored at $03B1+the value in WPTR.

   When removing a character from the type ahead buffer, use the following sequence. Compare the RPTR with WPTR and if they are equal, the buffer is empty and you must wait until a character is available from the keyboard. If they are not equal, increment the RPTR and compair it to $4E. If it equals $4E, set it to $0. Now get the character from location $03B1+the value in RPTR.

   If you are implementing your own type ahead, you can do it however you wish. This information is made available in case you want to check for input from another device as well as the standard system CONSOLE: and have characters from that device be put in the system type ahead buffer.

2. The example drivers in this document did not show the setting of the IORESULT in the X register. This would be done in the code specific to your driver and should allways be set to something (0 if there are no errors). If there are errors, set it as described elsewhere in this document and the Pascal Manuals.

3. For further information, see the newest edition of the Apple II Reference Manual.

4. These listings from the BIOS are included to show you how we implemented certain system drivers. You cannot rely on the locations of these to stay in the same place in the BIOS in future releases of Apple II Pascal nor can you rely on the routines themselves staying the same. They are only included as examples and to give you information that may not be documented elsewhere. This is not a complete BIOS listing so you may find references to routines or locations that are not included in this listing. The only locations that will be sure to remain the same for future releases are those mentioned in the LOCATIONS section above. We are against you poking the BIOS yourself to change or overwrite any of these routines. We did not include this information so you could poke the BIOS. If you do modify the BIOS, it is completely at your own risk! We have provided the ATTACH utility so you can add your own drivers the system without poking the BIOS and this is the way it should be done! If you have special requirements that are not solved by ATTACH, please

contact Apple Technical Support.

```
;---------------------------------------
;
; ZERO PAGE PERMANENTS
;
;---------------------------------------
FIRST       .EQU 0F0            ;START ZERO PAGE USE
BAS1L       .EQU FIRST          ;SCREEN 1 PTR
BAS1H       .EQU FIRST+1
BAS2L       .EQU FIRST+2        ;SCREEN 2 PTR
BAS2H       .EQU FIRST+3
CH          .EQU FIRST+4        ;HORIZ CURSOR, 0..79
CV          .EQU FIRST+5        ;VERT CURSOR, 0..23
TEMP1       .EQU FIRST+6
TEMP2       .EQU FIRST+7
SYSCOM      .EQU FIRST+8        ;2 BYTES PTR TO SYSCOM AREA

;---------------------------------------
;
; BF00 PAGE PERMANENTS
;
;---------------------------------------
CONCKVECTOR .EQU 0BF0A          ;4 BYTES
SCRMODE     .EQU 0BF0E
LFFLAG      .EQU 0BF0F
NLEFT       .EQU 0BF11
ESCNT       .EQU 0BF12
RANDL       .EQU 0BF13
RANDH       .EQU 0BF14
CONFLGS     .EQU 0BF15
BREAK       .EQU 0BF16          ;2 BYTES
RPTR        .EQU 0BF18          ;1 BYTE
WPTR        .EQU 0BF19          ;1 BYTE
RETL        .EQU 0BF1A
RETH        .EQU 0BF1B
SPCHAR      .EQU 0BF1C          ;00 MEANS DO ALL SPECIAL CHARACTER CHECKING
                                ;01 MEANS DON'T CHECK FOR APPLE SCREEN STUFF
                                ;02 MEANS DON'T CHECK FOR OTHER SCREEN STUFF
IBREAK      .EQU 0BF1D          ;INTERP STORES BREAK & SYSCOM ADR HERE FOR
ISYSCOM     .EQU 0BF1F          ;USER ROUTINES TO GET AT
VERSION     .EQU 0BF21          ;VERSION OF SYSTEM SET TO 2 FOR APPLE 1.1
FLAVOR      .EQU 0BF22          ;SEE TABLE IN INTERP BOOT
SLTTYPS     .EQU 0BF27          ;BF27..0BF2E
XITLOC      .EQU 0BF2F          ;INTERP INITS THIS TO LOCATION OF XIT
            ;FORTRAN PROTECTION USES BF56..BF7F
            ;VENDOR BOOT DEVICES CAN USE BFC0..BFFF

;---------------------------------------
;
; MISCELANEOUS PROGRAM EQUATES
;
;---------------------------------------
BUFFER      .EQU 0200           ;TEMP HSHIFT BUFFER (OVERLAPS DISK BUF)
CONBUF      .EQU 03B1           ;78 CHAR TYPE-AHEAD BUF
CBUFLEN     .EQU 04E            ;78 DECIMAL
NCTRLS      .EQU 14.            ;# CTRL CHARS IN TABLE
SIGVALUE    .EQU 1
```

```
BYTEPSEC    .EQU 256.           ;DISK INFO FOR DISKSTAT
SECPTRAK    .EQU 16.
TRAKPDSK    .EQU 35.
UDJVP       .EQU 0E8            ;0 PAGE JUMP VECTOR POINTER LOCATIONS
DISKNUMP    .EQU 0EA
JVBFOLD     .EQU 0EC
JVAFOLD     .EQU 0EE
HCMODE      .EQU 0E1            ;THESE TWO BYTES USED FOR HIRES STUFF
HSMODE      .EQU 0E0


JVECTRS     .WORD       UDJMPVEC
            .WORD       DISKNUM
            .WORD       BIOS
            .WORD       BIOSAF

;---------------------------------------
;
; HARD RESET INITIALIZATION
;
;---------------------------------------
START       CLD                 ;SET HEX MODE
            SEI                 ;MAKE SURE INTERRUPTS ARE OFF.

;---------------------------------------
;
; CLEAR ALL MEMORY 0 TO BFFF
; (RUN-TIME SYSTEM:0 TO TOPMEM + BF PAGE);
;
;---------------------------------------
            LDA #0
            STA ZEROL
            STA ZEROH
            TAY
            TAX
ZERLP       STA (ZEROL),Y       ;WRITE A BYTE OF 0
            INY                 ;BUMP POINTER
            BNE ZERLP           ;LOOP TILL NEXT PAGE
            INC ZEROH           ;BUMP MSB POINTER
            INX
            .IF RUNTIME=1
            CPX #TOPMEM         ;DONE CLEARING MEM?
            BNE $1
            LDX #0BF            ;CLEAR BF PAGE
            STX ZEROH
$1:         CPX #0C0
            BNE ZERLP
            .ELSE
            CPX #0C0            ;DONE CLEARING BFXX?
            BNE ZERLP
            .ENDC

;-----------------------
;
; CHECKSUM PROMS ON EACH SLOT
; TO FIND OUT WHO'S OUT THERE
;
; SUM TWICE TO TELL IF CARD THERE
```

```
; IF SUMS DONT MATCH THEN NO PROM IS THERE
; IF MS BYTE OF SUM=0 THEN NO PROM IS PRESENT
;
;-----------------------
            LDY #0C7            ;POINT TO SLOT 7 PROM
NXTCRD      STY CKPTRH          ;(CKPTRL=0 FROM MEM CLEAR)
            JSR CKPAGE          ;16 BIT SUM IN X,A
            STA CHECKL
            STX CHECKH          ;SAVE FOR MATCH
            JSR CKPAGE          ;SUM AGAIN
            CPX #0              ;WAS MSB ZERO?
            BEQ NOPROM          ;YES NO PROM ON CARD
            CMP CHECKL          ;LSB MATCH?
            BNE NOPROM          ;NO, NO PROM ON CARD
            CPX CHECKH
            BNE NOPROM          ;MSB DIDNT MATCH
            BEQ SKIPIORTS       ;ALWAYS TAKEN

;---------------------------
;
; TABLE OF CN05 AND CN07 BYTES OF EACH CARD
;
;---------------------------
CN05BYTS    .BYTE 003,018,038,048
CN07BYTS    .BYTE 03C,038,018,048

;---------------------------
;
; NOW THAT WE KNOW A CARD IS THERE,
; EXAMINE GN05 AND CN07 BYTE TO
; DETERMINE WHICH CARD IT IS
;
; SET CARDTYPE AS FOLLOWS:
; 0=CKSUM NOT REPEATABLE OR MSB=0
; 1=CKSUM REPEATABLE,CARD NOT RECOGNIZED
; 2=DISK CARD (BYTE 07= 03C)
; 3=COM CARD  (BYTE 07= 038)
; 4=SERIAL    (BYTE 07= 018)
; 5=PRINTER   (BYTE 07= 048)
; 6=FIRMWARE  (BYTE 07= 048)
;----------------------------
SKIPIORTS   LDX #5              ;4 TYPES OF CARDS
NXTYP       LDY #5              ;CHECK BYTE CN05 OF CARD
            LDA (CKPTRL),Y
            CMP CN05BYTS-2,X    ;MATCH TABLE?
            BNE TRYNXT          ;NO, TRY NEXT IN LIST
            LDY #7
            LDA (CKPTRL),Y      ;TEST CN07 BYTE
            CMP CN07BYTS-2,X    ;MATCH TABLE?
            BEQ STOR            ;BOTH MATCHED, CARD RECOGNIZED
TRYNXT      DEX                 ;BUMP TO NEXT IN LIST
            CPX #2              ;TRY ALL TYPES IN LIST
            BCS NXTYP           ;IF NOT IN LIST,FALL THRU WITH X=1
STOR        CPX #4              ;IS IT A SERIAL CARD?
            BNE STOR1
            LDY #0B
            LDA (CKPTRL),Y
            CMP #SIGVALUE
```

```
            BNE STOR1
            LDX #6
STOR1       LDY CKPTRH
            TXA
            STA SLTTYPS-0C0,Y
NOPROM      LDY CKPTRH
            DEY                     ;BUMP TO NEXT LOWER SLOT
            CPY #0C0                ;SLOTS 7 DOWNTO 1 DONE?
            BNE NXTCRD              ;LOOP TILL 7 SLOTS DONE
                                    ;LEAVE WITH Areg:=0
;---------------------------------------
;
; SET UP CONCK VECTOR FOR KEYPRESS FUNCTION
;
;---------------------------------------
            BEQ $2                  ;ALWAYS BRANCHES
$1          JSR KCONCK              ;HERE ARE THE 2 INSTRUCTIONS TO BE TRANSFERRED
            RTS
$2          LDY #3                  ;TRANSFER 4 BYTES TO BFOA
$21         LDA $1,Y
            STA CONCKVECTOR,Y
            DEY
            BPL $21

            ;SET UP JUMP VECTOR POINTERS IN 0 PAGE
            LDY #7
$3          LDA JVECTRS,Y
            STA UDJVP,Y
            DEY
            BPL $3

;---------------------------------------
;
; SET SCREEN MODE ETC
;
;---------------------------------------
            LDA #80
            STA HCMODE
            LDA 0C051               ;SET TEXT MODE
            LDA 0C052               ;SET BOTTOM 4 GRAFIX
            LDA 0C054               ;SELECT PRIMARY PAGE
            LDA 0C057               ;SELECT HIRES GRAFIX
            LDA 0C010               ;CLEAR KEYBOARD STROBE
            JSR FORM                ;ERASE SCREEN
            JSR INVERT              ;PUT CURSOR ON SCREEN
            JSR DRESET              ;DO ONCE ONLY DISK INIT
            LDA SLTTYPS+3           ;WHAT CARD IN SLOT 3?
            LDY #030                ;SLOT 3
            JSR GENIT               ;SET BAUD IF COM OR SER THERE
            CPX #0                  ;WAS AN EXTERNAL CONSOLE THERE?
            BNE STARTUP             ;NO,USE APPLE SCREEN
            LDA #4
            STA SCRMODE             ;SET BIT 2 FOR EXT CON
STARTUP     JMP JPASCAL             ;FOLD IN INTERP AND START PASCAL

;-------------------------
;
; SUB TO CHECKSUM ONE PAGE
```

```
;
CKPAGE      LDA #0
            TAX                 ;CLEAR SUM
            TAY                 ;CLEAR INDEX
CKNX        CLC
            ADC (CKPTRL),Y      ;ADD BYTE
            BCC NOCRY
            INX                 ;INC HI BYTE IF CARRY
NOCRY       INY                 ;BUMP INDEX
            BNE CKNX            ;SUM 256 BYTES
            RTS                 ;RETURN SUM IN X,A AND Y=0

;-----------------------------------------------------------
;
;  BIOS HANDLERS FOR LOGICAL AND PHYSICAL DEVICES.
;
;-----------------------------------------------------------


;---------------------------------------
;
; CONSOLE CHECK FOR CHAR AVAIL
; STATUS AND ALL REGS PRESERVED
; IF CHAR AVAIL,PUT IN CONBUF AND INC WPTR.
;
; WARNING...THIS ROUTINE ALSO CALLED FROM DISK ROUTINES
;
;---------------------------------------
CONCK       PHP
            PHA
            TXA
            PHA
            TYA
            PHA
RNDINC      INC RANDL.          ;BUMP 16 BIT RANDOM SEED
            BNE RNDOK
            INC RANDH
RNDOK       LDA SLTTYPS+3       ;WHAT CARD IS IN SLOT 3?
            CMP #3              ;IS IT A COM CARD?
            BEQ COMCK           ;YES,GO CHECK IT
            CMP #4              ;IS IT A SERIAL CARD?
            BEQ JDONCK          ;YES,IT CANT BE TESTED
            CMP #6
            BEQ FIRMCK
TSTKBD      LDA 0C000           ;TEST APPLE KEYBOARD
            BPL JDONCK          ;NO CHAR AVAIL
            STA 0C010           ;CLEAR KEYBD STROBE
            AND #07F            ;MASK OFF TOP BIT
            TAX                 ;See if checking for apple special chars is
            LDA SPCHAR          ;turned off.
            ROR A
            BCS NOTFOLP2        ;Jump if so
            TXA
            CMP #11.            ;CTRL-K?
```

```
            BNE NOTK
            LDA #05B            ;YES,REPLACE WITH LEFT SQR BRACKETT
NOTK        CMP #1              ;CTRL-A?
            BNE NTTAB
            JSR HTAB            ;YES,TAB NEXT MULT 40
            LDA CONFLGS
            AND #0FE
            STA CONFLGS         ;CLEAR AUTO-FOLLOW BIT
            JMP DONECK
NTTAB       CMP #26.            ;CTRL-Z?
            BNE NOTFOL          ;NO,PUT CHAR IN BUFFER
            LDA CONFLGS
            ORA #1
            STA CONFLGS         ;SET AUTO-FOLLOW BIT
            BNE DONECK          ;BR ALWAYS


COMCK       LDA 0C0BE           ;CHAR AVAIL?
            LSR A
            BCC DONECK          ;NO CHAR AVAIL
            LDA 0C0BF           ;GET CHAR FROM UART
GOTCHAR     AND #07F            ;MASK OFF BIT 7
NOTFOL      TAX
            LDA SPCHAR          ;See if console special char checking is
                                ;turned off.
            ROR A
NOTFOLP2    ROR A
            BCS NFMI1           ;Jump if so
            TXA
            LDY #055
            CMP (SYSCOM),Y      ;STOP CHAR?
            BNE NOTSTOP
            LDA CONFLGS
            EOR #080
            STA CONFLGS         ;YES,TOGGLE STOP BIT (BIT 7)
JDONCK      JMP DONECK

FIRMCK      LDA #1
            LDY #030
            JSR FIRMSTATUS
            BCC DONECK
            JSR FREAD1
            JMP GOTCHAR

NOTSTOP     DEY
            CMP (SYSCOM),Y
            BNE NOTBRK
            LDA CONFLGS
            AND #03F
            STA CONFLGS         ;CLEAR FLUSH&STOP BITS
            .IF RUNTIME=0
             JMP TOBREAK
            .ELSE
             JMP @BREAK          ;BREAK OUT
            .ENDC
NOTBRK      DEY
            CMP (SYSCOM),Y      ;FLUSH?
            BNE NOTFLUS
```

```
            LDA CONFLGS
            EOR #040
            STA CONFLGS         ;TOGGLE FLUSH BIT (BIT 6)
            JMP DONECK

NFMI1       TXA
NOTFLUS     LDX WPTR
            JSR BUMP
            CPX RPTR            ;BUFFER FULL?
            BNE BUFOK
            JSR BELL
            JMP DONECK          ;BEEP&IGNORE CHAR
BUFOK       STX WPTR
            STA CONBUF,X        ;PUT CHAR IN BUFFER
DONECK      BIT CONFLGS         ;IS STOP FLAG SET?
            BPL CKEXIT
            JMP RNDINC          ;LOOP IF IN STOP MODE

CKEXIT      PLA
            TAY
            PLA
            TAX
            PLA
            PLP
            RTS                 ;ELSE RESTORE STAT AND ALL REG AND RETURN
BUMP        INX                 ;BUMP BUFFER POINTER WITH WRAP-AROUND
            CPX #CBUFLEN
            BNE BMPRTS
            LDX #0
BMPRTS      RTS


;-----------------------------------
;
; INITIALIZE CONSOLE:
;
;-----------------------------------
CINIT       PLA
            STA TEMP1           ;SAVE RETURN ADDR
            PLA
            STA TEMP2
            PLA
            STA SYSCOM          ;SAVE PTR TO SYSCOM AREA
            PLA
            STA SYSCOM+1
            PLA
            STA BREAK           ;SAVE BREAK ADDRESS
            PLA
            STA BREAK+1
            LDA TEMP2
            PHA                 ;RESTORE RETURN ADDR
            LDA TEMP1
            PHA
            LDA RPTR            ;FLUSH TYPE-AHEAD BUFFER
            STA WPTR
            LDA CONFLGS
            AND #03E
```

```
            STA CONFLGS         ;CLEAR STOP,FLUSH,AUTO-FOLLOW BITS
            JSR TAB3            ;NO,HORIZ SHIFT FULL LEFT
CINIT2      LDX #0              ;CLEAR IORESULT
            RTS                 ;AND RETURN

;----------------------------------
;
; READ FROM CONSOLE:
; KEYBOARD,COM OR SERIAL CARD IN SLOT 3
;
;----------------------------------
CREAD       JSR ADJUST          ;HORIZ SCROLL IF NECESSARY
            LDY #030            ;SLOT 3
            LDA SLTTYPS+3       ;WHAT TYPE OF CARD?
            CMP #4              ;IS IT A SERIAL CARD?
            BNE CREAD2          ;NO,CONTINUE
            JSR RSER            ;YES, READ IT
            AND #7F             ;MASK OFF TOP BIT
            RTS
CREAD2      JSR CONCK           ;TEST CHAR
            LDX RPTR
            CPX WPTR
            BEQ CREAD           ;LOOP TILL SOMETHING IN BUFFER
            JSR BUMP
            STX RPTR            ;BUMP READ POINTER
            LDA CONBUF,X        ;GET CHAR FROM BUFFER
            LDX #0              ;CLEAR IORESULT
            RTS                 ;AND RETURN TO PASCAL

;----------------------------------
;
; INITIALIZE PRINTER:
; PRINTER IS ALWAYS IN SLOT 1
; IT MAY BE A PRINTER,COM,OR SERIAL CARD
;
;----------------------------------
PINIT       LDY #010            ;SLOT 1 ; 010
            LDA SLTTYPS+1       ;WHAT CARD IN SLOT 1?
            CMP #5              ;PRINTER CARD?
            BEQ CLRIO1          ;YES,NO INIT NEEDED
GENIT       CMP #4              ;SERIAL CARD?
            BEQ ISER            ;YES, INIT SER CARD
            CMP #3              ;COM CARD?
            BEQ ICOM            ;YES,INIT COM CARD
            CMP #6
            BEQ FIRMINIT
            LDX #9              ;NONE OF ABOVE,OFFLINE
            RTS

FIRMINIT    PHA
            JSR SER1
            LDY #0D
FVEC1       LDA (TEMP1),Y
            STA TEMP1
            LDY 6F8
            PLA
            JMP @TEMP1
```

```
;-----------------------------------
;
; INITIALIZE REMOTE:
; REMOTE IS ALWAYS IN SLOT 2
; IT MAY BE A COM OR SERIAL CARD
;
;-----------------------------------
RINIT       LDA SLTTYPS+2        ;WHAT CARD IN SLOT 2?
            LDY #020
            BNE GENIT            ;BR ALWAYS TAKEN

;-----------------------------------
;
; INIT COM CARD, Y=ONO
;
;-----------------------------------
ICOM        LDA #3               ;MASTER INIT
            STA 0C08E,Y          ;TO STATUS
            LDA #21.
            STA 0C08E,Y          ;SET BAUD ETC
CLRIO1      LDX #0               ;CLEAR IORESULT
            RTS                  ;AND RETURN

;-----------------------------------
;
; INIT SERIAL CARD, Y=ONO
;
;-----------------------------------
ISER        JSR SER1             ;ASSORTED GARBAGE
            JSR 0C800            ;SET UP SLOT DEPENDENTS
CLRIO3      LDX #0               ;CLEAR IORESULT
            RTS                  ;AND RETURN

;-----------------------------------
;
; ASSORTED SERIAL CARD SET-UP
;
;-----------------------------------
SER1        STY 06F8             ;STORE NO
            TYA
            LSR A
            LSR A
            LSR A
            LSR A
            ORA #0C0
            TAX                  ;MAKE OCN IN X
            LDA #0
            STA TEMP1
            STX TEMP2            ;SET UP INDIRECT ADDRESS
            LDA 0CFFF            ;TURN OFF ALL C8 ROMS
            LDA (TEMP1),Y        ;SELECT C8 BANK
            RTS

;-----------------------------------
;
; WRITE TO CONSOLE:
; VIDEO SCREEN,COM OR SER CARD IN SLOT 3
;
```

```
;-----------------------------------
CWRITE      JSR CONCK           ;CONSOLE CHAR AVAIL?
            BIT CONFLGS         ;IS FLUSH FLAG SET?
            BVS CLRIO           ;YES,DISCARD CHAR & RETURN
            TAX                 ;SAVE CHAR IN X
            LDY #030            ;SLOT 3;010
            LDA SLTTYPS+3       ;WHAT KIND OF CARD?
            CMP #3              ;COM CARD?
            BEQ WCOM            ;YES WRITE TO COM CARD SLOT 3
            CMP #4              ;SERIAL CARD?
            BEQ WSER            ;YES,WRITE TO SER CARD SLOT 3
            CMP #6
            BEQ WFIRM
            TXA                 ;ELSE RESTORE CHAR & SEND TO SCREEN
VIDOUT      STA TEMP1           ;SAVE CHAR FOR LATER
            JSR INVERT          ;REMOVE CURSOR
            LDY CH
            JSR VOUT2           ;DO THE BUSINESS
            JSR INVERT          ;RESTORE THE CURSOR
CLRIO       LDX #0              ;CLR IORESULT
            RTS                 ;RETURN FROM VIDOUT

WFIRM       TXA
            PHA
            LDA #0
            JSR IOWAIT
            JSR SER1
            LDY #0F
            JMP FVEC1

;-----------------------------------
;
; WRITE TO SERIAL CARD, Y=ONO,CHAR IN X
;
;-----------------------------------
WSER        JSR CONCK           ;CONSOLE CHAR?
            TXA
            PHA                 ;SAVE CHAR ON STACK
            JSR SER1            ;ASSORTED GARBAGE
            PLA
            STA 05B8,X          ;SET UP DATA BYTE
            JSR 0C9AA           ;SEND IT (SHOUT)
            LDX #0
            RTS

;-----------------------------------
;
; WRITE TO REMOTE:, CHAR IN A
;
;-----------------------------------
RWRITE      TAX                 ;SAVE CHAR
            LDA SLTTYPS+2       ;WHAT CARD IN SLOT 2?
            LDY #020
            BNE GENW2           ;BR ALWAYS TAKEN

;-----------------------------------
;
; WRITE TO PRINTER CARD SLOT1, CHAR IN X
```

```
;
;-----------------------------------
WPRN        JSR CONCK           ;CONSOLE CHAR AVAIL?
            LDA 0C1C1           ;TEST PRINTER READY
            BMI WPRN            ;LOOP TILL READY
            STX 0C090           ;SEND CHAR
CLRIO2      LDX #0
            RTS

;-----------------------------------
;
; WRITE TO COM CARD, Y=ONO,CHAR IN X
;
;-----------------------------------
WCOM        JSR CONCK           ;CONSOLE CHAR?
            LDA 0C08E,Y         ;TEST UART STATUS
            AND #2              ;READY?
            BEQ WCOM            ;NO,WAIT TILL READY
            TXA
            STA 0C08F,Y         ;SEND CHAR
            LDX #0
            RTS

;-----------------------------------
;
; WRITE TO PRINTER:, CHAR IN A
;
;-----------------------------------
PWRITE      TAX                 ;SAVE CHAR IN X
            LDA LFFLAG          ;TEST LINE-FEED FLAG
            BPL LFPASS          ;PASS IF BIT7=0
            CPX #10.            ;IS IT A LINE-FEED?
            BEQ CLRIO           ;YES,IGNORE
LFPASS      LDY #010            ;SLOT 1
            LDA SLTTYPS+1       ;WHAT KIND OF CARD?
GENW        CMP #5              ;PRINTER CARD?
            BEQ WPRN            ;YES WRITE TO PRINTER CARD
GENW2       CMPL#4              ;SERIAL CARD?
            BEQ WSER            ;YES WRITE TO SER CARD
            CMP #3              ;COM CARD?
            BEQ WCOM            ;YES WRITE TO COM CARD
            CMP #6
            BEQ WFIRM
OFFLINE     LDX #9
            RTS

;-----------------------------------
;
; READ FROM REMOTE:
;
;-----------------------------------
RREAD       LDA SLTTYPS+2       ;WHAT CARD IN SLOT 2?
            LDY #020
GENR        CMP #4              ;SERIAL CARD?
            BEQ RSER            ;GET FROM SER CARD
            CMP #3              ;COM CARD?
            BEQ RCOM            ;GET FROM COM CARD
            CMP #6
```

```
            BEQ RFIRM
            BNE OFFLINE          ;CARD NOT RECOG

;-----------------------------------
;
; READ FROM COM CARD, Y=NO
;
;-----------------------------------
RCOM        JSR CONCK            ;CHECK FOR CONSOLE CHAR
            LDA 0C08E,Y          ;TEST UART STATUS
            LSR A                ;TEST BIT 0
            BCC RCOM             ; WAIT FOR CHAR
            LDA 0C08F,Y          ;GET CHAR
            LDX #0
            RTS

RFIRM       LDA #1
            JSR IOWAIT
FREAD1      JSR SER1
            PHA
            LDY #0E
            JMP FVEC1

;-----------------------------------
;
; READ FROM SERIAL CARD, Y=0N0
;
;-----------------------------------
RSER        JSR CONCK            ;CONSOLE CHAR AVAIL?
            JSR SER1             ;ASSORTED GARBAGE
            JSR 0C84D            ;GET A BYTE (SHIFTIN)
            LDA 05B8,X           ;GET BYTE 0678+SLOT
            LDX #0
            RTS

FIRMSTATUS  PHA
            JSR SER1
            LDY #10
            JMP FVEC1

IOWAIT      JSR CONCK
            PHA
            JSR FIRMSTATUS
            PLA
            BCC IOWAIT
            RTS
```

APPLE COMPUTER, INC.
20525 Mariani Avenue
Cupertino, CA 95014

APPLE II PASCAL 1.2
ADDENDUM TO PASCAL TECHNICAL NOTE #11B

(December 1983)

For further information contact:
PCS Developer Technical Support
M/S 22-W, Phone (408) 996-1010

Disclaimer of All Warranties and Liabilities

Notice

APPLE COMPUTER, INC.
20525 Mariani Avenue
Cupertino, CA 95014

APPLE II PASCAL 1.2
ADDENDUM TO PASCAL TECHNICAL NOTE #11

(December 1983)

For further information contact:
PCS Developer Technical Support
M/S 22-W, Phone (408) 996-1010

Contents:

This document is meant to accompany Pascal Technical Note #11 - Apple Pascal 1.1 BIOS Reconfiguration Using Attach. It describes changes and additions to the Apple // Pascal 1.1 ATTACH facilities.

To use the software described in this addendum. you must have version 1.2 of Apple II Pascal.

ATTACH Software

This section describes the files on the ATTACH TOOLS disk which contains the ATTACH facilities provided for the Apple II Pascal system, version 1.2.

SYSTEM.ATTACH — attaches device drivers at startup time, using the information in ATTACH.DATA, and the driver code in the ATTACH.DRIVERS library. This version of SYSTEM.ATTACH is for use with the 64K and 128K Development Systems only. There is a special version for use with the Runtime Systems which is available on the Runtime System diskettes.

The following programs are provided for the creation and handling of the ATTACH.DATA file.

ATTACHUD.CODE — creates a version 1.2 ATTACH.DATA file from information supplied by the user

ADMERG.CODE — merges multiple version 1.2 ATTACH.DATA files into a single ATTACH.DATA file

CONVAD.CODE — converts an ATTACH.DATA file from version 1.1 to version 1.2

SHOWAD.CODE — shows the contents of a version 1.2 ATTACH.DATA file

The interfaces to the utilities ADMERG, CONVAD, and SHOWAD are self-explanatory, and we don't describe them in this addendum.

IM.CODE -- contains the interrupt manager (IM) for the 64K Pascal system

Changes to ATTACHUD

If you are familiar with the 1.1 version of ATTACHUD, you will find that the 1.2 version contains some additional prompts. After the question

> Do you want this driver aligned on
> a particular byte boundary? (Y/N)

ATTACHUD asks the new question

Do you want this driver to have a
transient initialization section? (Y/N)

If you respond with "Y", ATTACHUD will go on to ask you for the .PROC name of the transient initialization code, and its alignment requirements.

ATTACHUD also asks the new question

Will this driver use interrupts? (Y/N)

If you answer "Y" to this question, ATTACHUD will ensure that a record for the interrupt manager (IM) is present at the end of the ATTACH.DATA file.

Finally, note that unit numbers 13-20 are now available to user-defined devices. These numbers correspond to block-structured devices, and they must be controlled by user-written attach drivers.

Changes to the ATTACH Mechanism

Transient Initialization

As described in Pascal Technical Note #11, a device driver is attached at boot time. If the driver's data record (created by ATTACHUD) specifies that the driver should be initialized at startup time, then its initialization code is executed.

Under version 1.2, there is an additional step. The driver may be accompanied by a "transient initialization" module that is executed only at startup time.

After all the drivers are loaded onto the heap and initialized, each of the transients will be loaded and executed in the same order as their associated driver was loaded. They will overlay each other, going away after completion.

Each of the transients will have passed to it, on the stack, the address of the associated driver. This way communication can be set up between the two. Note that this is the address of the start of the driver, not start - 1.

In order to help data stucturing, the transient code may be loaded on a 0 to 256 byte boundary. Transients must be written in assembler, not use .ABSOLUTE (must be relocatable), and have a single .PROC at the beginning. The transient initialization code must be assembled as a separate module from the device driver itself. Like a device driver, it must be placed in the ATTACH.DRIVERS file using the LIBRARY utility.

Note that the transient initialization code is executed after the device driver's own (callable) initialization code is executed.

This facility was provided for the use of the Pascal ProFile Driver, but it is available to any user-defined device driver.

Interrupt Handling

Version 1.2 of Apple II Pascal supports interrupts from multiple devices.

The first part of this section describes interrupt handling on the Apple II. The second part discusses how to write a device driver that supports interrupts. A sample scheme for such a driver appears at the end of this section.

Important: The interrupt manager (IM) is shipped in the file IM.CODE. For the 64K Pascal systems, the IM driver must be placed in ATTACH.DRIVERS if any devices are to use interrupts. For the 128K Pascal systems, interrupt handling is built in, and the system will ignore the IM driver if it is present in ATTACH.DRIVERS.

The 48K runtime systems cannot use interrupts.

Handling Interrupts in Version 1.2

The main problem in handling interrupts is to save the context of the current program, and then restore that context once the interrupt has been processed. This includes saving the contents of various system registers, and restoring them once the driver returns.

When an interrupt can come from one of several devices, it is also necessary to identify which device, so that the appropriate driver can handle the interrupt.

A driver for a user device that supports interrupts must contain a section of code called the "interrupt service routine." This code will be called by the interrupt manager, as described below.

The interrupt manager (IM) itself is responsible for saving the current context and restoring it later. The interrupt service routines themselves are responsible for determining whether they should handle a given interrupt (just how they do this depends on the particular device; see below).

Interrupt service routines are set up in a linked chain (see item 3 in the following section). If an interrupt service routine recognizes an interrupt, it processes it and then returns to the IM. If the service routine doesn't recognize an interrupt, it transfers control to the next interrupt service routine in the chain. If none of the service routines claims an interrupt, then an error has occurred, and the system is restarted.

Thus, under this scheme, interrupts are handled in the following sequence.

- A device interrupt occurs. This disables interrupts and causes the processor to execute the code that starts at the address stored in the IRQ vector (located at $FFFE-FFFF).

- The IRQ points to the IM, which looks at the processor status on the stack and checks the break bit. If the break bit is set, the IM tranfers control

to the Pascal reset code which restarts the system.

- If the break bit is not set, the IM saves the current context and then transfers control to the first interrupt service routine in the chain.

- If the service routine doesn't recognize the interrupt, it transfers control to the next service routine in the chain. Otherwise, it processes the interrupt and then returns to the IM.

- If the last interrupt service routine in the chain doesn't recognize the interrupt, it transfers control to the reset code for the Pascal system.

- When the IM regains control, it restores the interrupted program's context which re-enables interrupts. Execution proceeds from the point at which it was interrupted.

A spurious interrupt can be generated as the result of a hardware malfunction, or of a BRK instruction in currently executing code. In the case of a hardware malfunction, the interrupt falls through the chain of routines, and control is ultimately passed to the Pascal system reset code. In the case of a BRK instruction, the break bit is set causing the IM to restart the system as decribed above.

To determine whether it should process an interrupt, an interrupt service routine can (in general) check the interrupt flag register for the appropriate card slot.

The location of the interrupt flag register, unfortunately, may vary according to the hardware; it is best if the peripheral card follows the conventions described in the Apple IIe Design Guidelines manual, in the section on "Peripheral Card Firmware."

For 64K Pascal systems, the code for the IM is in the form of an ATTACH driver. However, the IM cannot be called from a user program. (For 128K Pascal systems, interrupt handling is built in, and the IM code is ignored if it is present in ATTACH.DRIVERS).

User-written device drivers that support interrupts must also be ATTACH drivers. The following section discusses how to write such a driver.

Drivers for Devices with Interrupts

The following considerations must be taken into account when you write a driver for a device that generates interrupts.

1. Any volume number appropriate for a user-defined device (128-143) can be used, except for number 128 (decimal), which has been defined as the standard number for the large disk driver. The IM itself is assigned the highest available number. It is recommended that you use numbers in the 130-140 range.

   Note: If you use ADMERG, there is a chance of cancelling another driver that had already been installed with the same number, so it is important to

use SHOWAD to look at the ATTACH.DATA files before you run ADMERG.

2. SYSTEM.ATTACH enables interrupts after the full chain of interrupt service routines has been built and all transient initialization modules have been executed. Device driver code should never enable interrupts.

   In addition, if you wish to execute some code with interrupts disabled, this should not be done with just an SEI instruction. Instead you should use the sequence of PHP, SEI ..code.. PLP. This ensures that the system state is correctly restored when you exit the critical section (after the PLP).

3. Any driver that uses interrupts must initialize itself before the system starts up in order to link its interrupt service code into the chain of service routines. The initialization code should do the following (before exiting) in order to initialize the links:

```
        LDA     0FFFE           ; move IRQ vector into next
        STA     STOREIT         ; driver pointer
        LDA     0FFFF
        STA     STOREIT+1

        LDA     I_ADDRESS       ; move interrupt service routine
        STA     0FFFE           ;  address into the IRQ vector
        LDA     I_ADDRESS+1
        STA     0FFFF

I_ADDRESS  .WORD  I_HANDLER
STOREIT    .WORD  0             ;  next driver pointer
```

   where: I_HANDLER is the entry point of the driver's interrupt service routine;

   STOREIT will contain the address of the next interrupt service routine to be called if the current one finds that its device did not generate the interrupt.

   Note: This code must be executed only once and must not be in a transient initialization module. The driver itself may also contain "regular" initialization code to reset the device or its buffer, and so forth.

4. At the start of its interrupt service routine(s), a device driver must first determine whether the driver's device hardware generated the interrupt.

   The details are device-dependent, but in general involve checking a register on the device's controller card (for example, an interrupt flag register on a 6522).

   If the interrupt was generated by the driver's device, the driver should process the interrupt and then return to the IM by an RTI instruction.

   If the interrupt was not generated by the driver's device, the driver should do an indirect jump to the next device driver (the address of the next driver is saved as STOREIT in the sample initialization code under item 3, above). If this device driver is the last in the chain, the jump will be to

the Pascal system reset code.

Note: The jump to Pascal system reset code is accomplished automatically, since the system initializes the IRQ vector to point to the reset code. If the initialization for all interrupt-based device drivers is correct (as shown in item 3), this pointer will be moved to the end of the interrupt service routine chain.

Important: If your device card has no way of signalling that it generated an interrupt, then its service routine must be the last service routine in the chain. It will have to assume that if it is called, it will handle an interrupt. This is not a good approach, since the routine won't be able to detect BRK or hardware failure interrupts.

To ensure that a driver is the last one in the interrupt drivers chain, assign it a unit number lower than all other interrupt driver unit numbers.

5. An interrupt service routine must be an integral part of the driver's code. This ensures that it will be loaded by SYSTEM.ATTACH. If you don't do this, your code is in danger of being released by the system -- a subsequent interrupt may cause unpredictable effects.

6. If you use the 64K Pascal system, the IM driver should be included on the boot diskette inside the ATTACH.DRIVERS file. You may use the standard library program (LIBRARY.CODE) to look into the file and/or transfer the code segment to another file. The code size of IM will be shown as approximately 280 bytes, but much of this size corresponds to relocation code that will not be resident at run time. At run time, the IM occupies approximately 200 bytes.

7. The program ATTACHUD.CODE is used to save information about a driver in ATTACH.DATA. For each driver, ATTACHUD will ask you if the driver uses interrupts. If you answer yes, ATTACHUD ensures that a data record for the IM driver is automatically included in the ATTACH.DATA file. Note that this data record is automatically included in ATTACH.DATA as long as at least one of your drivers uses interrupts.

   On the 64K Pascal system, the IM driver is automatically attached if the IM data record is present in ATTACH.DATA. If the record is not present, the IM driver is not attached. On the 128K Pascal system, the IM data record is ignored if it is present.

8. It is not, repeat not, necessary to save registers in an interrupt service routine. The IM saves them before jumping to the drivers chain, and restores them before resuming normal execution of the interrupted code. You should use the standard 'RTI' instruction at the end of the interrupt service routine: not an 'RTS'. The 'RTI' instruction transfers control back to the IM. (RTI is used because the IM saves additional status information in the processor status byte and then pushes this byte onto the stack.)

9. A change has been made to the 1.2 Pascal system to eliminate a problem associated with abnormal termination of the system with certain interrupting devices. This can occur when a program gets a system error or when a user

interrupts the program from the keyboard (CTRL-@).

When the system terminates abnormally, it executes a UNITCLEAR on all devices (1-20 and 128-143). This is done even when the driver's data record (in ATTACH.DATA) specifies that no initialization is to be done at boot time.

This presents a problem when the UNITCLEAR portion of a driver contains code to initialize the service routine chain (as described above in item 3). Drivers under version 1.2 must have some code to distinguish between the first initialization (which sets up the driver chain) and any subsequent initialization (i.e., a call to UNITCLEAR).

In the driver, these two kinds of initialization may be distinguished by a simple check of a byte of memory to see which type of initialization code needs to be run (if any). This is the scheme used in the example below.

The 1.2 system reinitializes all devices because some drivers may have pointers into the stack/heap space. If this space were released without reinitializing the device drivers, the pointers would now point to invalid code or data. The problem can't be solved by simply disabling further interrupts, since some external devices (e.g., a remote network printer server) may have to be notified of the reset; if interrupts were disabled, information coming back from the remote device could not be handled correctly.

10. Location $7F8 must contain the value $Cn, where n is the slot number of the card, if your card uses the $C800 expansion space. The reason for this is that when you are executing in your $C800 space and an interrupt occurs, the interrupt routine may decide to use its own $C800 space. When the interrupt has completed, the system must know if it needs to reselect the $C800 space for your card. The IM will take the contents of location $7F8 (which can be initialized any time before your driver enters the $C800 space), and use it to reselect your card. If you do not do this, it is very possible that your routines may not work correctly since your $C800 space will not be reselected. The only other way to avoid this is to disable all interrupts while you are in your $C800 space.

    Note: Interrupt service routines must never alter the contents of location $7F8, as this may cause the wrong $C800 space to be reselected after the interrupt has been serviced.

11. Interrupts are disabled when an interrupt occurs, and are re-enabled by the IM after the interrupt has been serviced. Only one interrupt may be handled at a time.

    Devices or drivers must never re-enable interrupts if they have been disabled by the IM.

12. There are additional restrictions on interrupts for applications that execute under the 64K Pascal system and that also use the auxiliary 64K memory on a IIe. Since the IM and all interrupt service routines are resident in the main RAM, if an interrupt occurs while the application is using the auxiliary RAM, the interrupt will not be serviced properly, and

may cause the system to crash.

For this reason, an application should disable interrupts while the auxiliary 64K is in use, or should be able to handle the interrupt management itself.

13. On the Apple IIe, the IM will save the state of the 80STORE and PAGE2 soft switches, and will deselect PAGE2 if 80STORE is selected. The original state of the PAGE2 switch is restored after the interrupt is serviced.

14. In the 128K Pascal system, the IM will save the state of the RAMRD and RAMWRT soft switches and will then select read main RAM and write main RAM. The original state is restored after the interrupt is serviced.

15. If an interrupt service routine uses any zero-page user temporaries ($0-$35), then it must save their contents, and restore them after the interrupt has been serviced.

16. If an application switches in the Monitor ROM, it must disable interrupts prior to doing so.

The following is a brief scenario for installing an interrupt-based device driver in your Pascal system.

1. Write the device driver and assemble it, according to the requirements given above.

2. Execute ATTACHUD to create an attach data file for your driver.

   If you have already defined other device drivers, call the new attach data file INTERRUPT.DATA, for example. Then execute ADMERG to append your driver data file INTERRUPT.DATA to the existing ATTACH.DATA file.

   If you do not have any other device drivers in your system, call the new attach data file ATTACH.DATA.

   Be sure to tell ATTACHUD that your driver uses interrupts.

   Next, execute LIBRARY.CODE to place your driver code in the ATTACH.DRIVERS file. (On the 64K Pascal system, you must include IM.CODE in ATTACH.DRIVERS, if it is not already present.)

   Note: If you change your device driver and reassemble it, you don't always need to run ATTACHUD a second time. Changes to driver code don't affect the data record in ATTACH.DATA unless you have changed something which affects the answer to one of the questions which ATTACHUD asks you. You will still have to use LIBRARY to place the new code in the ATTACH.DRIVERS file.

3. Along with the standard files for a bootable Pascal disk, the following files <u>must</u> be on your new boot diskette: SYSTEM.ATTACH, ATTACH.DRIVERS, and ATTACH.DATA. When you boot from the new diskette, the driver will be loaded, and you can test it and use it.

Remember that you can use SHOWAD to view the contents of ATTACH.DATA, and LIBRARY to view the contents of ATTACH.DRIVERS.

Changes from Version 1.1

Under version 1.1, interrupts were theoretically allowed, since the system disabled interrupts during time-critical operations such as disk accesses. Unfortunately, when a disk access was completed interrupts were never re-enabled, so that interrupts functioned correctly only until a program's first disk access!

Version 1.1 could support only one interrupting device per system.

This is the scheme described in Pascal Technical Note #11.

Version 1.2 of the Apple II Pascal system can support multiple interrupting devices. For 128K systems, this capability is built in. For 64K systems, the interrupt manager (IM) is shipped in the file IM.CODE.

Example of an Interrupt-Based Device Driver

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;                 Apple II Pascal 1.2 Sample Interrupt Driver
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;    Copyright 1983 - Apple Computer Inc.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;    This sample driver is a user-defined device driver.  It shows both
;    the overall skeleton of a user driver and more importantly it shows
;    how to write an interrupt-based device driver that uses the
;    interrupt manager (IM).
;
;
;       Macro Subroutines

; Save/restore word off the stack (used to save return addresses)
        .MACRO  POP
        PLA
        STA     %1
        PLA
        STA     %1+1
        .ENDM
;
        .MACRO  PUSH
        LDA     %1+1
        PHA
        LDA     %1
        PHA
        .ENDM
; The ol' switch macro (see SOS Reference Manual for description)
        .MACRO  SWITCH
        .IF     "%1" <> ""
        LDA     %1
        .ENDC
        .IF     "%2" <> ""
        CMP     #%2+1
        BCS     $010
        .ENDC
        ASL     A
        TAY
        LDA     %3+1,Y
        PHA
        LDA     %3,Y
        PHA
        .IF     "%4" <> "*"
        RTS
        .ENDC
$010    .ENDM
; Move first word into the second
```

```
        .MACRO  MOVE
        LDA     %1
        STA     %2
        LDA     %1+1
        STA     %2+1
        .ENDM

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;    Equates
;
;       Zero page (0-$35 is available) is used for return addresses,
;       and global temps.
;

; Zero page temporary locations

CSLIST  .EQU    0               ; Buffer address
CTRLWORD .EQU   2               ; storage for ctrl word

IRQ     .EQU    0FFFE           ; IRQ vector location
FLAG6522 .EQU   0C2ED           ; Interrupt Flag Register
                                ;   for a hypothetical card in Slot 2

;
; Error code equates
;
; Upon completion of the driver, the X register will hold an appropriate
; error code that will be converted into the Pascal reserved variable
; IORESULT.  The Pascal program should check IORESULT after all UNITSTATUS
; calls made to the driver.  Error code numbers 128-255 are to be used by
; your driver.
;

XNOERRS .EQU    0               ; no errors encountered
XBADCMD .EQU    3               ; bad command to driver
ERRCODE .EQU    128.            ; user defined error message

        .PROC   SAMPLE

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;    This is the main entry point for the ATTACH driver.  This driver
;    is defined as a 'user device' and therefore will only be used from
;    Pascal using direct I/O (i.e, UNITSTATUS, UNITREAD/WRITE).
;
;    Upon entrance, the X register will contain the type of call requested
;    (UNITREAD,WRITE,CLEAR, etc).  See ATTACH documentation for more
;    details on the stack setup.
;

START
        POP     RETURN          ; save return address
        TXA                     ; get type of call
```

```
        SWITCH  ,4,IDTABLE

BADREQ  LDX     #XBADCMD           ; if you got here, the call is in error!
        BNE     GOBACK             ; always taken
GOBACKOK LDX    #XNOERRS           ; go here if you want to return with no errs
GOBACK  PUSH    RETURN             ; or return with X register holding error code
        RTS                        ; main exit point

IDTABLE .EQU    *
        .WORD   READ-1
        .WORD   WRITE-1
        .WORD   INIT-1
        .WORD   BADREQ-1
        .WORD   U_STATUS-1


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;    INIT does two things: 1) moves the IRQ vectors to the appropriate
;    locations the first time called and 2) every additional call will be
;    meant to issue an appropriate initialization request to the driver
;    (if required).

INIT
        PHP
        SEI                        ; Disable Interrupts
        LDA     TYPE
        BEQ     $001               ; if zero then do init stuff
        .
        .
        Any UNITCLEAR call after the initial one by the system will jump
        to this area
        .
        .
        JMP     $090               ; always taken
$001    INC     TYPE               ; bump type field so next time we dont do it

;
;  This next section moves the IRQ vector into a temporary
;  location.  The MOVE macro is a 16-bit move instruction -- see above
;  macros for an explanation.
;

        MOVE    IRQ,JUMPTO
        MOVE    INTADR,IRQ     ; patch IRQ location and jump vector
        .
        .
        more code for initial initialization call
        .
        .
$090    PLP
        JMP     GOBACKOK
INTADR  .WORD   INTHNDLR           ; Interrupt handler address
JUMPTO  .WORD   0                  ; save area for next interrupt svc routine
```

```
TYPE     .BYTE   0                    ; if 0 then init call else cleanup call
RETURN   .WORD   0                    ; return address for Pascal


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;     READ is called when the program generates a UNITREAD
;
READ
         .
         .
         code for the UNITREAD call
         .
         .
         LDX      #ERRCODE            ; error completion code
         JMP      GOBACK

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;     WRITE is called when the program generates a UNITWRITE
;
WRITE
         .
         .
         code for the UNITWRITE call
         .
         .
         LDX      #ERRCODE            ; error completion code
         JMP      GOBACK


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;     U_STATUS is called when the program executes a UNITSTATUS call to
;     this particular device.
;
;     The order of the stack (4 bytes) is:
;
;     TOS => POINTER TO STATUS RECORD
;
;            CONTROL WORD      bits  15..13    12..2       1          0
;                                      user    reserved    status/    direction
;                                    defined   for future  control
;
;                                direction - 0 = status of output channel
;                                            1 = status of input channel
;                              status/ctrl - 0 = status call
;                                            1 = control call
;
;                              Bits 13-15 should have the number of the
;                              control/status request.
;
;
;
```

```
;     From Pascal, the call should be:
;
;                UNITSTATUS(130,BUFFER,OPTION)
;
;     where:
;           130 = Device driver number (currently 130 is used by this driver)
;           BUFFER = PACKED ARRAY [0..??] OF 0..255;
;                 This array is as big as needed by the code called.
;           OPTION = PACKED RECORD
;                       DIRECTION : 0..1;
;                       STAT_CTRL : 0..1;
;                       RESERVED : 0..2047;
;                       CODE : 0..7;
;                     END;
U_STATUS
        POP     CSLIST          ; see above
        POP     CTRLWORD        ; ditto
        LDA     #02             ; mask for status/control bit
        BIT     CTRLWORD        ; if bit 2 is set, zero flag will be cleared
        BNE     CONTROL         ; go do a control call
        JMP     STATUS          ; status request

;
;  This is the unitstatus call -- control request section
;
CONTROL
        LDA     CTRLWORD+1      ; get control word
        AND     #0E0            ; is it #0   :  0000 0000
        BEQ     CODE_ZERO       ; yes
        CMP     #20             ; is it #1   :  0010 0000
        BEQ     CODE_ONE        ; yes
        CMP     #40             ; is it #2   :  0100 0000
        BEQ     CODE_TWO
        CMP     #60             ; is it #3   :  0110 0000
        BEQ     CODE_THREE
        LDX     #XBADCODE       ; completion code error
        JMP     GOBACK
CODE_ZERO
        .
        .
        code for control code zero
        .
        .
        LDX     #ERRCODE        ; completion code error
        JMP     GOBACK          ; and report it
CODE_ONE
        .
        .
        code for control code one
        .
        .
        LDX     #ERRCODE        ; completion code error
        JMP     GOBACK          ; and report it
```

```
; repeat for codes 2 and 3
;
CODE_TWO
        .
        .
        LDX     #ERRCODE        ; completion code error
        JMP     GOBACK          ; and report it
CODE_THREE
        .
        .
        LDX     #ERRCODE        ; completion code error
        JMP     GOBACK          ; and report it

;
;  This is the unitstatus call -- status request section
;
STATUS
        LDA     CTRLWORD+1      ; get control word
        AND     #0E0            ; is it #0   :  0000 0000
        BEQ     SCODE_ONE       ; yes
        CMP     #20             ; is it #1   :  0010 0000
        BEQ     SCODE_TWO       ; yes
        LDX     #XBADCODE
        JMP     GOBACK
SCODE_ONE
        .
        .
        status code one
        .
        .
        LDX     #ERRCODE        ; completion code error
        JMP     GOBACK          ; and report it
SCODE_TWO
        .
        .
        status code two
        .
        .
        LDX     #ERRCODE        ; completion code error
        JMP     GOBACK          ; and report it


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;    This is the interrupt handler.  Remember that we don't have to save
;    the context of the system.  The code used to check for an interrupt
;    will have to be changed depending on your hardware.
;

INTHNDLR

;
; first we check to see if we generated the interrupt
;
```

```
        LDA     FLAG6522        ; check for 6522 interrupt flag
        AND     #80             ; mask out bit
        BPL     GOAHEAD         ; if bit was set, we generated it
        JMP     NEXT            ;  otherwise goto the next interrupt handler
;
; Since we generated the interrupt, we service it and then return to
; the interrupt manager with an 'RTI'
;
GOAHEAD
        .
        .
        interrupt handler code for our card
        .
        .
        RTI                     ; Go back to the Interrupt Manager

;
; If we got to NEXT, we must have decided that the interrupt was not
; generated by us.
;
NEXT
        JMP     @JUMPTO
        .END
```

Apple Pascal Object Module Format

Pacal Technical Note # 16

15 October 1981

INTRODUCTION

This document describes the object module format of codefiles currently produced in the Apple][and /// Pascal systems. The only difference between the format of the][and /// codefiles is the information contained in block 0 as noted below. The P-code for both systems is identical.

A CODEFILE ON DISKETTE

Codefiles may be unlinked files created by the compiler or assembler, library files with units which may be used by programs in other codefiles, or linked files composed of segments ready for execution. All codefiles (linked and unlinked) consist of a segment dictionary in block 0 of the file followed by a sequence of one or more code segments up to a total of sixteen segments.

Segments may be linked or unlinked code segments, or data segments for an intrinsic unit. Code segments may have interface text, code blocks, and linker information in that order in blocks on the diskette, though some of these parts may be present only for particular types of code segments. For example, interface text is only present in code segments of units. Data segments only have an entry in the segment dictionary: they do not occupy any blocks on the diskette since they have no code, interface, or linker information associated with them. The only difference between the format of][and /// codefiles is the information in block 0.

Each code segment begins on a boundary between diskette blocks (the 512-byte disk allocation quantum used by the Apple Pascal operating system). Each segment may occupy many blocks (up to a maximum of 32K bytes). A typical codefile is shown in Figure 0.

The following sections describe the parts of a codefile in greater detail. First the segment dictionary is described. Then the parts of a code segment are presented in the order in which they would occur in a file: the interface part, the code part, and finally linker information. The code part description is broken up into sections describing the similarities and differences between code parts for P-code and assembly language modules.

SEGMENT DICTIONARY: BLOCK ZERO OF A CODEFILE

The segment dictionary in block 0 of a codefile contains information regarding name, kind, relative address and length of each code segment. It is

represented by a record presented below in a pseudo-code presentation and illustrated in Figure 1.

The segment dictionary contains an entry for each code or data segment in the file. (The userprogram main segment is assigned segment number 1; the system main segment is assigned segment number 0. Both are placed in slot 0 of their respective codefiles by the compiler. This differs from the statement in the "Pascal Operating System Manual", page 250, which incorrectly states that "the main program is assigned segment #0".)

Each segment dictionary entry includes the segment's size (in bytes). This size is set to zero if there is no segment in the slot. The entry also contains the segment's disk location, which is set to 0 for a data segment of an intrinsic unit. Blocks in a codefile are numbered sequentially from 0, 0 being the segment dictionary. The disk location for non-data segments is given as the block number of the first block containing code for the segment.

```
RECORD {This record is composed of parallel 16 element arrays, one element for
each possible segment slot in the segment dictionary of a codefile.}

      DISKINFO:  ARRAY[0..15] OF
         RECORD
            CODELENG, CODEADDR:  INTEGER
         END;

         {The first array is composed of two-word records made up of two
         integers representing the length of the code part of a segment in
         bytes and the block number of the start of the code part of the
         segment. On the diskette, the CODEADDR field appears before the
         CODELENG in each pair.}

      SEGNAME:  ARRAY[0..15] OF PACKED ARRAY[0..7] OF CHAR;
         {This is a sixteen element array of eight character arrays which
         describe the segments by name. These eight characters are those
         which identify the main program and its segment procedures at compile
         time. Unused segment slots have name fields filled with eight
         ASCII space characters; if the name is less than eight characters it
         is padded on the right by spaces; if the name is longer than eight
         characters, it is truncated. Note that a blank field is allowed for
         an existing code segment. CODELENG=0 should be used to determine an
         empty slot.}

      SEGKIND:  ARRAY [0..15] OF

                {The next array describes the kind of segment in the
                particular entry location of the dictionary. The possible
                values are described below.}

                (LINKED, {=0. This represents a fully executable segment.
                   Either all external references (regular UNITS or EXTERNALS
                   or .REFs) have been resolved, or none were present.}

                HOSTSEG, {=1. This represents the outer block of a Pascal
```

program if the program has unresolved external references.}

SEGPROC, {=2. A Pascal segment procedure. This type is not currently used.}

UNITSEG, {=3. A compiled regular (as opposed to intrinsic) unit.}

SEPRTSEG, {=4. A separately compiled (set of) procedures or functions. Assembly language codefiles are always of this type.}

UNLINKED-INTRINS, {=5. An intrinsic unit containing unresolved calls to assembly language procedures or functions.}

LINKED-INTRINS, {=6. An intrinsic unit in its final, ready-to-run state.}

DATASEG); {=7. A specification of the data segment associated with an intrinsic unit telling how many bytes to allocate and which segment to use.}

TEXTADDR: ARRAY[0..15] OF INTEGER; {This array of integers gives the block number of the start of the interface part of each regular or intrinsic unit. The last block of the interface section is inferred from CODEADDR-1. Array elements corresponding to non-unit segments have the value zero. Segments are stored with their interface blocks (if any) first, followed by their code part blocks and finally their linker information blocks (containing symbol table elements for items used but not defined in the segment or for items defined in the segment and externally accessible.) Linker informations records are described in detail below.}

SEGINFO: PACKED ARRAY[0..15] OF
{This array has one word per segment entry.}

PACKED RECORD

SEGNUM: 0..255

{Bits 0 through 7 (the low order bits) of each word specify the segment number for that code. This is the position the code segment will occupy in the system's SEGTABLE at execution time. This table is 32 elements long in the Apple][and 64 elements long in the Apple ///. Thus valid numbers for the first field are 0..31 on the][and 0..63 on the ///.

The run time segment table contains an entry for each segment that is used in executing the program. There are entries for 6 segments that the system uses when executing a user program on the][; on the ///, 8

segments are used by the Pascal operating system. There is an entry for each segment in the segment dictionary of the program's code file. Finally, there is an entry for each code and data segment of each intrinsic unit.

At run time no two segments in the segment table can have the same number since the numbers are used to index the table. A number is assigned to a program segment when an entry is created for it in the code file's segment dictionary. The main program has segment number 1. The segments used by the system are 0 and 2..6 on the][and /// and, additionally, 62 and 63 on the ///. Also, segments 59 through 61 are reserved for use by the system. The segment number of an intrinsic unit is determined by the unit's heading when the unit is compiled. (These numbers can be found by examining the segment dictionary of the SYSTEM.LIBRARY file with the LIBMAP or LIBRARY utility programs.) The segment numbers of regular unit segments and of segment pocedures and functions are automatically assigned by the system; they begin at 7 and ascend. Note that after a regular unit is linked into a program, it may not have the same segment number shown for it in the library's segment dictionary when the library is examined with LIBMAP.

Since the Pascal system itself uses 6 slots on the][and 8 slots on the /// in the runtime SEGTABLE, this means that a program can have 26 user defined or intrinsic segments (6+26=32) on the][. A codefile is, as we have seen, limited to 16 segments by the number of spaces in the segment dictionary; this is true for both user codefiles and the SYSTEM.LIBRARY codefile. Thus on the][, 16 of the 26 can be in the user's codefile while the excess over 16 could be intrinsics. On the ///, there are 64 possible segments. However, the maximum which can be used is 56: 8 for the system, a maximum of 16 for the user program, up to 16 user program library code or data segments, and up to 16 system library code or data segments. 8+16+16+16=56.

Thus, segment numbers of the program itself, the segments used by the Pascal system, and of any intrinsic units used by the program are fixed before the program is compiled; the segments of regular units and of segment procedures and functions are not fixed and are assigned as the program is compiled and linked in ascending sequence beginning with 7. Normally, users need to specify segment numbers only when writing an intrinsic unit. The choice must avoid the fixed numbers 0..6 (and 59 through 63 on the ///) or any other intrinsic unit which may be used in the same program as the unit being written. In particular, the "magic units" PASCALIO and LONGINTIO occupy segments numbers 30 and 31.

Intrinsic unit segment numbers must also avoid conflict with numbers which may be assigned automatically to regular units and segment procedures. However, when unavoidable conflicts arise, the "Next Segment" compiler option described in the "Apple Pascal Language Manual Addendum" may be used to set the segment number to another value.}

```
MTYPE:  0..15;
```

{The second byte in the SEGINFO word has in bits 8 through 11 the "machine type" which tells what kind of code is present in the code segment. The machine types are:

0 Unidentified code. Perhaps from a previous compiler.

1 P-code, most significant byte first.

2 P-code, least significant byte first. A stream of packed ASCII characters fills the low byte of a word first, then the high byte. This is the kind of P-code used by Apple.

3 through 9 Assembled machine code, produced from assembly-language text. Machine type 7 identifies machine code for Apple's 6502.}

```
UNUSED:  0..1;

VERSION:  0..7
```

{The version number of the system. On the Apple][the current version number is 2; on the Apple /// the current version number is 3.}

```
END;

INTRINS-SEGS:  {ON THE ][} SET OF 0..31;
               {ON THE ///} SET OF 0..63;
```

{These words (two on the][, four on the ///) tell the system which intrinsic units are needed in order to execute the codefile. Each intrinsic unit in SYSTEM.LIBRARY (and in the program library on the Apple ///) is identified by a segment number (or two segment numbers if the intrinsic unit has a data segment.) Each of the bits in these words correspond to one of the thirty-two or sixty-four possible intrinsic segment numbers. If the n-th bit is set to 1, this indicates the program needs the intrinsic unit whose segment number in SYSTEM.LIBRARY (or in the program library on the Apple ///) is n.}

```
INT-NAM-CHECKSUM:  {Only on the ///}
  PACKED ARRAY[0..63] OF 0..255;
```

{These fields contain eight-bit checksums of the names of intrinsic units needed to run the codefile. Each entry corresponds to one of the sixty-four possible intrinsic segment numbers on the ///.

The checksum is calculated by shifting the characters of the UNIT name to upper case and adding up the resulting ASCII values of the characters of the UNIT name MOD 256. The name is padded with spaces on the right if it is shorter than eight characters; it is truncated to eight characters if it is longer than eight characters. Padding spaces are included in the checksums. Elements corresponding to unused segment numbers are set to zero.

These words are not used on the][; they must be zeroed.}

{UNUSED JUNK (FILLED WITH ZEROES) FOLLOWED BY}

COMMENT: PACKED ARRAY [0..79] OF CHAR
{the text following a Comment compiler option, starting in byte 432 of the header}

END;

THE INTERFACE PART

Code segments for units may have an INTERFACE part before their associated code blocks. This contains the ASCII text of the INTERFACE declaration in the source code of the UNIT. The construction of an INTERFACE part of a code segment from its source code is shown in Figure 2.

The Pascal compiler emits two block pages (1024 bytes) of text which are identical to the source text blocks except for the first and last pages. The information in the first page is moved up so the first character in the page is the character following "INTERFACE" in the original source. This may leave a considerable amount of unused characters in the first page. Useful information is terminated by a CR and followed by at least one ASCII NULL character (byte value 0). The last page is truncated after the token "IMPLEMENTATION"; it is possible that only one block of this page may be put out if "IMPLEMENTATION" occurs in the first block of the page.

There is some special encoding after the token "IMPLEMENTATION." The immediately following ten characters are composed of ASCII spaces except for an "E" in the ninth position. This is required by the Pascal compiler and librarian program to terminate the interface section. A "P" may occur instead of a space in the second of the ten character positions to signify to the Pascal compiler that the unit requires the PASCALIO unit. The fourth position may be occupied by an "L" if the unit requires the LONGINTIO unit. Failure to include these can cause the system units not to be loaded when needed causing a system crash. Note that these items—IMPLEMENTATION, E, P, and L—are all taken to be tokens by the compiler; thus, the order is significant, the spacing and case is not.

The INTERFACE text is not stripped of excess non-printing characters or

comments and is accessed by the compiler when the UNIT is USEd by another program. Leaving the comments in the INTERFACE part could lead to more complete internal program documentation but may increase size of codefile. This text is not necessary for execution.

The address of the INTERFACE part is given as a block number relative to the start of the code file in the TEXTADDR field described below. This field is zero for segments which are not UNIT code segments or do not have an interface text.

CODE PARTS

As has been mentioned, all non-data segments appear on the diskette as the text of an interface part (if the segment is a regular or intrinsic unit) followed by code blocks followed by linker information (if the segment has undefined elements or has elements which may be linked to other modules.) Data segments for intrinsic units do not occupy any disk blocks.

All code parts have the same general format illustrated in Figure 3. Each code part contains code for that segment's outer block, as well as the code for each of the (non-segment) procedures within that segment. Following code for various procedures associated with the segment is the procedure dictionary at the high address indicated by the CODELENG field of the associated entry in the segment dictionary in block 0 of the codefile. This procedure dictionary grows down; the code starts at the first byte of the block specified in the CODEADDR field of the segment dictionary and grows up.

Each procedure in a code part is assigned a procedure number starting at 1 for the outer block (the main program or segment procedure) and ranging as high as 160. All references to a procedure are made via its number. Translation from a procedure's number to the location of that procedure's code in the code segment is accomplished via the procedure dictionary at the end of the segment. This dictionary is an array indexed by the procedure number. Each array entry is a self-relative pointer to the code for the corresponding procedure. [Since the procedure dictionary starts at the high end of a code segment and works down toward lower addresses, the term "self relative pointer" could be ambiguous: it could be positive or negative depending on interpretation! In all that follows, a self relative pointer is taken to be the absolute distance (in bytes, a positive integer number) between the low order byte of the pointer and the low order byte of the word to which it points.] In other words, you subtract the pointer from its location to find the word pointed to.

Since zero is not a valid procedure number, the zero-th entry of the dictionary is used to store the segment number of the code segment in the low order (even) byte and the number of procedures in that code segment in the high order (odd) byte. The segment number corresponds to the value in the SEGNUM field of the segment dictionary entry.

There are currently two forms of code contained in procedures: P-code and assembly language (or TLA for "The Last Assembler", the familiar name of the assembler currently in use in the Apple Pascal system). Each procedure's code

section consists of two parts: the procedure code itself (in the lower portion of the section growing up toward higher addresses) and a table of attributes of the procedure pointed to by the entry in the procedure dictionary. This table of attributes is loosely known as the Jump Table (JTAB), a term more properly used to refer only to a portion of the table in P-code procedures. The format of the attribute table for a TLA procedure is very different from that for a P-code procedure. These formats are described in the following two sections.

While the compiler and the assembler produce "pure" P-code or TLA code sections, it is possible to produce segments with mixed procedure code type using the Linker. In this case the MTYPE field in the segment dictionary is set to the value for assembly language code, because the code for that segment is now machine specific. The interpreter is able to determine the type of code of a particular procedure via information contained in the procedure's attribute table as is discussed below.

P-CODE PROCEDURE ATTRIBUTE TABLES

The format of a P-code attribute table is illustrated in Figure 4. The contents of the P-code attribute table are:

PROCEDURE NUMBER: Low order, even byte of the word pointed to by the segment dictionary entry. Refers to the number given this procedure in the procedure dictionary of the parent code segment.

LEX LEVEL: High order, odd byte of the same word. Specifies the absolute lexical nesting level for the procedure.

ENTER IC: A self-relative pointer (again, a positive number, pointing back) to the first p-code instruction to be executed for the procedure.

EXIT IC: A self-relative pointer to the beginning of the block of p-code instructions which must be executed to terminate the procedure properly.

PARAMETER SIZE: The number of words of parameters passed to a procedure from its caller.

DATA SIZE: The size of the procedure's activation record in bytes, excluding the Markstack and PARAMETER SIZE. The activation record includes variables and temporary space used by the procedure.

Between these attributes and the procedure code there may be an optional section called the "jump table". Its entries are addresses within the procedure code (as self-relative pointers). During execution, the JTAB system register points to the attributes and jump table section of the currently executing procedure (points to the byte containing the procedure number).

In executing jumps in P-code, a jump opcode has a single byte operand. This is a signed offset: the high order byte is taken to be the sign extension of bit 7. If the offset is non-negative (a short forward jump), it is added to the

interpreter program counter, IPC. (A value of zero for the jump offset makes any jump a two-byte NOP.) If it is negative (a backward or long forward jump), then the operand DIV 2 is used as a word offset into JTAB to find a self-relative pointer, and the instruction program counter is then set to the byte address (JTAB^[operand DIV 2]) - contents of (JTAB[operand DIV 2]).

ASSEMBLY LANGUAGE (TLA) PROCEDURE ATTRIBUTE TABLES

The format of a JTAB for an assembly procedure is very different from that for a P-code procedure. It is illustrated in Figure 5.

The highest word in the JTAB in an assembly procedure always has a zero in its PROCEDURE NUMBER field. In what was the LEX LEVEL field of a P-code procedure JTAB (the high order byte) is either a zero (indicating that BASE RELATIVE relocation is to be relative to the host program activation record) or a non-zero number (indicating the number of the segment relative to which BASE RELATIVE relocation should take place.) In the case of INTRINSIC units without explicitly specified data segments, the number placed in this field is 1. When the interpreter encounters a zero in the procedure number field as it loads the segment, it realizes it must fix up references in the TLA code according to information contained in the rest of the attribute table.

The second highest word of the attribute table is, as before, the ENTER IC: the self-relative pointer to the first instruction to be executed for this procedure. Following this are four relocation tables used by the interpreter at fix-up time.

Working down from the high address start of the JTAB we encounter in order the BASE RELATIVE, SEGMENT RELATIVE, SELF RELATIVE, and INTERPRETER RELATIVE relocation tables. The format of all of these tables is the same: the highest address word of each table specifies the number of entries (possibly zero) which follow (at lower diskette addresses) in the table. Then follows that many single-word entries, which are self relative pointers to locations in the code which must be "fixed up" by the addition of the appropriate relative relocation constant known to the interpreter at load time.

In the case of the BASE RELATIVE relocation table, the value contained in the interpreter's BASE pseudo-register is added if the LEX LEVEL (high order) byte of the procedure's attribute table is zero; if the byte is non-zero, the relocations will be relative to the segment whose segment number is contained in the field. The BASE register is a pointer to the activation record of the most recently invoked base procedure (lexical level 0). Global (lex level 0) variables are accessed by indexing off BASE. The TLA .PUBLIC and .PRIVATE constructs define those entities whose use results in entries into this table.

In the case of the SEGMENT RELATIVE table, the value of the address of the lowest byte in the segment is added. The TLA .REF and .DEF are the relevant constructs.

SELF RELATIVE items have the procedure address (i.e., the address of the lowest byte in the procedure) added.

INTERPRETER RELATIVE items access the Pascal interpreter procedures or variables. They should never be used.

LINKER INFORMATION

Following the code part of a segment there may be Linker information. The starting location of linker information is not included in the segment dictionary as was the case with the starting location of the interface and code parts. It must be inferred. Linker information items may be present for unlinked code segments (i.e., a segment containing unresolved external references) as well as for segments containing items which may be referenced from other segments (e.g., .PROC and .FUNC elements in assembly language programs which may be accessed as EXTERNAL PROCEDURES and FUNCTIONS.) The Linker information begins at the first block boundary following the last block of code for a segment. It is described in detail below. The linker information is a series of records, one for each unit, routine or variable which is referenced but not defined in the source as well as records for items defined to be accessible from other modules. There are records for the following types of items:

```
     litypes =

{0}    ( EOFMARK,  {end-of link-information marker}

         {External reference types: designates fields to be updated by the
         linker}

{1}    UNITREF, {references to invisibly used units— i.e., a reference in
            one unit to another unit. Used in the case of one non-intrinsic
            unit using another non-intrinsic unit.}

{2}    GLOBREF, {references to external global addresses:  results from a
            .REF construct in an assembly language program.}

{3}    PUBLREF, {references to a variable in the global data segment of the host
            program:  results from a .PUBLIC in assembly language code or use
            of variables declared in the INTERFACE part of regular units.  (They
            are stored in another segment in intrinsic units— the data segment
            of the unit.)

{4}    PRIVREF, {references to variables of an assembly language routine or
            regular unit to be stored in the host program's global data segment
            and yet be inaccessible to the host program.  Space is allocated by
            the Linker.  Generated by .PRIVATE in assembly language.  Also,
            generated by use of global variables declared in the IMPLEMENTATION
            part of regular units.  (In intrinsic units, these are also stored
            in the data segment of the unit.)

{5}    CONSTREF, {references to a globally declared constant in the host
            program.  Generated by .CONST in assembly language.
```

{defining types: gives linker values to fix references}

{6} GLOBDEF, {Global address location. Generated by .DEF (and .PROC and .FUNC) in assembly language}

{7} PUBLDEF, {A variable location in the host program. Generated by VAR declaration in Pascal}

{8} CONSTDEF, {A host program constant definition. Generated by CONSTANT in Pascal.}

{procedure/function information:
Assembler to Pascal and Pascal to Pascal interfaces}

{9} EXTPROC, { References to procedure declared to be external in Pascal: generated by PROCEDURE...EXTERNAL}

{10} EXTFUNC, { References to function declared to be external in Pascal: generated by FUNCTION...EXTERNAL}

{11} SEPPROC, {Separate Procedure definition to be linked into Pascal: generated by .PROC in assembly language.}

{12} SEPFUNC, {Separate Function definition to be linked into Pascal: generated by .FUNC in assembly language.}

{13} SEPPREF, {Not currently used. Was once used for references to procedures in a "separate unit", a concept which has been removed from the current implementation.}

{14} SEPFREF, {Not currently used. Was once used for references to functions in a "separate unit", a concept which has been removed from the current implementation.}

The exact format of data in the linker information block is dependent on the type of entity. They are described by the following record.

```
OPFORMAT = (WORD,BYTE,BIG);
LIENTRY = RECORD

   NAME:  PACKED ARRAY[0..7] OF CHAR; { The name of the symbol.  }

   CASE LITYPE:  LITYPES OF

      GLOBREF,
      PUBLREF,
      PRIVREF,
      CONSTREF,
      UNITREF,

      SEPPREF, {Not currently used}
      SEPFREF:  {Not currently used}

         (FORMAT:  OPFORMAT; {The format of the operand represented by
```

the named (and currently undefined) symbol. May be BIG, BYTE or WORD. (See page 229 of the "Pascal Operating System Manual".)}

NREFS: INTEGER; {The number of references to this symbol in the compiled code segment. There will be this number of pointers after this record into the code segment. These specify the addresses of references to the symbol.}

NWORDS: LCRANGE; {where LCRANGE is 1..MAXLC, currently MAXINT. This field is meaningful only in the case of a PRIVREF type in which case it is the size of the privates in words.}

GLOBDEF:

(HOMEPROC: PROCRANGE; {which procedure the global definition appears in.}

ICOFFSET: ICRANGE); {The byte offset of the occurence in assembly language. IC stands for instruction count.}

PUBLDEF:

(BASEOFFSET: LCRANGE); {compiler assigned word offset into host program data segment.}

CONSTDEF:

(CONSTVAL: INTEGER); {User's defined value}

EXTPROC, EXTFUNC, SEPPROC, SEPFUNC:

(SRCPROC: PROCRANGE; {PROCRANGE = 1..MAXPROC. MAXPROC is currently 160. This field is the procedure number of this procedure definition in its source segment.}

NPARAMS: INTEGER); {Number of parameters expected (really number of words of parameters expected).}

EOFMARK:

(NEXTBASELC: LCRANGE; {Private variable allocation information— amount of space the host used in its data area. Meaningful only for host segments.}

PRIVDATASEG: SEGNUMBER); {Data segment number associated with intrinsic unit code segment. Otherwise not used.}

If the LITYPE is one of the first case variants, then following this portion of the record is a list of pointers into the code segment. Each of these pointers is the absolute byte address within the code segment of the reference to the variable, UNIT or routine named in the LIENTRY. This pointer list is contained in eight-word records, but only the first ((NREF-1) MOD 8)+1 words of the last record are valid.

APPENDIX: SUMMARY OF IMPORTANT RECORD DEFINITIONS

I. SEGMENT DICTIONARY: BLOCK ZERO OF A CODEFILE

```
RECORD

      DISKINFO:  ARRAY[0..15] OF
         RECORD
            CODELENG, CODEADDR:  INTEGER
         END;

      SEGNAME:  ARRAY[0..15] OF PACKED ARRAY[0..7] OF CHAR;

      SEGKIND:  ARRAY [0..15] OF
                (LINKED,
                HOSTSEG,
                SEGPROC,
                UNITSEG,
                SEPRTSEG,
                UNLINKED-INTRINS,
                LINKED-INTRINS,
                DATASEG);

      TEXTADDR:  ARRAY[0..15] OF INTEGER;

      SEGINFO:  PACKED ARRAY[0..15] OF
            PACKED RECORD

            SEGNUM:  0..255
            MTYPE:  0..15;
            UNUSED:  0..1;
            VERSION:  0..7

            END;

      INTRINS-SEGS:  {ON THE ][} SET OF 0..31;
                     {ON THE ///} SET OF 0..63;

      INT-NAM-CHECKSUM:  {Only on the ///}
         PACKED ARRAY[0..63] OF 0..255;

         {These words are not used on the ][; they must be zeroed.}

      {UNUSED JUNK (FILLED WITH ZEROES) FOLLOWED BY}

      COMMENT:  PACKED ARRAY [0..79] OF CHAR
         {the text following a Comment compiler option, starting in byte 432 of
         the header}

      END;
```

II. LINKER INFORMATION

The linker information is a series of records, one for each unit, routine or variable which is referenced but not defined in the source as well as records for items defined to be accessible from other modules. There are records for the following types of items:

```
litypes =

    ( EOFMARK,  {end-of link-information marker}
    UNITREF, {references to invisibly used units.}
    GLOBREF, {references to external global addresses.}
    PUBLREF,  {references to a variable in the global data segment of the
       host program.}
    PRIVREF,  {references to variables of an assembly language routine to
       be stored in the host program's global data segment and yet be
       inaccessible to the host program.}
    CONSTREF, {references to a globally declared constant in the host
       program.}
    GLOBDEF,  {Global address location.}
    PUBLDEF,  {A variable location in the host program.}
    CONSTDEF,  {A host program constant definition.}
    EXTPROC, { References to procedure declared to be external in
       Pascal.}
    EXTFUNC, { References to function declared to be external in
       Pascal.}
    SEPPROC, {Separate Procedure definition to be linked into Pascal.}
    SEPFUNC, {Separate Function definition to be linked into Pascal.}
    SEPPREF, {Not currently used.}
    SEPFREF); {Not currently used.}
```

The exact format of data in the linker information block is dependent on the type of entity. They are described by the following record.

```
LIENTRY = RECORD

   NAME:  PACKED ARRAY[0..7] OF CHAR; { The name of the symbol.  }

   CASE LITYPE:  LITYPES OF

      GLOBREF,
      PUBLREF,
      PRIVREF,
      CONSTREF,
      UNITREF,

      SEPPREF, {Not currently used}
      SEPFREF:  {Not currently used}

         (FORMAT:  OPFORMAT; {The format of the operand represented by
            the named (and currently undefined) symbol.  May be BIG, BYTE
            or WORD.)}

         NREFS:  INTEGER; {The number of references to this symbol in the
```

compiled code segment. There will be this number of pointers after this record into the code segment. These specify the addresses of references to the symbol.}

NWORDS: LCRANGE; {where LCRANGE is 1..MAXLC, currently MAXINT. This field is meaningful only in the case of a PRIVREF type in which case it is the size of the privates in words.}

GLOBDEF:

(HOMEPROC: PROCRANGE; {which procedure the global definition appears in.}

ICOFFSET: ICRANGE); {The byte offset of the occurence in assembly language. IC stands for instruction count.}

PUBLDEF:

(BASEOFFSET: LCRANGE); {compiler assigned word offset into host program data segment.}

CONSTDEF:

(CONSTVAL: INTEGER); {User's defined value}

EXTPROC, EXTFUNC, SEPPROC {not used}, SEPFUNC {not used}:

(SRCPROC: PROCRANGE; {PROCRANGE = 1..MAXPROC. MAXPROC is currently 160. .This field is the procedure number of this procedure definition in its source segment.}

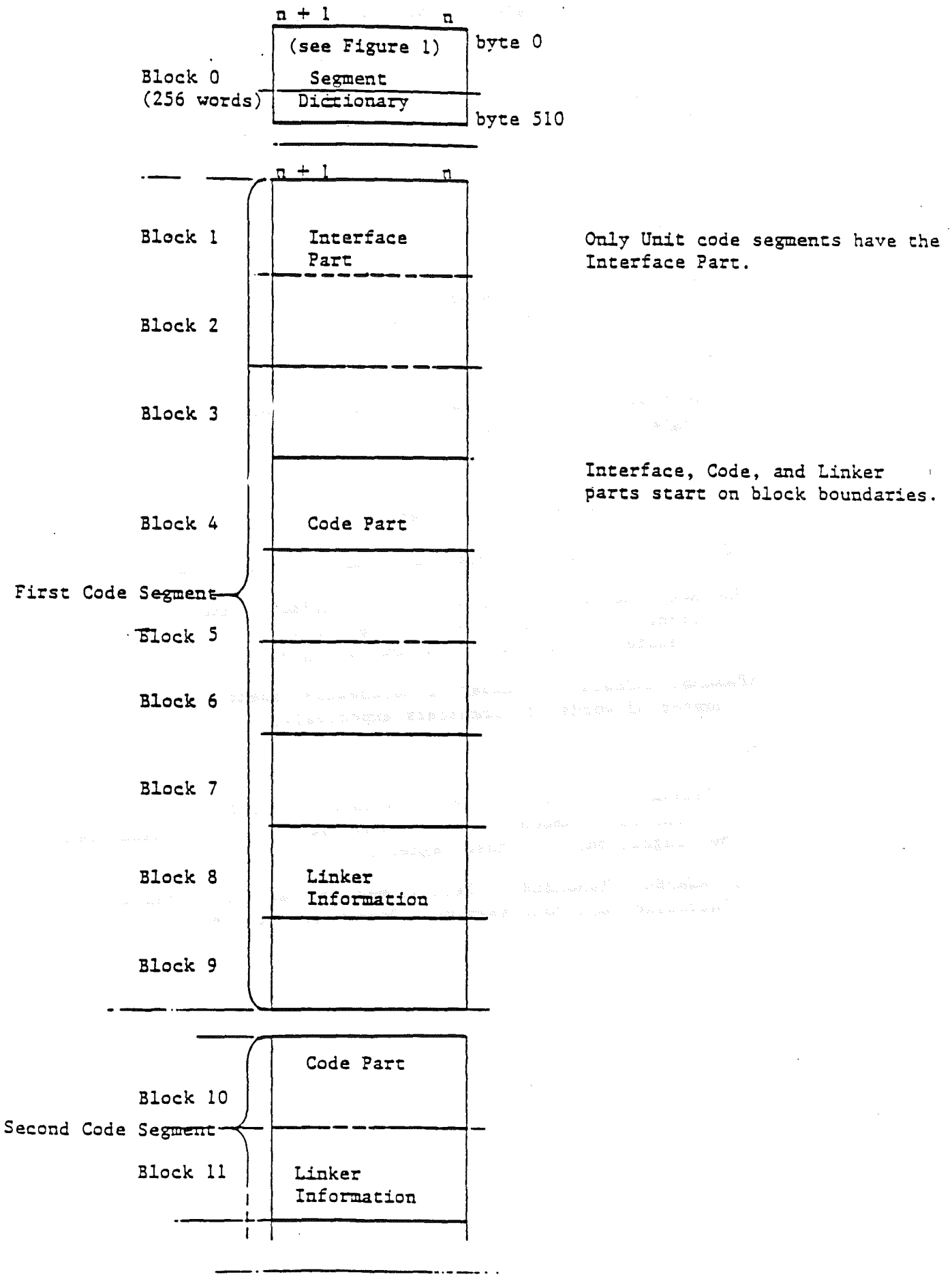
NPARAMS: INTEGER); {Number of parameters expected (really number of words of parameters expected).}

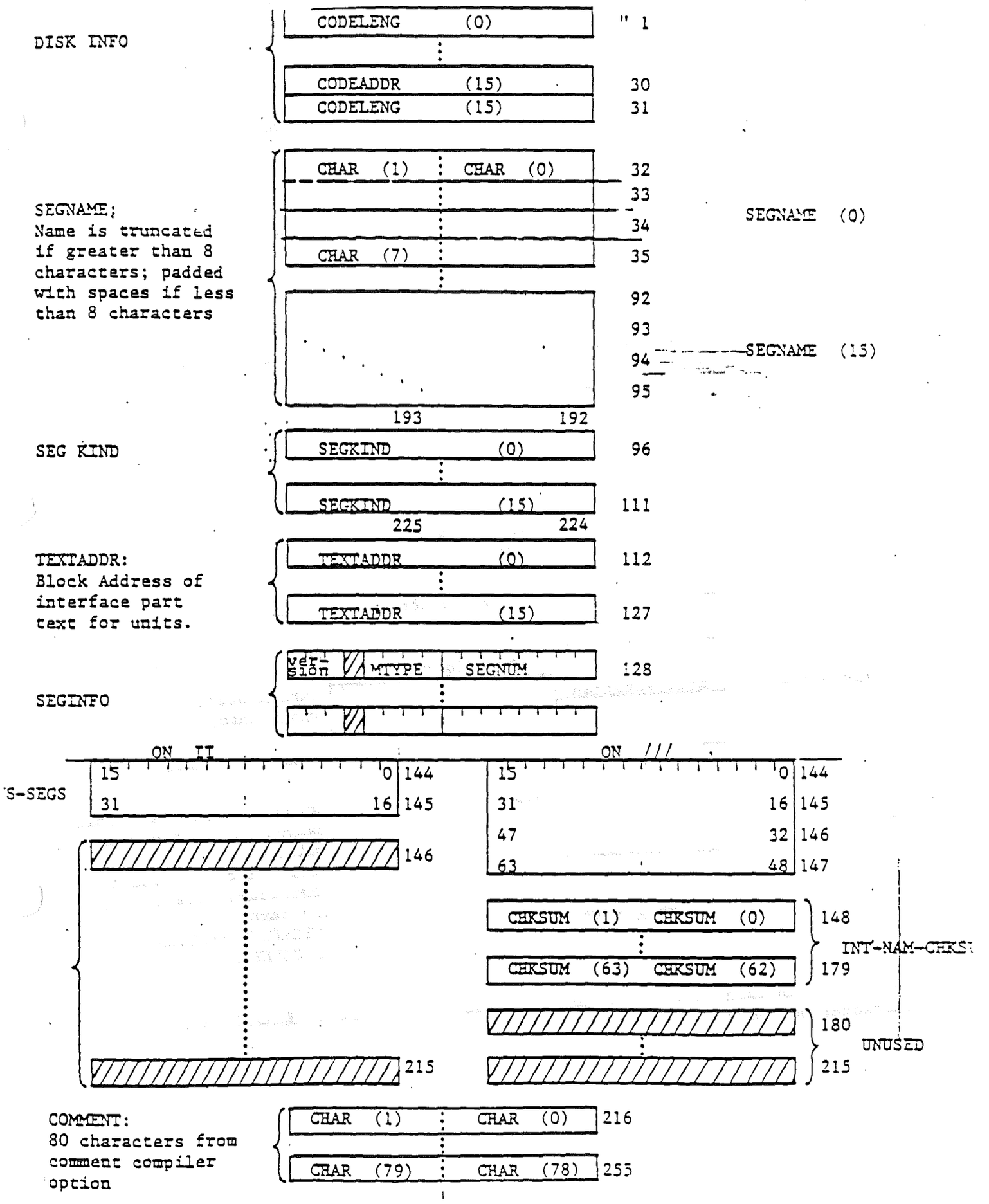
EOFMARK:

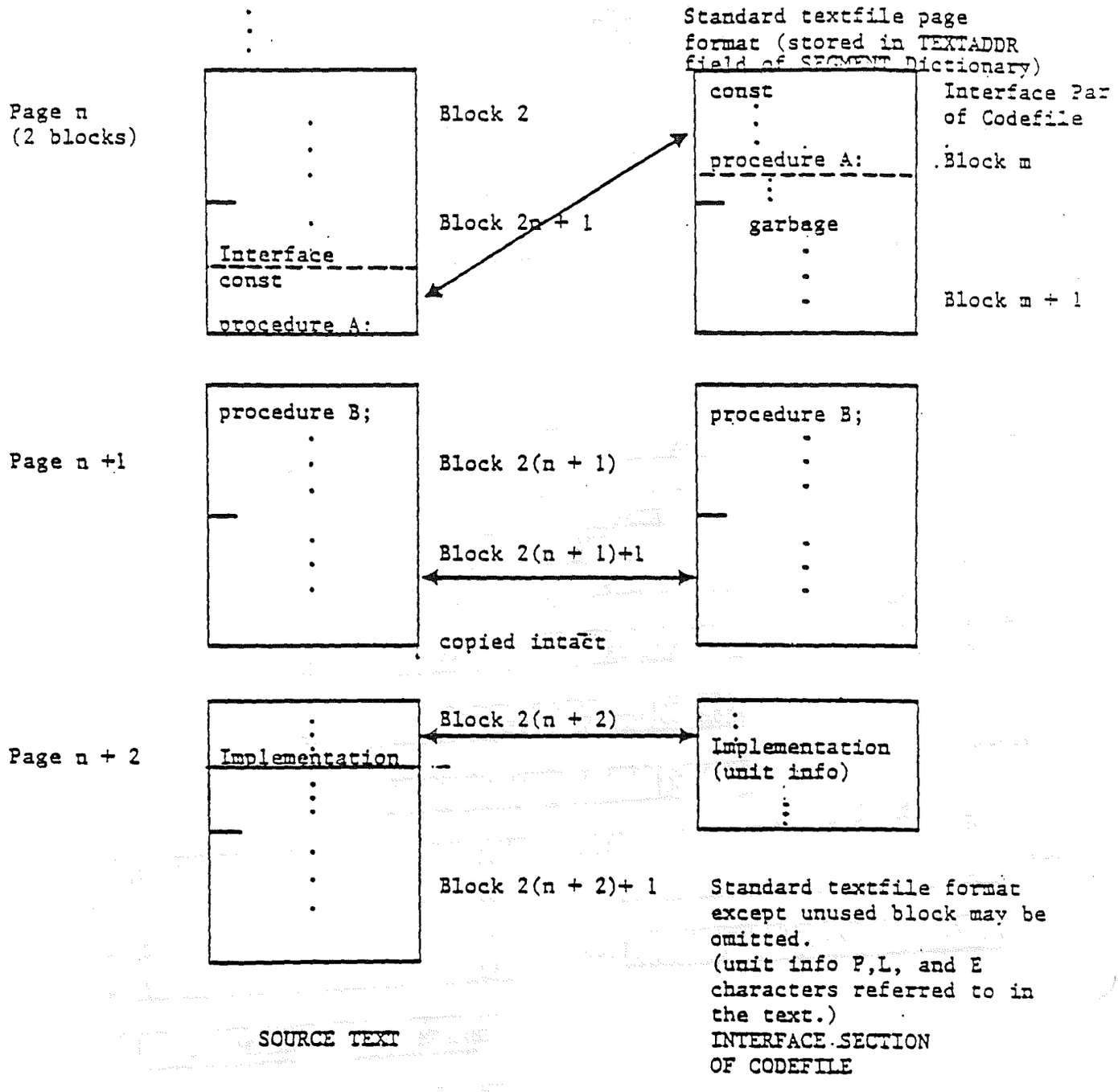
(NEXTBASELC: LCRANGE; {Private variable allocation information— amount of space the host used in its data area. Meaningful only for host segments.}

PRIVDATASEG: SEGNUMBER); {Data segment number associated with intrinsic unit code segment. Otherwise not used.}

FIGURE 0: THE CODEFILE ON DISKETTE:
A TYPICAL CODEFILE




DISK INFO
CODELENG (0)
" 1
CODEADDR (15)
30
CODELENG (15)
31
SEGNAME;
Name is truncated if greater than 8 characters; padded with spaces if less than 8 characters
CHAR (1)
CHAR (0)
32
33
34
CHAR (7)
35
SEGNAME (0)
92
93
94
95
SEGNAME (15)
193
192
SEG KIND
SEGKIND (0)
96
SEGKIND (15)
111
225
224
TEXTADDR:
Block Address of interface part text for units.
TEXTADDR (0)
112
TEXTADDR (15)
127
SEGINFO
version
MTYPE
SEGNUM
128
ON II
ON ///
S-SEGS
15
0
144
31
16
145
146
215
15
0
144
31
16
145
47
32
146
63
48
147
CHKSUM (1)
CHKSUM (0)
148
CHKSUM (63)
CHKSUM (62)
179
INT-NAM-CHKS
180
215
UNUSED
COMMENT:
80 characters from comment compiler option
CHAR (1)
CHAR (0)
216
CHAR (79)
CHAR (78)
255



Valid data in each block of a text file end with an ASCII 13,ASCII null character sequence.

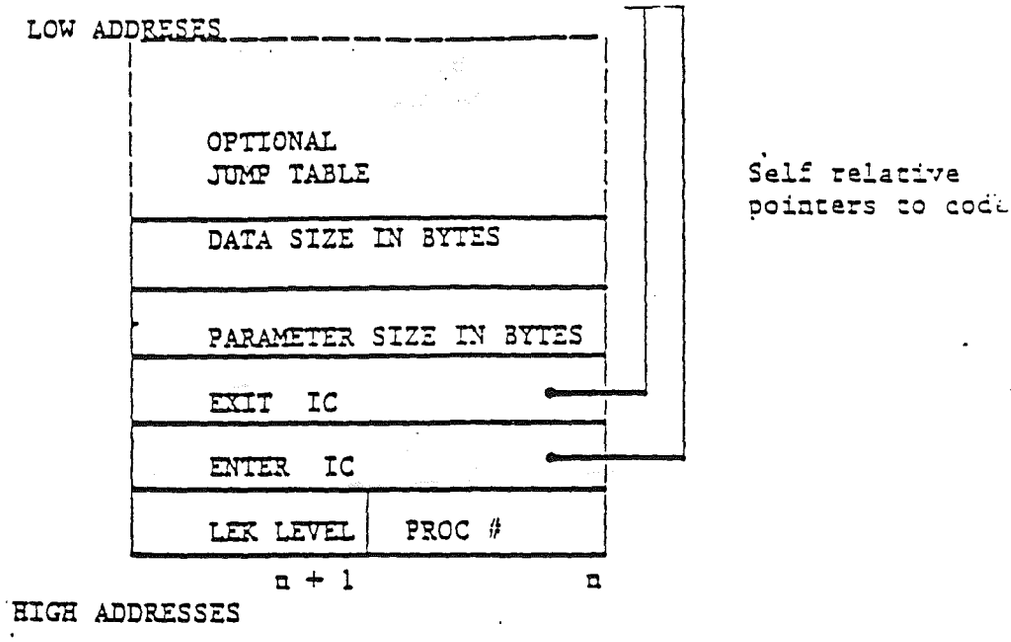
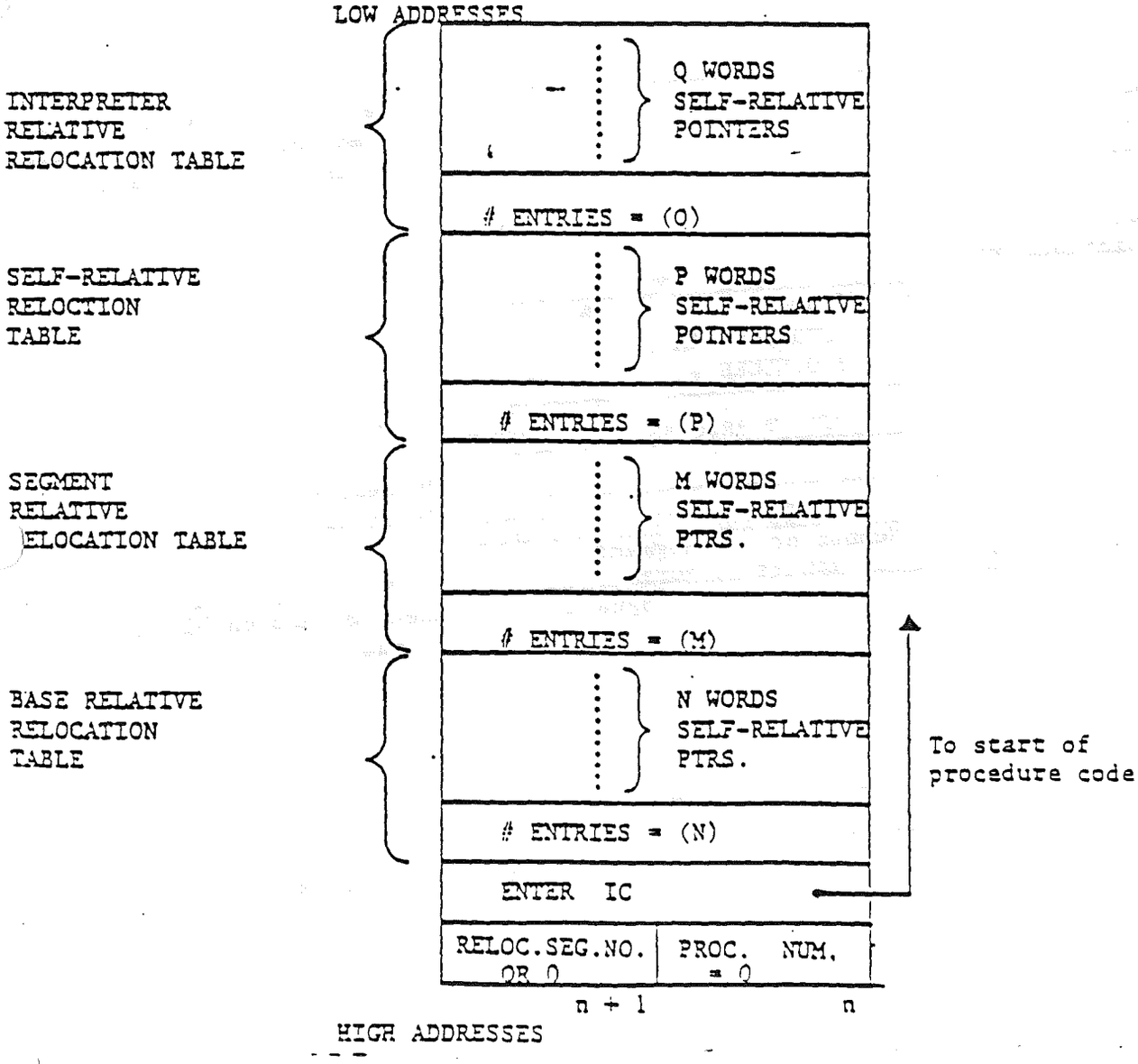
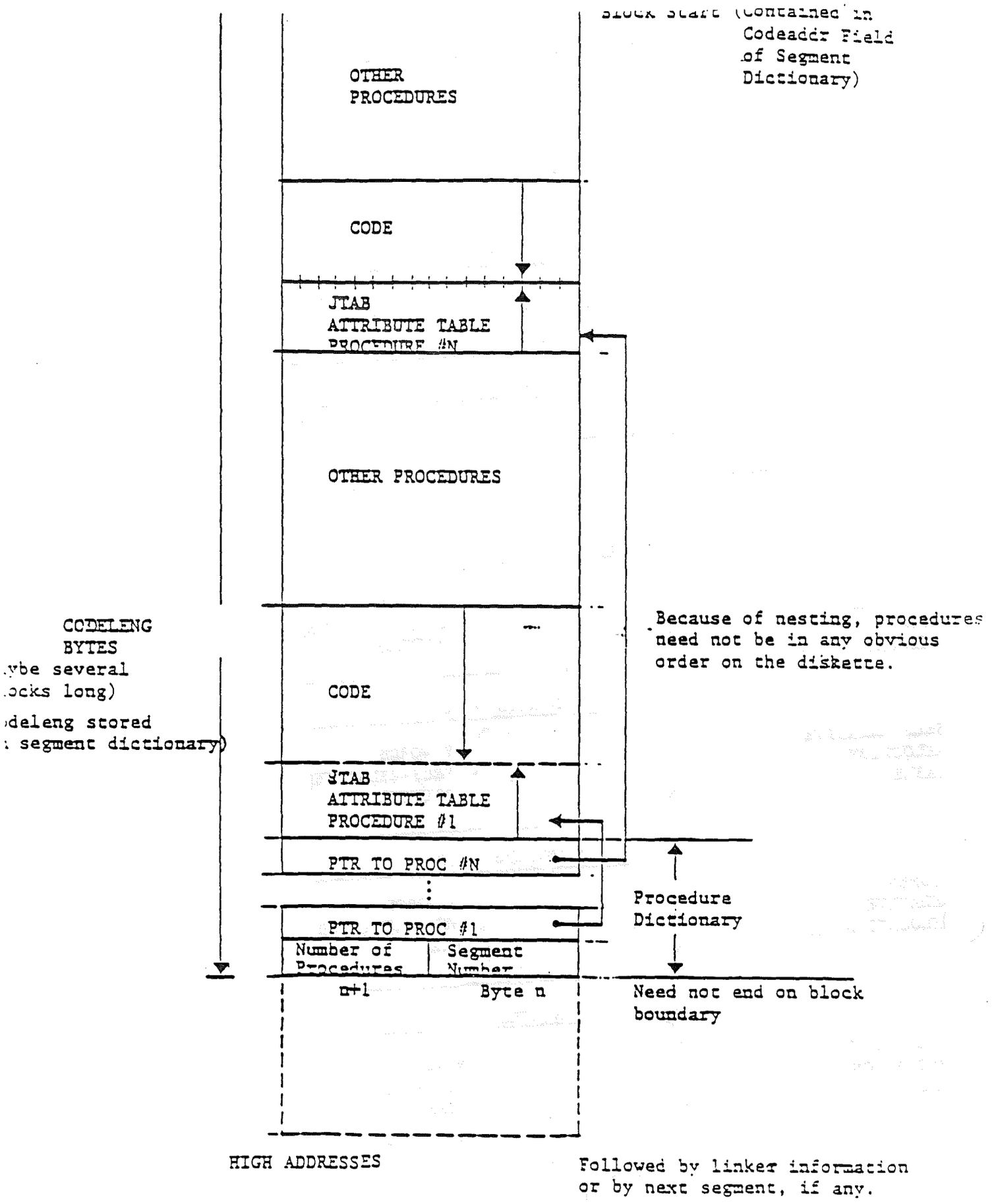


FIGURE 5: TLA ASSEMBLY LANGUAGE PROCEDURE ATTRIBUTE TABLE




Block Start (Contained in Codeaddr Field of Segment Dictionary)
OTHER PROCEDURES
CODE
JTAB
ATTRIBUTE TABLE
PROCEDURE #N
OTHER PROCEDURES
CODELENG BYTES
ybe several
ocks long)
deleng stored
segment dictionary)
CODE
JTAB
ATTRIBUTE TABLE
PROCEDURE #1
PTR TO PROC #N
PTR TO PROC #1
Number of Procedures
Segment Number
n+1
Byte n
Because of nesting, procedures need not be in any obvious order on the diskette.
Procedure Dictionary
Need not end on block boundary
HIGH ADDRESSES
Followed by linker information or by next segment, if any.

APPLE COMPUTER, INC.
20525 Mariani Avenue
Cupertino, CA 95014

PASCAL TECHNICAL NOTE #20
APPLE II PASCAL 1.2
VOLUME MANAGER UNIT TECHNICAL SPECIFICATION
(January 1984)

For further information contact:
PCS Developer Technical Support
M/S 22-W, Phone (408) 996-1010

Disclaimer of All Warranties and Liabilities

Notice

INTRODUCTION

Apple // Pascal release 1.2 only supports the UCSD file format. With the introduction of the Profile for use with the Apple //, the ProDOS operating system has been chosen as the operating system for the support of large mass storage devices. Clearly, the UCSD file format and thus the use of Pascal with a Profile is severely limited (i.e. non-existent) unless there is some means for Pascal to share the resources of the Profile with ProDOS.

The Pascal Profile Manager is a collective term for a set of programs that allow Pascal to share the Profile with ProDOS. These programs supply both user and programatic means to allocate and deallocate Pascal space on the Profile and to assign UCSD file format volumes (known as pseudo-volumes) to the Pascal area of the disk. These pseudo-volumes act analogously to the standard UCSD volumes that currently are found on floppies.

The Volume Manager Unit is a programatic means by which an application program can take advantage of the Profile for the storing of both data and code files. Use of the Volume Manager Unit assumes that the end-user has the PPM program and has already created a Pascal Area on his/her Profile. This unit allows a program to create and manage pseudo-volumes on a Profile. Its functions are:

a. Create a Pascal pseudo-volume.
b. Delete a Pascal pseudo-volume.
c. Assign a pseudo-volume for use.
d. Release a psedo-volume from use.
e. Set or clear write-protection for a pseudo-volume.
f. "Krunch" the Pascal region of the Profile to give space back to ProDOS.
g. Modify the name and/or description field of a pseudo-volume
h. Select the Profile drive to act upon
i. Get the current contents of the Pascal area volume directory
j. Get the current contents of the Profile driver status record
k. Volume Display and Error Reporting

PASCAL USAGE OF THE PROFILE

1. The Pascal Area

The Pascal area of the disk is a contiguous set of blocks that occupies the highest end of the disk, i.e. highest block number down to that block whose number is equal to highest block number minus the total number of blocks that that the Pascal region occupies. This area is not static but expands and contracts as pseudo-volumes are created or deleted and the region is krunched. To insure that Pascal can freely expand, it is a "requirement" that the blocks just below the Pascal region be available and that they be contiguous. Currently a problem may arise if ProDOS has fragmented the disk such that

there may be enough logical space for Pascal but not enough contiguous physical space.

The Pascal area is divided into two areas. The first is the Pascal volume directory that specifies the currently allocated Pascal pseudo-volumes in the Pascal area. The second is the pseudo-volumes themselves, each of which having its own volume directory (UCSD format) and its accompanying files.

2. Modifications to the Prodos Directory

The PPM accesses the ProDOS volume directory when it initializes the Profile for use by Pascal. It makes two changes to the directory contents.

The first change is a file entry that specifies the Pascal area on the disk. This file entry is placed in the first available entry slot in the ProDOS volume directory. An error will occur if there is no available slot to put this entry.

Once this slot has been made available, PPM will initialize it with a file entry with the following contents:

```
Stype = 4     this is a ProDOS foreign file structure
name_length = 10
file_name = 'PASCAL.AREA'
file_type = 0EFH  this is a special type to denote the Pascal area
key_pointer = first block used (in this case the second to the
              last block on the disk)
blocks_used = 2
header_pointer = 2
access = 0 (backup bit is not set)
```

All other fields are set to 0. PPM will look for this entry (primarily the name 'PASCAL.AREA') in the ProDOS directory to determine if the disk has been initialized for Pascal use.

The file entry for the Pascal area increments the number of files in the ProDOS directory and the key_pointer for this file now points to TOTAL_BLOCKS - 2. Thus the Pascal area occupies the last two blocks available on the Profile. Blocks_used in the file entry is set to 2. When the Pascal area expands or contracts, the key_pointer and blocks_used values are updated accordingly. With any access to this file entry (i.e. if the Pascal area is expanded or contracted by adding or deleting pseudo-volumes) the backup bit will not be set. However, a ProDOS based Backup program can explicitly backup the Pascal area as a whole. At any time that it cannot expand due to ProDOS using the required blocks, an error is reported. Because ProDOS can fragment its area on the Profile, it is quite possible for Pascal to be unable to expand, though there is logically enough room on the disk to do so. Currently, the only means to correct this is to have the user do the following:

a. backup the Pascal region
b. backup the ProDOS region
c. reformat the disk

d. restore the ProDOS region
e. restore the Pascal region
f. get back to real work

3. Volume Directory Format

The Pascal volume directory contains two separate but contiguous data structures that specify the contents of the Pascal area on the Profile. The volume directory occupies 2 blocks to support 31 pseudo-volumes. It is found at the physical block specified in the ProDOS volume directory as the value of KEY_POINTER, i.e. it occupies the first block in the area pointed to by this value. To access the Pascal area volume directory requires reading the ProDOS volume header via a UNITREAD of block 2, getting the value of KEY_POINTER and using this in a UNITREAD of block number KEY_POINTER. The volume manager maintains a 1K buffer to read in this directory. It is important to define the directory data structures in the volume manager as contiguous to insure that the data read in is interpreted correctly.

The first portion of the volume directory is the actual directory for the pseudo-volumes. It is an array with the following declaration:

```
TYPE  RTYPE = (HEADER, REGULAR)

VAR   VDIR:  ARRAY [0..31] OF
               PACKED RECORD
                 CASE RTYPE OF
                   HEADER: (PSEUDO_DEVICE_LENGTH:INTEGER;
                            CUR_NUM_VOLS:INTEGER;
                            PPM_NAME: STRING[3]);
                   REGULAR: (START:INTEGER;
                             LENGTH:INTEGER;
                             DEFAULT_UNIT:0.255;
                             FILLER:0..127;
                             WP:BOOLEAN;
                             OLDDRIVERADDR:INTEGER
               END;
```

The HEADER specifies information about the Pascal area. It specifies the size in blocks in PSEUDO_DEVICE_LENGTH, the number of currently allocated pseudo-volumes in CUR_NUM_VOLS, and a special validity check value in PPM_NAME, which is a three character string containing the value 'PPM'. The header information is accessed via a reference to VDIR[0]. The REGULAR entry specifies information for each pseudo-volume. START is the starting block address for the pseudo- volume and LENGTH is the length of the pseudo-volume in blocks. DEFAULT_UNIT specifies the default Pascal unit number that this pseudo-volume should be assigned to upon booting the system. This value is set by the volume manager either by the user or an application program and remains valid if it is not released. If the system is shut down, the pseudo-volume will remain assigned and will be active once the system is rebooted. WP is a Boolean that specifies if the pseudo-volume is write- protected. OLDDRIVERADDR holds the address of this unit's (if assigned) previous driver address. It is used when normal floppy unit numbers are assigned to pseudo-volumes so that when released the floppies can be activated again. Each

REGULAR entry is accessed via an index (from 1 to 31). This index value is thus associated with a pseudo-volume. All references to pseudo- volumes in the volume manager are made with these indexes.

Immediately following the VDIR array is an array of description fields for each pseudo-volume:

VDESC: ARRAY [0..31] OF STRING[15]

The description field is used to differentiate pseudo-volumes with the same name. It is set when the pseudo-volume is created. This array is accessed with the same index as VDIR.

The volume directory does not maintain the names of the pseudo- volumes. These are found in the directories in each pseudo-volume. When the volume manager is activated, it reads each pseudo-volume directory to construct an array of the pseudo-volume names:

VNAMES: ARRAY [0..31] OF STRING[7]

Each pseudo-volume name is stored here so that the volume manager can use it in its display of pseudo-volumes. The name is set when the pseudo-volume is created and can be changed by the Pascal filer. The names in this array are accessed via the same index as VDIR. This array is set up when the volume manager is initialized and after there is a delete of a pseudo-volume. Creating a pseudo-volume will add to the array at the end.

4. Pascal Pseudo-Volume Format

Each Pascal Pseudo-volume is a standard UCSD format volume. Block 0 and 1 of the pseudo-volume are reserved for bootstrap loaders (which in this case are irrelevent!). The directory for the volume is in blocks 2 through 5 of the pseudo-volume. When a pseudo-volume is created the directory for that pseudo- volume is initialized with the following values:

dfirstblock = 0 first logical block of the volume

dlastblock = 6 first available block after the directory

dvid = name of the volume used in create

deovblk = size of volume specified in create

dnumfiles = 0 no files yet

dloadtime = set to current system date

dlastboot = 0

The Pascal Tech Note #4 describes the format for the UCSD directory.

Files within this subdirectory are allocated via the standard Pascal I/O routines in a contiguous manner.

5. Volume Name Format

A valid Pascal UCSD format volume name may be up to seven characters in length and can include any printable ASCII character except ' ', '=', '$', '?', and ','. When ever a user is prompted to enter a volume name, they may enter it in either upper or lower case, however, all lower case letters will be forced to upper case before this volume name is used. For example, if the user enters 'death' as a volume name, it will be uppercased to 'DEATH'.

6. System Limitations

1. Number of Profiles Supported

The Profile driver will currently support only three Profiles, because a Profile interface card can only be plugged into slots 4, 5, or 7 in an Apple //.

The unit numbers used by the Pascal system to refer to the Profiles must be in the range 128 to 143. If they are not, the volume manager will not be able to find them and an error (No Profiles on the system) will result.

2. Number of Pseudo-volumes per Profile Supported

The number of Pascal Pseudo-volumes supported per Profile is limited to 31. This is due to the limitation as to the size of the Pascal area Volume directory. The volume directory is accessed by the Profile driver at boot time in order to assign the default pseudo-volumes. Extending the number of pseudo-volumes supported will require that the driver be changed in order to handle a larger volume directory. Currently the volume directory is 256 bytes.

3. Number of Pseudo-volumes Selected

The Profile driver will allow up to 30 pseudo-volumes to be mounted at any one time. This limit is imposed by the Pascal system as to its number of allowed units. It is reflected in the driver in the data structure STATUS_RECORD. To increase this number requires a change to the Pascal system and to both the Profile Driver and SYSTEM.ATTACH.

4. Pascal Blocked Device Volumes versus User Devices

The Pascal supports the following device numbers as "blocked devices". This implies that they may be accessed like floppies via RESET, REWRITE, READLN, etc.

Blocked Device Unit Numbers

4, 5, 9 - 20

The following unit numbers are for "User devices". They

can only be accessed via UNITREAD and UNITWRITE, which implies that Pascal files are not supported for these devices.

User Device Unit Numbers

128 - 143

Thus this system will only support 14 blocked devices on-line at any time. The other 16 volumes are only useful for programs that do their own physical I/O to these volumes. Any floppies attached to the system will use some of the blocked devive unit numbers which leaves fewer of these for pseudo-volumes on the Profile. A user may assign the normal floppie device number to pseudo-volumes, but this will effectively make these floppies inaccessible for use until the pseudo-volumes are released.

5. Volume Name Conflicts

This design allows a user to designate pseudo-volumes with the same name on a single Profile. Many applications may require pseudo-volumes that have the same name, i.e. DATA. In order to support this requirement, we must allow multiple pseudo-volumes with the same name, however, there must be a way to differentiate them both for the user using the Volume Manager Program and for an application program assigning and releasing pseudo-volumes programmatically. To do this, each pseudo-volume entry in the volume directory has an associated description field which is 15 characters in length. This is much the same as extending the volume name by 15 characters.

In order to have pseudo-volumes with the same name on a single Profile, they must have different description fields. For example,

name	description
DATA	QUICKFILE
DATA	PFSREPORT
DATA	MY LIFE STORY

When a pseudo-volume is created this field is specified. When ever a specific pseudo-volume is to be referenced by name, the description field contents are used to differentiate between those with the same name. A user will simply point to the pseudo-volume via cursor motion, using the description field contents displayed as a mnemonic device helping he/she to know which volume is which. A program must pass the expected description contents to the volume manager so it can decide which is which. The rules for finding pseudo-volumes are:

1. If there is only one pseudo-volume with the the name requested then act on that pseudo-volume.

2. If there are more than one pseudo-volume with the name requested, then match description fields, return the one matched. If no description content is supplied, return an error.

The volume manager will not allow a user or a program to create pseudo-volumes which cannot be differentiated.

For new applications, it will be important to document how to create and copy their floppy volumes into pseudo-volumes. In this case, the description field will be used by users when hand-assigning Pascal unit numbers prior to execution of the application. Applications that use the volume manager unit can specify this description itself and when assigning unit numbers it can use it to find its own pseudo-volumes.

6. Unit number Conflicts - #4 and #12

Pascal normally allocates its blocked device unit numbers (4, 5, 9 - 12; 13 - 20 are new with Pascal 1.2) to floppies. Unit #4 is normally the floppy drive used to boot the system. It is also by definition, the Pascal system disk which can be referenced via '*'. It is possible to assign this (and any other unit number) to a pseudo-volume. If a user assigns what is normally a floppy drive unit number to a pseudo-volume, they have effectively made that floppy unusable until such time as they release the unit number.

If unit #4 is assigned to a pseudo-volume, the volume manager will then assign unit #12 to the device that was assigned to unit #4. In the usual case, this will be the original boot floppy drive. By doing so, this floppy drive will remain accessible. If unit #4 is assigned and unit #12 is currently not assigned, then unit #12 will be assigned automatically to the device that is normally assigned to be unit #4. Conversely, when unit #4 is released, unit #12 which has been re-assigned will be put back to its original (default) device. If unit #12 is currently assigned (to a pseudo-volume) it will be released and assigned to the normal unit #4 floppy drive. It this case when unit #4 is released, unit #12 will be released from the floppy drive unit, BUT it will NOT be reassigned to its previously assigned pseudo-volume.

This scheme has been adopted because unit #12 is normally assigned to the sixth floppy drive device, which normally does not exist. It will be common practice to assign unit #4 to a pseudo-volume in order to use it as the system "disk" on the Profile. Re-assignment of unit #12 when it has been released from the normal unit #4 floppy drive will only take place if unit #12 was previously assigned to a device, which implies that it has its own driver. Assignment of unit #12 to a pseudo-volume can not be restored.

NOTE: If a user assigns unit #4 to a pseudo-volume (which causes unit #12 to be assigned to the boot floppy device) and then assigns unit #12 to a pseudo-volume, this action will make the boot floppy device inaccessible until unit #4 is released.

IMPORTANT NOTE: Assigning a pseudo-volume to unit #4 will immediately release the current unit #4, making it unavailable for use. This may have serious effects. If a program is running that has been invoked from unit #4 (for example, the PPM/volume manager program) and a pseudo-volume is assigned to unit #4, the Pascal operating system will request that the user put in the disk that is in the normal unit #4 device when they exit the program. This is because the system requires the program to be on-line to exit and because the program has been put off-line by the assignment of unit #4. The only recourse is for the user to re-boot the system. Assignment of unit #4 should be done with some thought. Also, if the user intends to place his/her system volume (Pascal development system) in a pseudo-volume and assign this pseudo-volume to unit #4, they must insure that the files for the Pascal operating system occupy the same logical blocks in the pseudo-volume as they occupy on the boot diskette.

If the user assigns a unit number that corresponds to a device that has been configured into the system via ATTACH, that device will become unavailable. Any product that assumes a unit number for a device should warn the user not to assign that unit number to a pseudo-volume when that device must be used.

7. Default Assigning of Pseudo-volumes

The volume directory maintains a mapping of pseudo-volumes to their currently assigned Pascal unit numbers. This assumes that a Pascal area has been initialized, pseudo-volumes have been created in it, and some number of them have been assigned to unit numbers and have not been officially released, i.e. the system has been shutdown without ever releasing these pseudo-volumes. Whenever the system is booted, the Profile driver when activated will read the volume directory from the Profile to determine if and what pseudo-volumes are currently assigned. It will prompt the user

Assign volumes to their default unit number? (Y/N)

and if the user types 'Y' the driver will update its status record to effectively assign these pseudo-volumes to their unit numbers. If the user types 'N' they will not be assigned and will not be accessible.

8. Profile Driver Status Record

The Profile driver maintains a status record that maps Pascal pseudo-volumes to Pascal unit numbers. When a pseudo-volume is assigned, the status record is updated to reflect the assignment. The status record is an array that is mapped into the standard Pascal unit numbers via the mapping

PASCAL UNIT NUMBER	INDEX
4	1
5	2
9	3
.	.
.	.
20	14
128	15
.	.
.	.
143	30

The format of the status record is shown below:

```
STATUS_RECORD = ARRAY [1..30] OF
                PACKED RECORD
                  DRIVE: 0..7;
                  DFMT_DRIVE: 0..7;
                  FILL1: 0..255;
                  WRITE_PROTECT: BOOLEAN;
                  PRESENT: BOOLEAN;
                  START: INTEGER;
                  LGTH: INTEGER;
                END;
```

When a pseudo-volume is assigned/released, write-protected, and at boot time this status record is updated. Each entry in the status record corresponds to a Pascal Unit number. The field PRESENT, if a 1, connotes that this unit number is assigned. The field DRIVE specifies the Profile drive on which the pseudo-volume resides, START gives the physical block number of the starting block of the pseudo-volume, and LGTH is the length of the pseudo-volume in blocks. WRITE_PROTECT, if a 1, implies that this pseudo-volume is write-protected. DFMT_DRIVE is used to assign the last used drive when the volume manager program/unit is restarted. When the system is booted the default mount drive (DFMT_DRIVE) is set to 0. If the next drive command or SELECT_DRIVE procedure is called, this value is changed to reflect the new drive and stored in the Profile driver. When the volume manager is exited and then at some point re-invoked, it will read this value from the driver and use it as the current drive. If the system is shutdown, this value will revert to 0. This data structure is not intended to be accessed by any program other than the volume manager and the Profile driver itself.

9. Use of the Profile Driver

Both the PPM and the volume manager assume that there is a Profile driver attached and that the name of this driver is 'PROFILE'. At initialization time for both these programs, if no Profile driver is found (identified by its name 'PROFILE') then an error message is issued:

ERROR: There is no Profile driver available for this Pascal system

and the program will terminate.

The Profile Driver is supplied as the file ATTACH.DRIVERS and its associated data file is ATTACH.DATA. This driver is configured to be unit #128.

THE VOLUME DISPLAY

The volume display occupies the major portion of the screen and is used to display the pseudo-volumes available for use on the currently selected Profile drive. This display has two uses:

1. Display the pseudo-volumes available

2. Serve as the means to select a pseudo-volume upon which to apply one of the actions in the volume manager command line.

The format for the the volume display is shown below with example pseudo-volumes:

Profile drive: 0

WP	Name	Description	Unit	WP	Name	Description	Unit
*	DATA	QUICKFILE	#9	*	ACCOUNT	PFSREPORT	#134
	DATA	DBMSTUFF		*	DATA	PFSFILE	
	LETTERS		#13		FUN	SOME GAMES	
	PASSYS	PASCALSYS	#4		MOREFUN	NOT A GAME	
	PASDEVO	SOME TOOLS	#5				
	BOB	OUR SAVIOR					
	YHVH1	STARK FIST					
	AP	APPLEACCOUNT					
	GL	APPLEACCOUNT					
	AR	APPLEACCOUNT					
	<none>	PFSDATAVOL					
	TOOLS	MORE TOOLS	#15				
	TEXT	DOCUMENTS	#16				
	YETI						
	PICTURE	TURTLEGRAPHICS	#19				
*	RESUME	FUTURES					

This format will allow up to 31 pseudo-volumes to be displayed at one time. If there is less than 17 pseudo-volumes to be displayed, the right hand column header is suppressed.

The first line shows which Profile drive is active by giving the drive number (in this case 0). As other drives are selected, this number will change.

The fields in the display are described below:

WP - this is the write-protect attribute for the pseudo-volume. If it is write-protected, a '*' will be displayed in this column.

NAME - this is the name of the pseudo-volume. It can be up to 7 characters in length. Multiple pseudo-volumes can have the same name if and only if their description fields are different. It is possible for a pseudo-volume to not have a name, i.e. some applications use the entire volume for data wiping out the directory. If no name is found the string "<none>" is displayed.

DESCRIPTION - this is the description field for the pseudo-volume that helps to both differentiate it from others with the same name and also serve as a reminder to the user what the contents of that pseudo-volume are.

UNIT - if the pseudo-volume is currently mounted then its Pascal unit number will be displayed, else this field will be blank.

SELECTING A VOLUME

When an action that affects an individual pseudo-volume is selected from the prompt line, the characters '->' will appear next to the first pseudo-volume displayed. By using the up or down arrow keys (as defined by the Pascal system and machine in use) the user can move the pointer from one pseudo-volume to another. UP will move the cursor up on the screen and DOWN will move it down. The pseudo-volumes are 'numbered' from top to bottom with the first column 'numbered' from 1 to 16 and the second column 'numbered' from 17 to 32. UP moves down the 'numbers' and DOWN moves up the numbers!! If more than 32 pseudo-volumes are allowed in the display then multiple screen pages are used to display the pseudo- volumes. Movement between screen pages is done using the UP and DOWN arrow keys and a to be determined modifier key.

For a standard Apple // system, CTRL-O is defined be UP and CTRL-L is defined to be DOWN. This is the convention followed by Pascal on the Apple //. For the //e the up-arrow and down-arrow keys are respectively UP and DOWN.

Once the pointer has been moved to the desired pseudo- volume, typing a RIGHT ARROW will select that pseudo-volume for the action specified in responce to the prompt line. When a pseudo- volume is selected, it will be highlighted. The volume manager may ask for further prompting/information once RIGHT ARROW has been typed. When prompted, typing ESCAPE will cancel both pseudo-volume selection and the action selected from the prompt line.

The right-arrow key on both the Apple // and the //e corresponds to RIGHT ARROW.

THE VOLUME MANAGER UNIT SPECIFICATION

1. Introduction

The Volume Manager Unit (VOLUME_MANAGER) is a programatic interface, to allow developers to write programs that can manage Pascal pseudo-volumes on a Profile. It supplies the following generic capabilities through lower level procedure calls:

a. Create a Pascal pseudo-volume

b. Delete a Pascal pseudo-volume (this is not a recommended practice for application programs to do.)

c. Assign a Pascal Unit number to a pseudo-volume to make it available for use

d. Release a Pascal Unit number from a pseudo-volume

e. Set the write-protection attribute for a pseudo-volume

f. Krunch the Pascal region to give space on the Profile back to ProDOS (this is also not a recommended practice for applications to perform. This ability will be built in to the Delete call as well as being a stand alone procedure.)

g. Modify the name and/or description field of a pseudo-volume.

h. Select the Current Profile drive on which to perform the above actions (an application program would not normally have to do this except to search for a pseudo-volume that it needs to assign.)

i. Get the contents of the Pascal area volume directory. This is for information purposes only. A program cannot change its contents.

j. Get the contents of the status record in the Profile driver. Again this is for information purposes only.

k. Utilize the volume display (section 4.4.2 and 4.4.3) to display and select pseudo-volumes.

l. Error reporting.

The user has the option to do their own screen management for input and/or error reporting.

An application program cannot initialize a Pascal region on a Profile. This must be done via the Pascal Profile Manager by the end-user. Applications that require this action will need to document this requirement so that the user of the application can correctly set up the Profile for use.

The volume manager unit is a REGULAR unit and must be linked to a host program.

2. Volume Manager Unit Interface

1. Constants

MAX_VOLS

This is the maximum number of pseudo-volumes that can be allocated in the Pascal area on a Profile. This number is 31.

MAX_DRIVE

This is the highest drive number for use in referencing the Profile drives. This number is currently 7, but only drives 0, 1, 2 are supported.

VDIR_SIZE

This is the size of the volume directory in blocks. For 31 pseudo-volumes its value is 2.

MAXDUNIT

This constant represents the highest unit number for blocked devices which is 20.

2. Types

UNIT_RANGE

This is the range of unit numbers supported by the Pascal system. The range is 0 to 255.

RTYPE

This is used to differentiate the two types of record fields in the volume directory. The two types are HEADER which refers to the header information

in the volume directory and REGULAR which refers to the entry used for each pseudo-volume in the directory.

DRIVE_RANGE

This is the range of values for drive numbers used to reference Profile drives. The range is 0 to MAX_DRIVE.

STAT_REC

This is the declaration for the data structure STATUS_RECORD that is found in the Profile driver. It maintains information about the currently assigned Pascal unit numbers. Its format is:

```
STAT_REC = ARRAY [1 .. 30] OF
             PACKED RECORD
               DRIVE: 0 .. 7;
               DFMT_DRIVE: 0 .. 7;
               FILL1: 0 .. 255;
               WRITE_PROTECT: BOOLEAN;
               PRESENT: BOOLEAN;
               START: INTEGER;
               LGTH: INTEGER;
             END;
```

This structure is described above.

VDIR_STRUCT

This is the format for the volume directory. Its structure is:

```
VDIR_STRUCT = ARRAY [0 .. MAX_VOLS] OF
               PACKED RECORD
                 CASE RTYPE OF
                   HEADER: (PSEUDO_DEVICE_LENGTH:INTEGER;
                            CUR_NUM_VOLS:INTEGER;
                            PPM_NAME:STRING[3]);
                   REGULAR: (START:INTEGER;
                             LENGTH:INTEGER;
                             DEFAULT_UNIT:UNIT_RANGE;
                             FILLER:0 .. 127;
                             WP:BOOLEAN;
                             OLDDRIVERADDR:INTEGER)
               END;
```

This data structure is fully described above.

DESC_ARRAY

This array holds the description fields for each pseudo-volume. It is important that any program that gets the volume directory contents must also declare

this data structure contiguous to the volume directory data structure. Its format is:

```
DESC_ARRAY: ARRAY [0 .. MAX_VOLS] OF STRING[15]
```

N_ARRAY

This array will hold the names of the pseudo-volumes. It also must be declared if the application program intends to get the volume directory contents. It does not have to be declared in any special place however. Its format is:

```
N_ARRAY: ARRAY [0 .. MAX_VOLS] OF STRING[7]
```

STRING7

This is a string of length 7. It should be used to declare any variable that is to hold a pseudo-volume name.

STRING15

This is a string of length 15. It should be used to declare any variable that us to hold a description field.

BLOCK_TYPE

This is a 512 element array of bytes that is used to hold blocks of data read in from a disk. It is primarily used for low-level routines and is not necessary for application programs.

3. Variables

VALID_DRIVE

This is a set that holds the valid drive numbers for all the available Profile drives. Its format is

```
VALID_DRIVE: SET OF DRIVE_RANGE
```

This variable is initialized when the volume manager is activated. If a drive number is in VALID_DRIVES it does not imply that this drive has a Pascal area. It only implies that this drive is active and that it has a ProDOS directory. An application program should use PASCAL_DRIVES to determine if this drive has a valid Pascal area.

PASCAL_DRIVES

This is the set that specifies all the available

Profiles that have Pascal areas. All of the volume manager unit functions can only be applied to Profiles that are specified in this set. Any application must check the drive number against this set prior to making any calls to the volume manager unit. Since SELECT_DRIVE must be called prior to making any other calls, it will check the drive number against this set and return an error if it is not in the set. A call to INIT_VM will put together both VALID_DRIVE and PASCAL_DRIVES. The format for this set is

PASCAL_DRIVES: SET OF DRIVE_RANGE

MY_UNIT

This is the unit number by which the Pascal system refers to the Profile driver. It is an integer.

ERR_LINE

This variable holds the line number on which errors are reported. Its value defaults to 3. An application program can change this value. It is only used when ERR_FMT (see below) is TRUE.

DISPLAY_ERR

This boolean variable is used to control whether or not the volume manager willl report errors to the screen. If TRUE, then errors will be reported, else they will not be reported.

ERR_FMT

If this boolean variable is TRUE then errors will be reported on ERR_LINE, else they will be reported on the current line of the display.

VM_ERROR

This integer variable will contain an error code if an error has occurred on a call to the volume manager. If it is 0, then no error has occurred.

VM_IO_ERROR

This integer variable will contain the value of IORESULT after any call to the volume manager. If it is 0 then no error has occurred.

CUR_DRIVE

This is the current drive number for the currently

accessible Profile unto which volume manager actions can occur.

CUR_INDEX

This is the index of the currently selected pseudo-volume on the current drive. It is only set via the volume selection routine SEL_VOLUME.

VDIR_BYTES

This is the size of the volume directory plus the description array in bytes. It is used in reading and writing the contents from and to the Profile. It is initialized by the volume manager unit.

VDIR

This is the current copy of the volume directory of the currently selected drive. It is initialized by SELECT_DRIVE.

VDESC

This is the current copy of the array of descriptions that corresponds to the pseudo-volumes of the currently selected drive. It is initialized by SELECT_DRIVE.

VNAMES

This is the current array of volume names for the pseudo-volumes of the currently selected drive. It is initialized by SELECT_DRIVE.

STATUS_REC

This is the current copy of the status record from the Profile driver. It is initialized by INIT_VM.

4. Procedures and Functions

CREATE_VOLUME

Call format:

INDEX := CREATE_VOLUME(NAME, DESC, SIZE)

where NAME is a 7 byte string that will be the name of the volume, DESC is a 15 byte string that denotes the description field (this may be null), and SIZE which is an integer that denotes the number of blocks

this pseudo-volume is to occupy. INDEX is a user-supplied integer to hold the index value that is returned.

CREATE_VOLUME will create a pseudo-volume on the currently selected Profile drive. It will be assigned a name, its description field will be specified, and it will be SIZE blocks in length. This function will then return an index value that must be used in any other call to act on this pseudo-volume. It is up to the calling program to save this index value. (It can be found however through a VOLUME_INDEX call described below.) If an error occurs, INDEX will be set to 0. Use of this function will change the index values that correspond to the pseudo-volumes on the Profile.

Errors reported:

a. Not enough room - there is not enough room in the Pascal region to allocate a pseudo-volume of this size or the Pascal region cannot expand into the ProDOS area. A Krunch may alleviate this problem.

b. Directory full - there is no more room in the volume directory to allocate this pseudo-volume.

c. Name conflict - a pseudo-volume with this name already exists and the description field does not differentiate them. This can be solved either by specifying the description field or changing it.

d. Illegal volume name

e. Volume size must be greater than 6 blocks.

DELETE_VOLUME

Call format:

DELETE_VOLUME(INDEX, KRUNCH_FLAG)

where INDEX is an index into the volume directory that specifies which volume to act upon and KRUNCH_FLAG is a Boolean.

DELETE_VOLUME will delete the pseudo-volume specified by INDEX, which corresponds to a pseudo-volume (either through a create or VOLUME_INDEX call) only if it contains to

files (if so an error occurs). If KRUNCH_FLAG is set to TRUE, the volume manager will then krunch the Pascal region, else it will not. This procedure follows the name matching convention specified above. Use of this procedure will cause a change in the indexes used to specify pseudo-volumes. If this procedure is used, an application program should update its own copy of the indexes prior to making any calls that use an index.

Errors reported:

a. No such volume - a volume with the INDEX passed was not found.

b. Write-protect error - if the pseudo-volume is write-protected it cannot be deleted

c. Volume has files cannot delete.

ASSIGN_VOLUME

Call format:

ASSIGN_VOLUME(INDEX, UNIT_NUMBER)

where INDEX is an integer and UNIT_NUMBER is an integer in the range 4, 5, 9 - 20, 128 - 143.

This procedure will assign the Pascal unit number (UNIT_NUMBER) to the pseudo-volume specified by INDEX. The unit number must be in the correct range. This action will make the pseudo-volume accessible through the normal Pascal I/O routines. If this unit number is already assigned, the current devive (or volume) will be released from this unit number and the new one will be assigned.

Errors reported:

a. No such volume - a volume with the index passed was not found.

b. Illegal Unit Number - the unit number passed to this procedure was out of range.

c. Cannot assign Profile driver unit number.

RELEASE_VOLUME

Call format:

RELEASE_VOLUME(UNIT_NUMBER)

where UNIT_NUMBER is an integer in the range 4, 5, 9 - 20, 128 - 143.

This procedure will release the pseudo-volume assigned to the Pascal unit number (UNIT_NUMBER). Doing so will make this pseudo-volume inaccessible to Pascal I/O calls.

Errors reported:

a. Not assigned - this unit is currently not assigned.

b. Illegal Unit Number - the unit number passed is not in the legal range.

WP_VOLUME

Call format:

WP_VOLUME(INDEX, WP_FLAG)

where INDEX is an integer and WP_FLAG is a Boolean.

WP_VOLUME will set or unset the write-protect attribute of the volume specified by INDEX. If WP_FLAG is TRUE then it will be write-protected else it will be unwrite-protected.

Errors reported:

a. No such volume - there is no volume specified by this index

KRUNCH_AREA

Call format:

KRUNCH_AREA

This procedure will krunch the Pascal region of the currently active Profile.

SELECT_DRIVE

Call format:

SELECT_DRIVE(DRIVE_NUMBER)

where DRIVE_NUMBER is an integer in the range 0 to 7.

SELECT_DRIVE will select a Profile for the Volume Manager to act upon. The available set of Profile drives is given in the set PASCAL_DRIVES found in the global variables. All volume manager calls are specific to a single Profile. To switch Profiles requires this call.

Errors reported:

a. Drive not active - this drive is not available for use.

b. Illegal drive number - the drive number passed is out of range.

c. No Pascal area on drive.

MODIFY_VOLUME

Call format:

MODIFY_VOLUME(INDEX, NAME, DESCRIPTION)

where NAME is a 7 character string and DESCRIPTION is a 15 character string, INDEX is an integer

This procedure will modify the name and/or the description field of a pseudo-volume specified by INDEX. Either string passed may be null. This will leave the current contents unchanged. Errors that can occur are:

a. No such volume - there is no such volume specified by this index

b. Illegal volume name

c. Write protect error

d. Name conflict

VOLUME_INDEX

Call format:

INDEX := VOLUME_INDEX(NAME, DESCRIPTION)

where NAME is 7 byte string, DESCRIPTION is a 15 byte string, and INDEX is an integer.

VOLUME_INDEX will look up a volume in the volume directory and return its index, which

is then used to perform any volume manager action on that volume. This routine will follow the volume name matching conventions specified above. This call will usually proceed any other volume manager call.
Use of these indexes can be made easier if the calling program maintains a mapping between pseudo-volume names and their indicies once this call has been made. After the deletion of pseudo-volume, however, the application cannot assume that the indexes will remain the same.

Errors reported:

a. No such volume - a volume with this name cannot be found.

GET_VIDR

Call format:

GET_VDIR(VOL_DIRECTORY, NAME_ARRAY, DRIVE_NUMBER)

where VOL_DIRECTORY is of type VDIR_STRUCT (defined in Volume manager interface section) and NAME_ARRAY is of type N_ARRAY (also defined in the interface section. DRIVE_NUMBER is an integer in the range 0 to 7.

This procedure will return the contents of the volume directory on the Profile drive designated by DRIVE_NUMBER. The contents are returned in the user supplied data structure VOL_DIRECTORY which is declared to be of type VDIR_STRUCT. The names of the pseudo-volumes are returned in NAME_ARRAY.

It is important to declare in the application program the following data structures in this order and contiguous:

```
VOL_DIRECTORY: VDIR_STRUCT;
DESCRIPTIONS: DESC_ARRAY;
```

because this call will fill both these data structures.

Errors reported:

a. Illegal drive number - must be in the range 0 to 7

b. Invalid drive - this drive is not available

c. No Pascal area on this drive - this Profile does not contain a Pascal area

GET_STATREC

Call format:

GET_STATREC(STATUS_RECORD)

where STATUS_RECORD is of type STAT_REC (defined in the interface section of the unit.)

This procedure will return the contents of the status record foound in the Profile driver. This contains information about the currently assigned Pascal unit numbers.

Errors reported:

a. No Profile driver - there is no Profile driver attached

INIT_VM

Call format:

INIT_VM

This procedure will initialize the volume manager unit. It sets various global variables, identifies the Profile driver and its unit number, and sets the value for CUR_DRIVE. It DOES NOT initialize the volume directory or status record data structures. The caller must immediately call SELECT_DRIVE with an appropriate drive number to initialize these data structures prior to making any other calls to the volume manager unit. If the volume manager unit is configured such that it is swapped in and out of memory (NOLOAD option) then this procedure must be called whenever the volume manager unit is swapped back in followed by a call to SELECT_DRIVE. This procedure sets up the sets VALID_DRIVE and PASCAL_DRIVES.

Errors reported:

a. No profile driver attached - this is essentially a fatal error since no actions can occur without a profile.

WP_DISPLAY

Call format:

WP_DISPLAY(INDEX, WP)

where INDEX is an integer and WP is a Boolean.

An application may have the volume manager unit display the volume selection screen (shown above) This procedure will update the write-protect field in the display that corresponds to the pseudo-volume specified by INDEX. If WP is true a '*' will be placed in the column or if it is false a ' ' will be placed there.

Errors reported:

a. No such volume - this index value does not correspond to an existing pseudo-volume.

NAME_DISPLAY

Call format:

NAME_DISPLAY(INDEX, NAME)

where INDEX is an integer and NAME is a seven character string.

An application may have the volume manager unit display the volume selection screen (shown above) This procedure will update the name field for the pseudo-volume specified by INDEX with the name passed in NAME.

Errors reported:

a. No such volume - this index value does not correspond to an existing pseudo-volume.

DESC_DISPLAY

Call format:

DESC_DISPLAY(INDEX, DESC)

where INDEX is an integer and DESC is a 15 character string.

An application may have the volume manager unit display the volume selection screen (shown above) This procedure will update the description field for the pseudo-volume specified by INDEX with the string passed in DESC.

Errors reported:

a. No such volume - this index value does not correspond to an existing pseudo-volume.

UNIT_DISPLAY

Call format:

UNIT_DISPLAY(INDEX, UNIT_NUM)

where INDEX and UNIT_NUM are integers.

An application may have the volume manager unit display the volume selection screen (shown above) This procedure will update the unit number display for the pseudo-volume specified by INDEX. If UNIT_NUM is a valid UCSD unit number it will update the display to show the number, else it will set the unit number display to blanks (meaning that this pseudo-volume is not assigned.) When a pseudo-volume is released, the display can be updated by calling this procedure with UNIT_NUM equal to 0.

Errors reported:

a. No such volume - this index value does not correspond to an existing pseudo-volume.

VOL_DISPLAY

Call format:

VOL_DISPLAY(INDEX)

where INDEX is an integer.

An application may have the volume manager unit display the volume selection screen (shown above) This procedure will update all the information (write_protect, name, description, and unit number) for the pseudo-volume specified by INDEX.

Errors reported:

a. No such volume - this index value does not correspond to an existing pseudo-volume.

TITLE_DISPLAY

Call format:

TITLE_DISPLAY

An application may have the volume manager unit display the volume selection screen (shown above)

This procedure displays the column headings for the volume display.

SCREEN_DISPLAY

Call format:

SCREEN_DISPLAY

An application may have the volume manager unit display the volume selection screen (shown above) This procedure will put the complete volume display on the screen for the currently selected Profile. It requires that SELECT_DRIVE has been called. After any create or delete of a pseudo-volume, this procedure should be called to update the complete volume display.

SEL_VOLUME

Call format:

INDEX := SEL_VOLUME

where INDEX is an integer.

An application may have the volume manager unit display the volume selection screen (shown above) If the volume display is used, this routine can be called to have a user select a pseudo-volume from the display as described in the section above. The pseudo-volume selected is specified by the value returned in INDEX. If INDEX is 0 this specifies that the user has aborted the selection process and that NO pseudo-volume has been selected.

REPORT_ERROR

Call format:

REPORT_ERROR

An application may choose to have the volume manager unit report any errors that may have occurred to the screen. If DISPLAY_ERR is true, this procedure will report an error message to the screen. If ERR_FMT is true, the error messages will be displayed on line 3 else they will be displayed at the current position of the cursor. The error displayed will be based on the value of VM_ERROR or VM_IO_ERROR with VM_ERROR having the highest precedence. If both these values are 0 (no error) then no error message will be displayed. After

a volume manager routine has been called, an application program can then call REPORT_ERROR to report any errors that may have occurred.

S_CLEARSCREEN

Call format:

S_CLEARSCREEN

This procedure will clear the screen. It is supplied as a low-level screen management procedure.

S_CLEARLINE

Call format:

S_CLEARLINE

This procedure will clear the current line (i.e. the line in which the cursor currently lies.) It assumes that the cursor is in column 0.

USING THE VOLUME MANAGER UNIT

1. Introduction

This section reviews in detail the way an application writer will use the Volume Manager Unit. Details for the procedure and function calls are given above. The actions that can be performed with this unit are shown below:

a. Creating A Pascal Pseudo-volume
b. Deleting A Pascal Pseudo-volume
c. Assigning A Pascal Pseudo-volume
d. Releasing A Pascal Pseudo-volume
e. Setting the Write-protection of a Pascal Pseudo-volume
f. Krunching the Pascal region of the Profile
g. Modify the name/description field of a pseudo-volume
h. Selecting the Profile Drive to Use
i. Getting the Index for a Pseudo-volume
j. Getting the Pascal area Volume Directory
k. Getting the Profile Driver Status Record
l. Screen management routines
m. Error reporting

Each of these actions is performed on the current drive selected, thus it is important for the user to know which drive they are performing these actions.

All existing pseudo-volumes are referenced via their volume directory index. This value can be obtained either when a program calls the volume manager to create a pseudo-volume or through a function call to the volume

manager given a pseudo- volume's name and description field. Also, a function is supplied that will allow an application program to use the human interface found in the volume manager program.

2. Data Structures

The data structures supplied in the interface section can be divided into 3 areas:

a. Profile information

b. Pseudo-volume directory information

c. Control of display and error reporting

2.1 Profile Information

The volume manager unit maintains a certain data structures that describe the state of the Profile driver. These are:

VALID_DRIVES - the set of all available Profile drive numbers (does not imply that these drives have Pascal areas)

PASCAL_DRIVES - the set of all Profiles with Pascal areas

CUR_DRIVE - the currently selected Profile drive number

STATUS_REC - the Profile driver status record which maps pseudo-volumes to Pascal unit numbers making them available for use

2.2 Pseudo-volume Directory Information

Once a Profile drive has been selected by a call to SELECT_DRIVE, the volume manager unit will maintain directory information for the pseudo-volumes on that Profile. This information is kept in the following data structures:

VDIR - this is the actual volume directory for the Pascal area on this Profile

VDESC - this is the array which holds the description fields for the pseudo-volumes

VNAMES - this is the array which holds the volume names for the pseudo-volumes

The volume manager unit will update both these data structures and their counterparts on the drive itself after any change

is made by a call to the volume manager.

2.3 Control of Display and Error Reporting

Use of the volume manager unit's display routines is based on the setting of some control flags:

DISPLAY_ERR - if TRUE the volume manager unit will report errors to the screen on the line specified by ERR_LINE (normally set to 3)

ERR_LINE - the line on which errors are reported

ERR_FMT - if TRUE report errors on ERR_LINE else report them at the current location of the cursor

When errors occur in the volume manager unit, two variables are set to reflect the error condition:

VM_ERROR - this holds an integer that denotes the error that has occurred

VM_IO_ERROR - if an I/O error occurs then this variable will have the value of IORESULT

3. Creating A Pascal Pseudo-volume

To create a Pascal pseudo-volume requires a call of the form:

```
INDEX := CREATE_VOLUME(NAME, DESC, SIZE)
```

This will create a pseudo-volume on the currently selected Profile with the name NAME, its description field will be set to the string passed in DESC, and it will be SIZE blocks in length. The index returned should be stored in the calling program, for it must be used for all other calls that will assign, delete, etc. this pseudo-volume. The index can also be obtained by a call to VOLUME_INDEX using the same name and description field. This call can return 3 possible errors, either to the calling program or by reporting them to the screen (if so desired.)

4. Deleting A Pascal Pseudo-volume

This is not a recommended practice for application programs to do. The end-user should only delete pseudo- volumes via the volume manager program (from the PPM). If an application needs to delete a pseudo-volume, it is done through the call

```
DELETE_VOLUME(INDEX, KRUNCH_FLAG)
```

The index corresponds to a pseudo-volume that is obtained either through a CREATE_VOLUME or VOLUME_INDEX call. After a pseudo-volume has been deleted, the Pascal region can be krunched if the KRUNCH_FLAG is set to TRUE. This call will return an error if there is no volume that corresponds to that index

or if the volume is write-protected.

5. Assigning A Pascal Pseudo-volume

For a program to use a pseudo-volume as a Pascal volume, the pseudo-volume must be assigned to a Pascal unit number. To do so requires a call of the form

ASSIGN_VOLUME(INDEX, UNIT_NUMBER)

The index value specifies the pseudo-volume to assign with the Pascal unit number passed via UNIT_NUMBER. An error will occur if there is no corresponding pseudo-volume or if the UNIT_NUMBER value is not in the correct range of Pascal unit numbers (4, 5, 9 - 20, 128 - 143).

6. Releasing A Pascal Pseudo-volume

To release a pseudo-volume from its assigned Pascal unit number, requires a call of the form:

RELEASE_VOLUME(UNIT_NUMBER)

where UNIT_NUMBER corresponds to the Pascal unit number that has been assigned. It is recommended that any application that assigns unit numbers will also release them before completion of execution. This will free the user from having to hand-release these pseudo-volumes before executing another program. This call can return two errors, one of which if the unit is not currently assigned or if the unit number is not in the correct range.

7. Setting the Write-protection of a Pascal Pseudo-volume

To set or clear the write-protect attribute of pseudo- volume, make the call

WP_VOLUME(INDEX, WP_FLAG)

INDEX selects the volume and if WP_FLAG is true it will be write-protected, else the write-protect attribute will be cleared. An error will occur if the is no volume that corresponds to the index passed.

8. Krunching the Pascal Region of the Profile

This is not a recommended practice for application programs. The only time it may be necessary is if when a create of volume is attempted and there is not enough room for the volume a call to KRUNCH_AREA may free up enough space for the volume. The call is simply

KRUNCH_AREA

9. Modify the name/description field of a Pseudo-volume

An application can change the name and/or the description field of a pseudo-volume. This is not a recommended practice. Calling MODIFY_VOLUME(INDEX, NAME, DESCRIPTION) will change the specified values.

Either the NAME or DESCRIPTION parameter may be null, to not change the field.

10. Selecting the Profile Drive to Use

All volume manager actions are performed on the currently selected Profile drive. Each Profile drive is assigned a drive number (in the range 0 to 7). The default Profile is drive 0. To select a Profile, the application program should check the set PASCAL_DRIVES in the volume manager interface to determine which drives are active. PASCAL_DRIVES is set up when the volume manager is initialized. Any currently active Profile drives will be placed in it. If a user turns off a Profile after PASCAL_DRIVES is set, then any action to that Profile will result in an I/O error. For example,

```
IF 0 IN PASCAL_DRIVES THEN SELECT_DRIVE(0)
```

SELECT_DRIVE will return an error if the drive is not active, i.e. if it is not in PASCAL_DRIVES or if an illegal drive number (out of range) is passed.

11. Getting the Index of a Pseudo-volume

In order to act upon a pseudo-volume, you require the index that corresponds to that pseudo-volume. To get the index requires a call

```
INDEX := VOLUME_INDEX(NAME, DESCRIPTION)
```

This function will return the index that corresponds to the pseudo-volume whose name and description field match the values passed. If an error occurs it will return a value of 0 to INDEX. The rules for matching are:

a. if there is only one pseudo-volume with the name NAME then return its index

b. if there are more than one pseudo-volume with the same name, then match description fields. If there is no match then return an error. If there is a clear match then return the index.

c. if no name is matched then return an error.

12. Getting the Pascal Area Volume Directory

Normally, an application program will not have to know about the contents of the Pascal area volume directory. In such cases as it may, this procedure is supplied to allow a program to inspect the contents (but it may not change them.) The program needs to declare the following data structures in its global data section in the following order and format:

```
VAR

    VOLUME_DIRECTORY: VDIR_STRUCT;
    DESCRIPTIONS: DESC_ARRAY;
    NAME_ARRAY: N_ARRAY;
```


Calling GET_VDIR will transfer the information into these data structures. Care must be made that the programmmer does not put any other data structures amidst these for they will be wiped out! Use of this procedure will not set CUR_DRIVE to this drive_number.

13. Getting the Profile Driver Status Record

Using GET_STATREC is also not intended for the usual use of the volume manager unit. Again, this only supplies information and the user cannot change the contents. The program must declare the data structure STATUS_RECORD shown below in its global data area:

```
VAR

    STATUS_RECORD: STAT_REC;
```

14. Error Handling

After any call to the volume manager unit, there is a possibility that an error occurred. This is registered in the VMERROR variable found in the volume manager interface. After any call, this variable should be checked to see if an error has occurred. Any I/O errors are noted in the variable VM_IO_ERROR. It should also be checked. The error values are shown below for VMERROR:

0 - No error

1 - No such pseudo-volume

2 - Not enough room to allocate pseudo-volume

3 - Volume directory full

4 - Name conflict

5 - Illegal unit number

6 - Pseudo-volume not assigned

7 - Profile Drive not active

8 - Illegal drive number

9 - Illegal volume name

10 - Write Protect error

11 - No Pascal Area on this Profile

12 - No Profile driver attached

13 - Volume size must be greater than 6 blocks

14 - ProDOS directory is full

15 - Pseudo-volume contains files cannot delete

16 - Cannot assign unit number used for Profile driver

17 - The ProDOS directory has a ProDOS file called PASCAL.AREA

VM_IO_ERROR will contain the standard IORESULT value for any I/O errors that may have occurred. Use of these two variables parallels the use of IORESULT in Pascal programs. After a call has been made to the volume manager, the application should check VM_ERROR and VM_IO_ERROR, to determine the success of the call.

The application program has the choice whether or not it wishes to report any errors that may occur while using the volume manager unit. Also, it can allow the volume manager unit to report the errors. Two variables found in the interface control error reporting. They are:

DISPLAY_ERROR - if TRUE then the volume manager will report errors to the console, else no error messages will be displayed

ERR_FMT - if TRUE and if DISPLAY_ERROR is TRUE then all error messages will be displayed on line ERR_LINE which is set to 3, by default, of the console, else if ERR_FMT is FALSE and DISPLAY_ERROR is TRUE then error messages will be displayed on the current line of the console, i.e. at the current cursor position

ERR_LINE - this variable specifies on which line to report errors. It is set to line 3 by default. An application program can change this value to suit its needs. It is only used if ERR_FMT is set TRUE.

The volume manager supplies an error reporting procedure REPORT_ERROR, that will print an error message based on the current values of VMERROR or VM_IO_ERROR. An application program can call this procedure to report any errors. This procedure will report errors given the settings of the above flags.

15. Managing the Screen Display

For the most part, the application program is expected to manage its own screen display a propos to its purposes. The volume manager unit supplies the routines necessary to use the volume display shown in the section above. After an an application has performed a SELECT_DRIVE it can display the available pseudo-volumes on that drive by calling SCREEN_DISPLAY. Various fields within that display can be updated after any volume manager unit call following the protocols given below:

After the creation of a pseudo-volume:

call SCREEN_DISPLAY

After the deletion of a pseudo-volume:

call SCREEN_DISPLAY

After assigning a pseudo-volume:

call SCREEN_DISPLAY

After releasing a pseudo-volume:

call UNIT_DISPLAY with the index of the pseudo-volume that has been released with a unit number of 0

After clearing or setting of write_protection:

call WP_DISPLAY with the index of the pseudo-volume and a boolean where TRUE means write-protection has been set and FALSE means write-protection has been cleared

After krunching:

no update to the screen is necessary

After modifying the name or description field:

call either/both NAME_DISPLAY and/or DESC_DISPLAY with the index of the pseudo-volume and the new value for that field

After selecting a Profile drive:

call SCREEN_DISPLAY

To have the user select a pseudo-volume:

once SCREEN_DISPLAY has been called, call SEL_VOLUME to have the user select a pseudo-volume, this call will return its index

An application can use the volume manager unit's error reporting mechanism if it so chooses. If it chooses to do it itself, the variables VM_ERROR and VM_IO_ERROR are available to the application program to use to determine what if any error has occured and to report it in its own manner.

16. Swapping the Volume Manager Unit In and Out of Memory

When an application program that uses the volume manager unit is loaded,

the initialization code for the unit is executed. This code will set up VALID_DRIVES and some internal variables used by the volume manager unit. To conserve space in an application, this unit can be NOLOADed so that it is resident only when required. If this is done a call to INIT_VM must be made prior to using any other functions in the volume manager unit. This call will set up these variables again.

APPLE//e TECHNOTE #1

Revision of notes on the Apple//e Dec 82*
5-July 84

This technote explains the difference between the Apple//e and Apple][+. It also provides a quick reference for the softswitches and makes some programming suggestions.

For further information contact:
PCS Developer Technical Support
M/S 22w. Phone (408) 996-1010

TECHNICAL OVERVIEW OF THE APPLE IIe

This document is designed for software developers who have some familiarity with the Apple II. Its function is to provide a quick overview of technical information that may affect software design and to a lesser extent hardware design. It is by no means a complete description of the Apple IIe as the manuals provided with the product serve this purpose. An effort has been made to extract from the manuals information that is not obvious. An effort has also been made to point out potential problems resulting from the new design and, where appropriate, to give suggestions on how to avoid the problems.

GENERAL:

1. Full ASCII keyboard with auto-repeat feature, alpha lock and Apple keys.

2. Custom-ICs are used for memory management and I/O control thereby reducing chip count.

3. The language card and slot 0 have been replaced by built in (look alike) RAM called bank-switched memory.

4. An additional slot has been added. It is called the auxiliary slot. This slot has several functions:
   a. It is used for testing and diagnostics.
   b. It serves as the slot for the 80 column card (logically it is mapped as being in slot 3 - $C300).
   c. It serves as the slot for the 80 column / 64K ram card.

5. The back panel is designed for direct mounting of DB-9, DB-19 and DB-25 connectors. This feature allows peripherals to be attached to the back of the Apple IIe rather than to the peripheral cards.

6. It looks like an Apple II.

7. In addition to an introductory booklet,the APPLE IIe OWNER'S manual, there is an APPLE IIe 80-COLUMN TEXT CARD manual, an EXTENDED 80-COLUMN TEXT CARD SUPPLEMENT, and an APPLE IIe REFERENCE manual. Also new revisions of the APPLESOFT TUTORIAL, DOS and APPLESOFT REFERENCE manuals have been written. Additional documentation such as A GUIDE TO THE NEW FEATURES OF THE APPLE IIe COMPUTER and Technical Support handouts have been developed.

8. Applesoft has not been changed at all. Integer BASIC can still be used; it must be loaded into the bank-switched memory (language card).

9. The Autostart ROM has been replaced by a new ROM capable of supporting 80 columns. The autostart entry points have been maintained.

10. The Apple IIe with its auxiliary card functions like an Apple II plus with a language card, upper and lowercase capability and an 80 column card. This means that software and hardware which would operate

improperly in an Apple II plus equipped with these features will probably operate improperly in an Apple IIe. NOTE: the Apple IIe will shift to upper case only if the Alpha Lock key is pressed down.

KEYBOARD:

1. All ASCII code-generating keys will start repeating if held down for more than half a second.
   a. If another key is momentarily pressed while a key is repeating, the new key will begin to repeat.
   b. Since the escape key repeats, care must be taken when using it. In a series of escapes every other escape cancels the effect of the previous escape.

2. ASCII codes
   a. The delete key issues the ASCII code 127 (DEL). In immediate mode a checkerboard is displayed to signify that this key has been pressed.
   b. Up arrow key code is 11, down arrow key code is 10, tab key code is 9.

   c. Pascal 1.1 was designed to funcion with the Apple II keyboard. It therefore has some keyboard related differences when run on the Apple IIe. For example, if the up arrow key is pressed while running Pascal 1.1 without the 80 column firmware active a [will be displayed. Another such difference results in the user needing to press shift-2 in order to generate the @ sign. On the Apple II the @ key was obtained by pressing shift-P.

   NOTE: BASIC still adds 128 to its ASCII codes to signify a keypress.

3. Open & Closed Apple keys
   a. They do not modify or generate ASCII codes.
   b. They cannot be detected by looking at $C000 the normal keypress method.
   c. They are connected directly to the game push buttons. Their key press can be detected by looking for a game push button being pressed. Their presence means there are always game push buttons. This will cause problems for games that determine that there are game paddles by the presence of push buttons. If a joy stick is connected to the Apple IIe and it has a way of locking down one of the buttons or if it is of the Atari type, which has reverse polarity, then when the computer is turned on or control-reset is pressed the computer detects what appears to be an Apple key being pressed and so goes through diagnostics each time reset is pressed.
   d. In combination with control-reset
      1. Open Apple does a cold start after scrambling several bytes of RAM. This combination of key presses replaces the 'give up and start all over again ' key (power switch). It can also substitute for PR#6 with the added advantage that if 80 column firmware is active it will be properly disconnected. PR#6 disconnects the 80 column firmware but leaves the 80 column hardware enabled resulting in improper functioning of the text screen.
      2. Closed Apple sends the computer through an onboard diagnostics test used during production testing. This is only a partial

diagnostic.

3. Pressing both Apple keys results in the diagnostic test being run with output to the speaker.

NOTE: After the diagnostics test has successfully been completed the rather non-informative message "kernel OK" is given. This message means that the diagnostics are completed and that no problems were found. Press reset to reboot the system.

4. The Caps lock key must be down to create the upper case letters needed for BASIC and DOS commands. BASIC permits lower case letters within quotes. When 80 column firmware is active a restrict mode may be selected that will automatically shift anything outside double quotation marks to upper case. Restrict mode is entered by typing PR#3 then escape followed by R.

5. The ARROW keys in combination with ESCAPE function in the same way as I,J, K & M. NOTE: If you want to use an ARROW key to copy something from the screen after 'escaping' to that line press some other key to deactivate the escape mode prior to using the arrow key otherwise you will just continue escaping and will not do any copying.

6. The shift key mod may be duplicated by soldering a 500 ohm resister across the logic board connector at X6..
NOTE: the warranty is voided by doing this.

7. Integer BASIC was not designed to recognize the full set of ASCII characters so some of the keys on the Apple IIe that are not on the Apple II plus may do strange things. Integer BASIC will recognize normal lower case characters but treat them as upper case. Integer and Applesoft BASIC running in bank-switched memory can be made to recognize lower case as explained below.

8. Pressing reset on an Apple IIe does a full 64K reset where the Apple II resets only the lower 48K. This is most noticable by Pascal users who have only one disk drive since they must press reset part of the way through boot-up which then starts the booting process over again. In general a program which runs in the bank-switched or auxiliary memory should set up reset jump vectors which bank in the appropriate memory before jumping back into the program. These jump vectors need to be in the lower 48K of main memory.

9. Pressing reset sets the monitor routines to display video in NORMAL mode. A reset does not inform Applesoft that it should be displaying in NORMAL mode and so it continues to fiddle with the output characters' ASCII code. On the Apple II plus this situation was not detectable. On an Apple IIe the effects of this can be detected. From immediate mode type FLASH then press reset. If you then try to print something to the screen or make a listing some characters are not displayed correctly - (numbers become lowercase letters). Giving any of the following commands corrects the situation: NORMAL, INVERSE, FLASH.

VIDEO OUTPUT:

1. When you boot the Apple IIe via PR#n, DOS 3.3 does not initialize the 80 column firmware nor turn the card on or off. If the 80 column firmware was active when the disk was booted then after booting, the 80 column hardware is still on but the firmware is not. This makes for garbage on the screen. Using control-Open Apple-Reset rather than PR#6 prevents this. Programs that use 80 columns should turn 80 columns off (PRINT CHR$(21)) at the end of the program. To ensure that your program will not be clobbered by this half in and half out situation your program needs to completely turn all of the 80 column card on or off. This is how. First determine the configuration of the computer by using the identification routines. If the computer is an Apple IIe with an 80 column card then do a PR#3 to turn the card on and then if you don't want 80 columns issue the command PRINT CHR$(21) to turn it off.

   NOTE: other 80 column cards on the Apple II or Apple IIe using DOS will have similar problems.

2. To turn the card on from BASIC the command PR#3 must be typed in or the program must issue PRINT CHR$(4);"PR#3". Pascal programs will automatically turn the card on. Runtime Pascal programs may be designed to prevent this from happening. From the monitor type C300G to turn the card on. Assembly language requires a JMP to $C300. Issuing these commands when there is no card in slot 3 or the auxiliary slot will do undetermined things. The Apple IIe will usually reboot the disk drive rather than hang like in the Apple II plus but anything may happen. Reminder issuing a PR#3 to turn on the 80 column card is like issuing a PR#1 to turn on a printer. These commands only set software pointers and the periperal is not actually initialized until the first character is sent to the peripheral. For this reason any screen setting such as VTAB issued after PR#3 but before a PRINT command will be ineffective since it will be changed when the peripheral is initialized.

3. Presence of the 80-column card in the video expansion slot can be determined by writing to a screen location on the card and then checking to see that the value found at that location is the value written. i.e. RAM exists at that location.

4. The Apple IIe can be identified by a six at $FBB3 (64435). Licensed developers can obtain full identification routines from Apple's PCS Technical Support Group.

5. 80 column features are controlled from programs by printing control characters. For example, PRINT CHR$(21) deactivates the 80 column card if one exists. It does nothing if there is no card.

6. 80 column features are controlled from immediate mode by using escape sequences. For example, typing the escape key (don't hold it down) followed by control-Q deactivates the 80 column card.

7. If 80 column firmware is active it can display either 40 or 80 column text.

8. The cursor may be used to identify the status of the computer.

a. BLINKING CURSOR means autostart ROM is active. This will happen if an image of autostart ROM has been placed in the bank-switched memory (language card) and control has been turned over to it.
b. FLASHING CHECKER BOARD CURSOR means the new monitor firmware is active and that the 80 column features are inactive.
c. SOLID INVERSE SPACE CURSOR (40 or 80 column width) means the 80 column features and the new monitor ROM are both active.
d. INVERSE PLUS SIGN CURSOR (40 or 80 column width) means the 80 column features and the new monitor ROM are both active and the escape key has been pressed.

9. The 'other half' of the 80 column screen is located in 1K of RAM on the 80 column or extended 80 column card. The address range of this RAM is from $400 to $7FF (text page one). Data being displayed from this 'other half' is shuffled in with data from text page one on the main board and shrunk by the I/O control chip to produce the 80 column screen. Data from the main board is displayed as the odd columns while data from the 'other half' -the 80 column card- is displayed as the even columns. Display columns are numbered starting from one.

10. A vertical blanking signal is available at $C019 to help graphics animators. Vertical blanking takes approximately 4 milliseconds. Updating of screen characters or switching of pages during vertical blanking prevents the displaying of graphics while the graphics is being updated.

11. The auxiliary slot for the 80 column card has the same memory mapping as slot 3. Therefore, the 80 column card is treated like a peripheral.

12. If the 80 column firmware is inactive and reset is pressed while running in autostart ROM on the language card (either Integer or Applesoft BASIC), the computer returns control to that language (i.e., the language card) without change--DOS does it.

13. Reset deactivates the 80 column card firmware and hardware.

14. Using commas to tab while in Applesoft does not work with 80 columns. You can tab with 80 columns by poking 36,n where n is the column you want to tab to. Poking 36,n will also work for 40 column display as long as n is < 40. Poking 36 with a number >39 probably will cause the program to crash if displaying in 40 columns.

15. If Integer or Applesoft BASIC is running in the bank-switched memory (language card) with the autostart ROM then they will not accept lower case characters. If you want to be able to accept lower case characters then type PR#3 to activate the 80 column firmware. This will replace the autostart ROM by the new ROM. The card may then be deactivated with an escape control-Q and the new monitor ROM will continue to accept lower case.

16. The 80 column display running under the new ROM is markedly slower then 40 columns under autostart ROM.

17. Scrolling windows can be set up anywhere on the 80 column screen. The

width of a scrolling window is limited to an even number. If an attempt is made to set up a odd-numbered window width the width is reduced by one. Therefore, if an attempt is made to set a window width equal 31 by placing the number 31 in the location $33 the actual window width will be 30.

18. There are two built in character sets.
    a. The standard character set displays uppercase characters in NORMAL, INVERSE, and FLASH as is the case with the Apple II plus. It will also display lower case NORMAL characters. NOTE: do not use lowercase normal characters in programs you want to run on an Apple II plus since it cannot display lower case characters.
    b. The alternate set provides for upper and lower case in both normal and inverse.
    c. Atempting to display lower case inverse characters without having the 80 column firmware switched in will not work. Even if the alternate character set is banked in these characters will be displayed as special characters.

    d. When the 80 column firmware is activated the alternate character set is used. This means that software designed to be used with 80 columns must be designed for the alternate character set. Attempting to switch to the standard character set while 80 column firmware is active will result in some of the characters being misinterpreted by Applesoft.

    e. Since both character sets are designed to display the underline character and the descenders of lower case letters, all the characters have been moved up one row of dots. This may cause some visually unpleasant lettering on the top row of a text display which uses inverse video. This can be corrected by placing one row of inverse blanks above the first line of print.

19. Unlike the Apple II plus the Apple IIe's GETLINE routine is affected by the INVERSE FLAG (location $32) setting. On the Apple II plus all BASIC input or Assembly language input using GETLINE is displayed in normal mode. On the Apple IIe input will be displayed in accordance with the value in location $32-(inverse, normal, flash). This is most noticable while in immediate mode. Typing the BASIC command INVERSE results in future keypresses being displayed in inverse. HOME and clear-to-the-end-of-line gives inverse blanks.

20. When displaying in 80 columns, if you look at CH (36) you will find it = 0 even if the cursor is not at the left edge of the screen. This is done to fool BASIC which knows only about 40 columns and to provide windows. Some other 80 column boards set this location to 40. Placing a value into 1403 ($57B) performs an HTAB to that value.

21. Some of the I/O Scratchpad RAM Addresses located in the text screen buffer are used by the 80 column firmware. These are used in accord with the protocol for their use but some programs may have used these areas incorrectly and will have problems. The most common abuse of these protocols is when a lo-res picture is BLOADED into $400-$7FF. When this is done the values in the scratchpad area are changed to match what they

were when the picture was saved. In the past the most common result is that the disk drive 'grinds' since the read head gets lost. A similar 'loss of control' will happen to any peripheral including the 80 column card. Since slot 3 in the Apple IIe is dedicated to the 80 column firmware it uses one scratchpad area dedicated to slot 3 even if there is no 80 column card. Therefore, any BLOADing into the area $400 to $7FF will affect the operation of the output routines, possibly crashing the program. The solution is to BLOAD the picture into a buffer and then move all but the scratchpad area into the screen buffer. A simpler solution, but one that may crash the program if an interrupt occurs during loading, is to save the scratchpad data then restore it after loading the picture. From machine language you could disable interrupts during this operation.

VIDEO SOFT SWITCHES:

1. ALT. CHAR SET - This switch sets up the alternate character set. This switch should be used with care by BASIC programmers as previously pointed out.

2. 80 STORE - If 80 store is active then the PAGE2/NOT PAGE2 switch serves as a bank switching switch rather then video display switch. This is true of the hi res pages also if the extended 80-column card is present. This switch should be used only by experienced programmers.

3. 80 COLUMNS - This switch is designed to assist assembly language programmers who are using their own screen writing routines. This switch turns only the disply hardware on and not the firmware. Programs which use the monitor I/O routines COUT & RDKEY cannot use this switch alone but must use it in combination with PR#3. This is true for both BASIC and the Monitor.

4. TEXT/GRAPHICS, MIXED/NOT MIXED, PAGE2/NOT PAGE2 and HIRES/NOT HIRES serve the same function as in the Apple II plus. The PAGE2/NOT PAGE2 switch serves the additional function of bank switcher as mentioned above. The state of these switches may be found by reading status bytes.

5. VERTICAL BLANKING can be read to determine correct display timing for animated graphics.

6. Two soft switches affect the input/output memory space ($C100 to $C7FF).
   a. The SLOTC3ROM switch is used to select between the space alocated to slot 3 and built in ROM alocated to controling Apple's 80-column cards.
      1. When the computer is reset or turned on it checks for a card in the auxiliary slot. If it finds one the SLOTC3ROM switch is turned off. This banks in the built in C3xx ROM. NOTE: this does not turn on the 80-column card. It simply provides the card with the ROM it will need if a PR#3 or equivalent command is given.
      2. This switch may be turned off -built in ROM banked in- and the 80-column ROM used even if there is no 80-column card. To get into this mode turn off the switch (POKE 49162,0) and give the

PRINT chr$(4);"PR#3" command. Without a card, 80 columns cannot be displayed but features such as upercase restrict are available.

b. The SLOTCXROM switch is used to select between the space alocated to slots 1 through 7 and built in ROM alocated to controling Apple's 80-column card and the built in diagnostics.
   1. If the SLOTCXROM switch is on then the 80 column firmware is mapped in even if the SLOTC#ROM switch is off.
   2. The SLOTCXROM is turned on when one or both of the Apple keys is pressed during reset.

7. The state of the soft switches may be found by reading the appropriate memory locations. With the exception of the SLOTCXROM switch if the value read is >127 then the switch is on.

8. When Pascal 1.1 is initialized it turns on the following soft switches: HIRES, TEXT, NOT-MIXED & NOT-PAGE2. If the 80-column firmware is turned on then 80 STORE is turned on. Since it is not intuitive that the HIRES switch is on even when the program does not use hires and that 80 STORE is on even when not storing things in memory some unexpected things may happen. The unexpected events have most impact on programs directly writing to the text screen and programs using the auxiliary memory. Note that if your program turns off the 80 STORE switch it must turn it back on before it tries to use the 80-column firmware to display 80-column text.

MEMORY MAPPING & ADDRESSING:

1. The memory mapping of the Apple IIe matches the Apple II plus with a language card. Soft switches and the new monitor firmware may be used to bank in additional ROM and RAM.

2. Like the 6502 in the Apple II plus, the 6502A used in the Apple IIe activates the address bus twice during successive clock cycles during an indexed store operation. This may cause a device that toggles each time it is addressed to end up back where it started. In these cases read operations should be used rather than stores.

3. The $D000 to $FFFF memory space functions in a method identical to the language card on the Apple II plus but since it is built in it is referred to as bank-switched memory.

4. Pressing reset switches out bank-switched memory. If operating under DOS 3.3, DOS will switch back to bank-switched memory.

5. While in 80 columns the 1K of auxiliary RAM being used is from $400 to $7FF. The 80 column text card with 1K of RAM uses sparse memory mapping. this means that writing to the location $C00 or $800 on the card is the same as writing to the location $400.

AUXILIARY RAM:

1. 1K of additional RAM exists on the standard 80-column card in address

space $400-$7FF. 64K of additional RAM exists on the extended 80-column card in address space $0000 - $FFFF. This additional RAM is banked in by addressing (writing to) soft switches. To determine if main memory or auxiliary memory is banked in look at the appropriate status bytes.

2. The following softswitch pairs switch between main RAM and auxiliary RAM in the specified ways:
   a. RAMRD - The setting of this switch affects which bank of memory is being read if the read operation is between memory locations $200 and $BFFF.
   b. RAMWRT - The setting of this switch affects which bank of memory is being written to if the write operation is between $200 and $BFFF.
   c. ALTZP - The setting of this switch pair affects which bank of memory is being written to and read from if the read or write operation is to a memory location between $0 and $1FF or between $D000 and $FFFF.
   d. 80STORE - This switch pair in combination with the PAGE2, HIRES and TEXT switch pairs determine in a complex way what display memory is being written to and read from. In general it changes the other switch pairs' functions from screen switching to bank selection. The memory being affected is the same as would be affected by the screen switching.

3. Switching auxiliary memory does not affect the bank-switched memory (language card)$D000 - $FFFF settings. If main board ROM is banked in and then auxiliary memory is switched in, the main board ROM is still active. If bank-switched memory is active and aux mem is switched in then the bank-switched memory in the auxiliary memory will be active.

4. The auxiliary memory provides storage and program expansion capabilities for BASIC, PASCAL and Machine language programs. Machine language programs can very effectivly use the extra memory since the program itself can run in the extra memory if need be. BASIC can use the extra memory to store machine language routines and pictures. Pascal can use the extra memory to store machine language procedures. Both BASIC and Pascal programs are limited to using the standard memory areas since they are unaware of the extra memory. With care BASIC programs could be CHAINED using the extra memory rather than a disk. Also, several programs could be in the computer at once with the posibility of one being in BASIC and the other in Pascal.

5. Programs using DOS would need to do all their input and output from either the main memory or the auxiliary memory but not from both unless a copy of DOS were placed in both banks and then both were kept informed of such events as switching the output device. If programs in auxiliary memory need to produce input or output which is not a DOS command they may use the routines COUT1 and KEYIN which do not go through DOS. Great care should be used since 80STORE may need to be used to affect where the output goes.

6. If you write data into the $400 to $7FF space in auxiliary memory and the computer is displaying 80 columns then your data will appear on the screen.

7. The routine AUXMOVE moves a block of data from anywhere in the memory area $200 to $BFFF. Data may be moved from auxiliary memory to main memory or from main memory to auxiliary memory.

8. The routine XFER transfers program control from a machine language program in main memory to one in auxiliary memory or the other way around.

INPUT /OUTPUT:

1. Sending control to a slot which does not have any device connected to it constitutes a NO-NO. In the Apple II plus this NO-NO usually resulted in the computer stopping dead. In the Apple IIe this NO-NO usually results in the disk booting.

2. The SLOTC3ROM soft switch pair selects between internal ROM at $C300 (the 80 column firmware) and slot three.

3. The SLOTCXROM soft switch pair selects between internal ROM from $C100 to $C7FF used by the built in diagnostics and 80 column firmware, and slots one through seven.

4. Very large peripheral cards which stick out the back of the computer will not be able to do so because of the new back panel.

5. Cards which depend on 'piggy backing' to IC sockets to obtain additional signals will probably no longer function properly since the main board is re-designed.

6. The Apple IIe's 80 column card is a peripheral. As is the case with the Apple II plus, two peripherals cannot receive input or send output at the same time. This means that the 80 column firmware must be made inactive before using an output device such as a printer or MODEM and will need to be reactivated when returning to 80 columns. Some software such as the Pascal BIOS does this automatically.

7. There are two locations on the Apple IIe designed for plugging in game paddles. One is a DB-9 connector on the back panel and the other is the same as on the Apple II plus. A game paddle or joy stick may be connected to one or the other location but not to both at the same time.

PASCAL 1.1

As explained earlier the Pascal system and the 80 column firmware when running under Pascal set several soft switches which may create unexpected situations because their settings are not intuitive. Namely, Pascal turns on the HIRES switch during initialization and the 80 column firmware turns on the 80STORE soft switch. As a result of these settings the following unexpected situations may occur.

1. If an Assembly language Pascal procedure is designed to store and retrieve data and it tries to do so from locations $2000 to $4000 in the auxiliary memory it will not do so properly. This is because the HIRES switch is on along with 80STORE and it overrides the RAMRD and

RAMWRT switches. If HIRES is switched off then this area may be used.

2. If a program turns on the PAGE2 soft switch the program may self destruct. If HIRES and 80STORE are still on when PAGE2 is turned on, any data being sent to or retrieved from the $2000 to $4000 memory space will be sending and retrieving from the auxiliary memory space rather than the expected main RAM. If the 80 column card does not have auxiliary memory then the data is coming from and going to thin air. Since the Pascal heap in larger programs will grow into this space the program self destructs. If you must have the PAGE2 switch on then turn off one of the other two switches. NOTE: Pascal does not use the PAGE2 switch to display text or graphics but through trickery it can be turned on.

BACK PANEL:

1. The back panel is designed to support the mounting of DB9, DB19 and DB25 connectors. Cables from peripheral cards run to these connectors. External cables then run from these connectors to the peripheral.

2. The four DB19 mounting holes are reserved for disk drive connections.

3. The Apple IIe's accessory kit contains materials for attaching cables designed for the Apple II plus.

4. Peripherals which use more than 25 lines will need to use two or more of the DB connectors to route their wires through the back panel.

HARDWARE:

1. The Apple IIe uses the 6502A but it still runs at one MHz.

2. There is a 470 ohm resistor on both the open apple and closed apple key. These resistors are on the keyboard.

3. The Apple IIe's data bus is now buffered and may cause timming differences in connection with using the DMA line.

4. The shift-key mod used in the Apple II+ to simulate upper case can be simulated in the Apple//e by soldering the solder blob found on the main board at location X-6.

INTERRUPTS:

When an interrupt occurs the Apple IIe saves the status of the text page (page1 or 2) and SLOTCXROM switches, then sets the page to page 1 and SLOTCXROM to Slot ROM. After the interrupt has been handled these two switch settings are restored.

DOCUMENTATION ERRATA:

1. To connect the game input switches (push buttons) to other hardware use aprox. 500 ohm pull-down resistor connected to ground and a momentary-contact switch to +5V.

2. The MOVE routine in the Apple IIe is the same as in the Apple II plus and therefor the 'Y' register should be set to 0 before calling it.

3. The SLOTCXROM switches are reversed. The slot ROM is selected by writing to 49158 ($C006). The internal ROM is selected by writing to 49159 ($C007).

ADDENDUM TO TECHNICAL OVERVIEW OF THE APPLE//e

1. An unusual condition appears on the text screen using an Apple//e when a text display is switched from inverse to normal or normal to inverse. This only takes place if the change is being made while printing to the bottom line of a scrolling window. If going from normal to inverse the text appears in inverse but the right end of the line is black which is just like on the Apple][+. If going from inverse to normal the text appears in normal but the right end of the line is white. This condition happens because when the screen is scrolled after the printing of the last line, the new bottom line is filled with blanks in the current mode (inverse or normal). This cleans off the old text on that line in preperation for printing text on the line. The screen display is then switched to the new mode and the last line is printed. This condition can be corrected if you must change text modes on a scrolling window. To do this end the last print statement with a semicolon to suppress the scrolling. Follow this by the change of mode and a print statement without any text.

2. If the HOME command is given on an Apple//e while the text mode is in inverse the whole screen becomes white. On the Apple][+ the screen would clear to black..

3. The following is a list of all the special use locations in memory locations $C000 through $C0FF. Note that in some cases different switches are activated depending upon if they are read from (PEEK, LDA) or written to (POKE, STA). If reading a value can indicate the state of a soft switch, the state having the symbol (*) is the state which will return a value greater than 128 ($7F).

LOCATION	EFFECT OF READING	EFFECT OF WRITTING
49152 ($C000)	get keyboard input	pg1&2 sw show diff txt & gr buff
49153 ($C001)		pg1&2 sw bank swich txt & gr buff
49154 ($C002)		read from main memory
49155 ($C003)		read from auxiliary memory
49156 ($C004)		write to main memory
49157 ($C005)		write to auxiliary memory
49158 ($C006)		select card ROM all slots
49159 ($C007)		select internal ROM $C100-$CFFF
49160 ($C008)		read & write main stack,z-pg.,LC
49161 ($C009)		read & write alt. stack,z-pg.,LC
49162 ($C00A)		select internal ROM $C300-$C3FF
49163 ($C00B)		select card ROM slot three
49164 ($C00C)		turn 80-column display off
49165 ($C00D)		turn 80-column display on
49166 ($C00E)		select Apple][char. set
49167 ($C00F)		select new full upper & lower char. set
49168 ($C010)	clear the keyboard strobe	clear the keyboard strobe
49169 ($C011)	indicates if LC first 4K bank one or (*)bank two is in	
49170 ($C012)	indicates if Autostart ROM or (*)LC is banked in	
49171 ($C013)	indicates if main or (*)aux RAM is being read from ($200-$BFFF)	
49172 ($C014)	indicates if main or (*)aux RAM is being written to ($200-$BFFF)	
49173 ($C015)	indicates if card or (*)internal ROM being read ($C100-$CFFF)	
49174 ($C016)	indicates if main stack,z-pg.,LC or (*)aux stack,z-pg.,LC	
49175 ($C017)	indicates if internal or (*)card ROM being read ($C300-$C3FF)	
49176 ($C018)	indicates if storing to main or (*)aux text & graphics buffers	
49177 ($C019)	indicates if vertical blanking or (*)not vertical blanking	
49178 ($C01A)	indicates if displaying graphics or (*)text	
49179 ($C01B)	indicates if displaying full page graphics or (*)mixed txt & gr	
49180 ($C01C)	indicates if displaying page 1 or (*)page 2	
49181 ($C01D)	indicates if displaying in lo-res or (*)hi-res	
49182 ($C01E)	indicates if using Apple][char set or (*)alternate char set	
49183 ($C01F)	indicates if displaying in 40 columns or (*)80-columns	
49184 ($C020)	toggle cassette output switch	
.		
.		
.		
49200 ($C030)	toggle speaker	
.		
.		
.		
49216 ($C040)	utility strobe single pulse	
.		
.		

```
  .
49232 ($C050)   turns graphics mode on          turns graphics mode on
49233 ($C051)   turns text mode on              turns text mode on
49234 ($C052)   turns mixed mode off            turns mixed mode off
49235 ($C053)   turns mixed mode on             turns mixed mode on
49236 ($C054)   display from page 1 buffer      display from page 1 buffer
49237 ($C055)   display from page 2 buffer      display from page 2 buffer
49238 ($C056)   display graphics as lo-res      display graphics as lo-res
49239 ($C057)   display graphics as hi-res      display graphics as hi-res
49240 ($C058)   turn annunciator 0 off          turn annunciator 0 off
49241 ($C059)   turn annunciator 0 on           turn annunciator 0 on
49242 ($C05A)   turn annunciator 1 off          turn annunciator 1 off
49243 ($C05B)   turn annunciator 1 on           turn annunciator 1 on
49244 ($C05C)   turn annunciator 2 off          turn annunciator 2 off
49245 ($C05D)   turn annunciator 2 on           turn annunciator 2 on
49246 ($C05E)   turn annunciator 3 off          turn annunciator 3 off
49247 ($C05F)   turn annunciator 3 on           turn annunciator 3 on
49248 ($C060)   indicates if cassette input toggle has no bit or (*)has a bit
49249 ($C061)   indicates if game push button 0 (open apple) is up or (*)down
49250 ($C062)   indicates if game push button 1 (closed apple) is up or (*)down
49251 ($C063)   indicates if game push button 2 is up or (*)down
49252 ($C064)   indicates if game controller 0 has timed out or (*)not
49253 ($C065)   indicates if game controller 1 has timed out or (*)not
49254 ($C066)   indicates if game controller 2 has timed out or (*)not
49255 ($C067)   indicates if game controller 3 has timed out or (*)not
  .
  .
  .
49264 ($C070)   game controller strobe          game controller strobe
  .
  .
  .
49280 ($C080)   select RAM read bank 2.  Write-protect RAM.
49281 ($C081)   select ROM read. Two or more successive reads write-enables RAM. bank 2
49282 ($C082)   select ROM read. Write protect RAM
49283 ($C083)   select RAM read bank 2. Two or more successive reads write-enables RAM ba
49284 ($C084)   select RAM read bank 2.  Write-protect RAM.
49285 ($C085)   select ROM read. Two or more successive reads write-enables RAM. bank 2
49286 ($C086)   select ROM read. Write protect RAM
49287 ($C087)   select RAM read bank 2. Two or more successive reads write-enables R  ba
49288 ($C088)   select RAM read bank 1.  Write-protect RAM.
49289 ($C089)   select ROM read. Two or more successive reads write-enables RAM. bank 1
49290 ($C08A)   select ROM read. Write protect RAM
49291 ($C08B)   select RAM read bank 1. Two or more successive reads write-enables RAM ba
49292 ($C08C)   select RAM read bank 1.  Write-protect RAM.
49293 ($C08D)   select ROM read. Two or more successive reads write-enables RAM. bank 1
49294 ($C08E)   select ROM read. Write protect RAM
49295 ($C08F)   select RAM read bank 1. Two or more successive reads write-enables RAM ba
49296 ($C090) - 49311 ($C09F) slot 1 device select
49312 ($C0A0) - 49327 ($C0AF) slot 2 device select
49328 ($C0B0) - 49343 ($C0BF) slot 3 device select
49344 ($C0C0) - 49359 ($C0CF) slot 4 device select
49360 ($C0D0) - 49376 ($C0DF) slot 5 device select
49376 ($C0E0) - 49391 ($C0EF) slot 6 device select
```

49392 ($C0F0) - 49407 ($C0FF) slot 7 device select

APPLE//e HARDWARE AND SOFTWARE GUIDE LINES

The following are some suggestions for writing programs for the Apple IIe.

ERAL:

1. Apple has developed interface routines which are designed to help profesional and amateur programmers write 'friendly' interfaces for their programs. These routines also help the programmer avoid some pitfalls associated with using 80-columns. These routines are part of the Applesoft Extension Package. It can be found on the disk supplied with the Apple//e Applesoft Tutorial and Reference Manual. Appendix E of the new Applesoft Tutorial explains how to use this and other supplied routines. A 6502 Machine Language version of these routines will be available soon.

2. Apple has made every effort to maintain the subroutine entry points in the Autostart ROM when the Apple//e ROM was written and will continue to do so in future revisions. This implies that if you use only the entry points supported in the Apple II or Apple//e Reference Manuals your programs should not need to be modified for future revisions. It also implies that if you enter at other locations or if you do such activities as check-summing the ROM, your product may need to be reved when the ROM is reved.

   ogrammers be forwarned
   pple gives no assurance that any locations within the 80-column firmware ($C100-$CFFF) will be maintained. Therefore, programmers should not attempt to 'patch into' any of these routines. The 80-column firmware also uses several 'scratch pad' locations. At this time the only such location which will be maintained between revisions is location 1403 ($57B) which gives the current horizontal cursor location for 80 columns.

3. Use the procedures outlined in the IDENTIFICATION ROUTINES document to recognise the hardware that is available. These routines are available to licensed software developers from PCS Marketing Technical Support.

TWARE SPECIFIC:

1. Before using a peripheral for output be sure the 80 column firmware is inactive.

2. Do not require the use of the reset key during program operation unless you are not concerned that the bank-switched RAM will be switched out.

3. If your software turned on the 80 column firmware be sure it turns it off before ending.

4. Do not check for the absence of game control paddles by having your program 'look' to see if both game buttons have been pressed. An alternate method is to timeout the paddles for, say, twice as long as the normal count of 256; if the 558 timer chip still doesn't timeout, there must not be any paddles.

5. Make sure that an 80 column card exists prior to trying to turn it on since not doing this will lead to unpredictable results.

6. If your program requires DOS or BASIC commands to be typed be sure to instruct the end user to use upper case letters or better still use your program to shift input to upper case.

7. Applesoft BASIC was designed to produce flashing characters. Because of this, incorrect characters appear when lowercase inverse or flashing characters are displayed by an Applesoft program using the standard 40 column display. If an Applesoft program first determines that the 80 column card is there it may correctly display the lowercase inverse characters by turning on the card. A full set of lowercase flashing characters is not available.

8. If your program expects certain string input design it to accept both upper and lower case.

9. Never have your program issue the PR#0 or IN#0 commands while the 80 column card is active.

10. A program running under DOS should turn the 80 column card on by the command PRINT CHR$(4);"PR#3".

11. If your program is generating animated graphics you might want to use the vertical blanking signal to prevent 'blinking'.

12. The 80 COL soft switch $C00D (49165) should not be used if monitor input / output routines are used.

13. If your BASIC / Assembly Language software boots to run, include in your documentation the need to boot by pressing control - open apple - reset rather than by entering a PR#. This is to ensure that the hardware and firmware are in sync. An alternative if you are willing to put up with a momentary flash accross the screen is to have your greeting program's first actions be the following. First, it should determine if an 80 column card is in the system. If one is, then turn the firmware on using the PR#3 command. Finally, if you do not want the card on you may turn it off with a Print CHR$(21) command.

14. If your program is a BASIC program and it uses 80 columns then do not use commas to do tabbing. Instead use POKES to 1403. For example POKE 1403,10 TABS to the 10th. column.

15. If your program has 80 column firmware active (either 80 or 40 columns displayed) and you want to send output to a printer or other output device you must turn off the 80 column firmware before you turn on

the other device. The following is an example: Use Home to clear the screen. Turn off the 80 column firmware by issuing a control character to the screen(PRINT CHR$(21) -control-U). Turn on the printer. When printing is completed or you want an intermediate message on the screen urn the printer off with a PRINT CHR$(4)"PR#0" & PRINT CHR$(4)"IN#0". Then turn the 80 column firmware back on with a PRINT CHR$(4);"PR#3". If you must have a message on the screen durning printing then place the message on the screen (40 columns) after the 80 column firmware is turned off but before turning on the printer. NOTE: the PR#0 and IN#0 is not required by Apple's card but may be by other cards.

16. If the 80 column firmware is active the BASIC GET command and the monitor KEYIN routine will immediatly execute the escape keypress and so escape codes are not available. Therefore, do not use these 'GET' commands when escape sequences are required and the 80 column firmware is active.An Assembly Language or BASIC routine which properly get input by looking at 49152 ($C000) can be used to detect an escape key being pressed.

17. If your program uses a reset trap or in some way is designed to recover from a reset and it uses the bank-switched memory (language card) it must turn the bank-switched memory back on. This would be done by having your reset jump vector point to a reset routine placed somewhere in the the lower 48K of memory. This routine would need to turn the bank-switched memory back on before jumping back into the program.

RDWARE:

1. Don't use any of the four DB19 slots in the back panel since these are reserved for disk drives.

2. Cables should connect to the card at the keyboard end of the card since this gives the user more freedom in selecting the slot into which the card is to be installed. It also prevents cable cramping.

3. Cables should use DB9 or DB25 connectors.

4. Cards which require 'piggy backing' into IC sockets may become obsoleted by this and future revisions of the main board.

5. Do not require cards to be placed in slot three if they are intended to be used in systems having the 80 column card.

6. Cables using DB-25 connectors for parallel I/O devices should block pin seven. This convention should be followed to prevent damage should the connector be accidently pluged into a serial device. Serial devices use this pin for ground.

7. Cards should be identifiable according to the protocol outlined in Pascal's ATTACH document which is excerpt here.

   Pascal 1.1 uses four firmware bytes to identify the peripheral card. Both the identifying bytes and the branch table are near the beginning of the $Cs00 ROM space. The identifiers are listed in

Table 1.

Address	Value
$Cs05	$38
$Cs07	$18
$Cs0B	$01 (the Generic Signature of new FW cards)
$Cs0C	$ci (the Device Signature; see below)

Table 1. Bytes Used for Device Identification

The first digit, c, of the Device Signature byte identifies the device class as listed in Table 2.

Digit	Class
$0	reserved
$1	printer
$2	joystick or other X-Y input device
$3	serial or parallel I/O card
$4	modem
$5	sound or speech device
$6	clock
$7	mass storage device
$8	80-column card
$9	network or bus interface
$A	special purpose (none of the above)
$B-F	reserved for future expansion

Table 2. Device Class Digit

The second digit, i, of the Device Signature byte is a unique identifier for the card, assigned by Apple Technical Support.

NOTE: Our 80 column card identifier is $88

APPLE //e TECHNOTE #3

Original Version
Published by Softalk Magazine
Sept. 1983

This article describes the double hi-resolution display mode which is available in the Apple //c and the Apple //e (with the Extended 80-column card). Double Hi-res graphics provides twice the horizontal resolution and more colors than the standard high-resolution mode. On a monochrome monitor double hi-res displays 560 horizontal by 192 vertical pixels, while on a color monitor, 16 colors are available.

For further information contact:
PCS Developer Technical Support
M/S 22-W. Phone (408) 996-1010

PB

DOUBLE HI-RES ON THE APPLE //e

What is It?

The double high-resolution display mode that is available for the Apple //e provides twice the horizontal resolution of the standard high-resolution mode. On a standard black-and-white video monitor, standard hi-res displays 280 columns and 192 rows of picture elements (pixels); the double hi-res mode displays 560x192 pixels. On a color monitor, the standard hi-res mode displays up to 140 columns of colors, each color being selected from the group of six colors available, with certain limitations. Double hi-res displays 140 columns of color, for which all 16 of the low-resolution colors are available.

Table 1. Comparison of Standard and Double Hi-Res Graphics

	Black/White	Color
Standard Hi-Res	280 x 192 pixels	140 columns 6 colors
Double Hi-Res	560 x 192 pixels	140 columns 16 colors

How Do I Install It?

Installation of the double hi-res mode on your Apple //e depends on the following three conditions, discussed in detail below:

1. Presence of a Revision B motherboard
2. Installation of an extended 80-column text card with jumper
3. A video monitor with a bandwidth of at least 14 MHz

First, your Apple //e must have a Revision B ("Rev-B") motherboard. To find out whether your //e's motherboard is a Rev-B board, check the part number on the edge of the board nearest the backpanel, above the slots. If the board is a Rev-B board, the part number will be 820-0064-B. (Double hi-res does not work on systems containing a Rev-A motherboard.) If your //e's motherboard is not a Rev-B board, and if you want to obtain one, contact your local Apple dealer.

The second condition for installing double hi-res on your //e is that your //e must have an extended 80-column text card installed. This card must be installed with a jumper connecting the two Molex-type pins on the board.

WARNING: If your //e is a Rev-A machine, do NOT insert into it an extended 80-column card with the jumper connection mentioned above. The system will not work at all if you do.

The last requirement for operation in double hi-res mode is that your video monitor must have a bandwidth of at least 14 MHz. This bandwidth is necessary because a television set that requires a modulator will not reproduce some characters or graphic elements clearly, due to the high speed at which the computer sends out dots in this mode. Because most of the video monitors having a bandwidth of up to 14 MHz are black-and-white, the working examples in this article do not apply to color monitors. If you have a video monitor, please use it -- instead of a television set -- to display the following examples.

Your Turn to be Creative -- or, Volunteers, Anyone?

At this writing, no programs exist that support double hi-res graphics. Moreover, none of the standard hi-res commands (such as HPLOT) work properly in double hi-res mode. Until such routines are available, users must write their own. If you've gotten this far, and want to continue, you'll probably already have used the system monitor, and you'll probably need very few explanations. If not, please refer to the ?Apple_//e_Reference_Manual? and then return to double hi-res operations.

Before going into the subtleties of double hi-res, you should be acquainted with standard hi-res functions. If you aren't, obtain the Apple //e Reference Manual (Part Number A2L2005) or the Apple][Reference Manual (which, however, is out of print), and please read the sections on high-resolution graphics before proceeding with the hands-on practice explained below.

You can find another good explanation of these features in the Apple][Graphics column by Ken Williams, in Softline magazine. We suggest that you start with Volume 1, Number 1 (September 1981), available from Softalk Publishing, Inc. The early columns are especially useful.

The tutorial that occupies the rest of this article assumes you are working at your Apple //e as you read. The second part of the lesson demonstrates the double hi-res mode; therefore, before embarking on the second part, you should

install a jumpered extended 80-column card in your Rev-B Apple //e.

Hands-On Practice with Standard Hi-Res

The Apple //e hi-res graphics display is bit-mapped. In other words, each dot on the screen corresponds to a bit in the Apple //e's memory. For a real-life example of bit-mapping, perform the following procedure, according to the instructions given below. (The symbol "<cr>" indicates a carriage return.)

1. Boot the system, using the DOS system master diskette.

2. When the prompt ("]") appears, press the RESET key.

3. Engage the CAPS LOCK key, and type HGR<cr>. (This instruction should clear the top of the screen.)

4. Type CALL -151 <cr>. (The system is now in the monitor mode, and the prompt should appear as an asterisk (*).)

5. Type 2100:1 <cr>. One single dot should appear in the upper left-hand corner of the screen.

Congratulations! You have just plotted your first hi-res pixel. (Not an astonishing feat, but you have to start somewhere...)

With a black-and-white monitor, the bits in memory have a simple correspondence with the dots (pixels) on the screen. A dot of light appears if the corresponding bit is set (has a value of 1), but remains invisible if the bit is off (has a value of zero). (The dot appears white on a black-and-white monitor, and green on a green-screen monitor, such as Apple's Monitor ///. For simplicity, we shall refer to an invisible dot as a "black" dot or pixel.) Two visible dots located next to each other appear as a single wide dot, and many adjacent dots appear as a line. To obtain a display of another dot and a line, follow the steps listed below:

1. Type 2080:40 <cr>. A dot should appear above and to the right of the dot you produced in the last exercise.

2. Type 2180:7F <cr>. A small horizontal line should appear below the first dot you produced.

From Bits and Bytes to Pixels

The seven low-order bits in each display byte control seven adjacent dots in a row. A group of 40 consecutive bytes in

memory controls a row of 280 dots (7 dots per byte, multiplied by 40 bytes). In the screen display, the least-significant bit of each byte appears as the leftmost pixel in a group of 7 pixels. The second-least-significant bit corresponds to the pixel directly to the right of the pixel previously displayed, and so on. To watch this procedure in action, follow the steps listed below. The dots will appear in the middle of your screen.

1. Type 2028:1 <cr>.

2. Type 2828:2 <cr>.

3. Type 3028:4 <cr>.

The three bits you specified in this exercise correspond to three pixels that are displayed one after another, from left to right.

The most-significant bit in each byte does not correspond to a pixel. Instead, this bit is used to shift the positions of the other seven bits in the byte. For a demonstration of this feature, follow the steps listed below:

1. Type 2050:8 <cr>.

2. Type 2850:8 <cr>.

3. Type 3050:8 <cr>.

You'll notice that the dots align themselves vertically. Now:

4. Type 2450:88 <cr>.

The new dot (that is, the one that corresponds to the bit you just specified) does not line up with the dots you displayed earlier. Instead, it appears to be shifted one "half-dot" to the right.

5. To demonstrate that this dot really is a "new" dot, and not just the "old" dot shifted by one dot position, type 2050:18 <cr>, 2850:18 <cr>.

You'll notice that the dot mentioned under Step 4 above (the dot that was not aligned with the other seven dots) is straddled by the dots above and below it. (The use of magnifying lenses is permitted.)

Shifting the pixel one "half-dot", by setting the high, most-significant bit is most often used for color displays. When the high bit of a byte is set, to generate this shifted

dot (which is also called the "half-dot shift"), then all the dots for that byte will be shifted one half dot. The half-dot shift does not exist in the double hi-res mode for the Apple //e.

The following figure shows the memory map for the standard hi-res graphics mode:

BASE \ HORIZONTAL OFFSET	$00	$01	$02	$03	...	$24	$25	$26	$27
$2000					...				
$2080					...				
$2100					...				
$2180					...				
$2200					...				
$2280					...				
$2300					...				
$2380					...				
$2028					...				
$20A8					...				
$2128					...				
$21A8					...				
$2228					...				
$22A8					...				
$2328					...				
$23A8					...				
$2050					...				
$2000					...				
$2150					...				
$21D0					...				
$2250					...				
$22D0					...				
$2350					...				
$23D0					...				

Standard Hi-res Memory Map

The following figure shows the box subdivisions for the memory map shown in the figure above:

OFFSET FROM BASE	BIT 6	5	4	3	2	1	0 LSB
+$0000							
+$0400							
+$0800							
+$0C00							
+$1000							
+$1400							
+$1800							
+$1C00							

For example, the first memory address of each screen line for the first few lines is as shown below:

$2000
$2400
$2800
$2C00
$3000
$3400
$3800
$3C00
$2080
$2480, etc.

Each of the 24 'boxes' contains 8 screen lines for a total of 192 vertical lines per screen. Each of the 40 'box' per line contains 7 pixels for a total of 280 pixels horizontally across each line.

The Intricacies of Double Hi-Res

Because the double hi-resolution graphics mode provides twice the horizontal dot density as standard hi-res graphics does, double hi-res requires twice as much memory as standard hi-res does. If you spent many hours memorizing the standard hi-res memory map, don't despair. Double hi-res still uses the hi-res graphics page (but only to represent half the picture, so to speak). In the double hi-res mode, the hi-res graphics page is compressed to fit into half of the display. The other half of the display is stored in memory (called the "auxiliary" or "aux" memory) on the Extended 80-Column card. (This article refers to the standard hi-res graphics page, which resides in main memory, as the "motherboard" or "MB" memory.)

The auxiliary memory uses the same addresses used by the standard hi-res graphics page (Page 1, $2000 through $3FFF). The hi-res graphics page stored in auxiliary memory is known as "hi-res page 1X." The graphics pages in auxiliary memory

are bank-switched memory, which you can switch in by activating some of the soft switches. (Adventurous readers may want to skip ahead to "Using the Auxiliary Memory," which appears later in this article.)

The memory mapping for the hi-res graphics display is analogous to the technique used for the 80-column display. The double hi-res display interleaves bytes from the two different memory pages (auxiliary and motherboard). Seven bits from a byte in the auxiliary memory bank are displayed first, followed by seven bits from the corresponding byte on the motherboard. The bits are shifted out the same way as in standard hi-res (least-significant bit first). In double hi-res, the most significant bit of each byte is ignored; thus, no half-dot shift can occur. (This feature is important, as you'll see when we examine double hi-res in color.)

The memory map for double hi-res appears below:

	$0		$1		$2		$3		.....		$25		$26		$27	
	AUX	MB	AUX	MB	AUX	MB	AUX	MB	AUX	MB	AUX	MB	AUX	MB	AUX	MB
$2000									.....							
$2080									.....							
$2100									.....							
$2180									.....							
$2200									.....							
$2280									.....							
$2300									.....							
$2380									.....							
$2028									.....							
$20A8									.....							
$2128									.....							
$21A8									.....							
$2228									.....							
$22A8									.....							
$2328									.....							
$23A8									.....							
$2050									.....							
$20D0									.....							
$2150									.....							
$21D0									.....							
$2250									.....							
$22D0									.....							
$2350									.....							
$23D0									.....							

Double Hi-res Memory Map

Where each box is subdivided exactly the same way it is in the standard hi-res mode.

Obtaining a Double-Hi-Res Display

To display the double hi-res mode, set the following soft switches:

	In Monitor Read	In Applesoft PEEK
HI-RES	$C057	49239
GR	$C050	49232
AN3	$C05E	49246
MIXED	$C053	49235

	In Monitor Write	In Applesoft POKE
80COL	$C00D	49165,0

Annunciator 3 (AN3) must be turned off to get into double hi-res mode. You turn it off by reading location 49246 ($C05E hex). Note that whenever you press CTRL-RESET, AN3 is turned on; therefore, each time you press CTRL-RESET, you must turn AN3 off again.

If you are using MIXED mode, then the bottom four lines on the screen will display text. If you have not turned on the 80-column card, then every second character in the bottom four lines of text will be a random character. (The reason is that although the hardware displays 80 columns of characters, the firmware only updates the 40-column screen, which consists of the characters in the odd-numbered columns. The characters in even-numbered columns then consist of random characters taken from text page 1X in the auxiliary memory.)

To remove the "even" characters from the bottom four lines on the screen, type PR#3<CR> from basic (type 3^P from the monitor). This procedure clears the memory locations on page 1X.

Using the Auxiliary Memory

The auxiliary memory consists of several different sections, which you can select by using the soft switches listed below. A pair of memory locations is dedicated to each switch. (One location turns the switch on; the other turns it off.) You activate a switch by writing to the appropriate memory location. The WRITE instruction itself is what activates the switch; therefore, it doesn't matter what data you write to the memory location. The soft switches are:

		From Monitor Write	From Applesoft POKE
80STORE	off:	$C000	49152,0
	on:	$C001	49153,0
RAMRD	off:	$C002	49154,0
	on:	$C003	49155,0
RAMWRT	off:	$C004	49156,0
	on:	$C005	49157,0
PAGE2	off:	$C054	49236,0
	on:	$C055	49237,0
HIRES	off:	$C056	49238,0
	on:	$C057	49239,0

A routine called AUXMOVE, located in the monitor ROM of the Apple //e, is also very handy, as we'll see below. AUXMOVE is located at address C311.

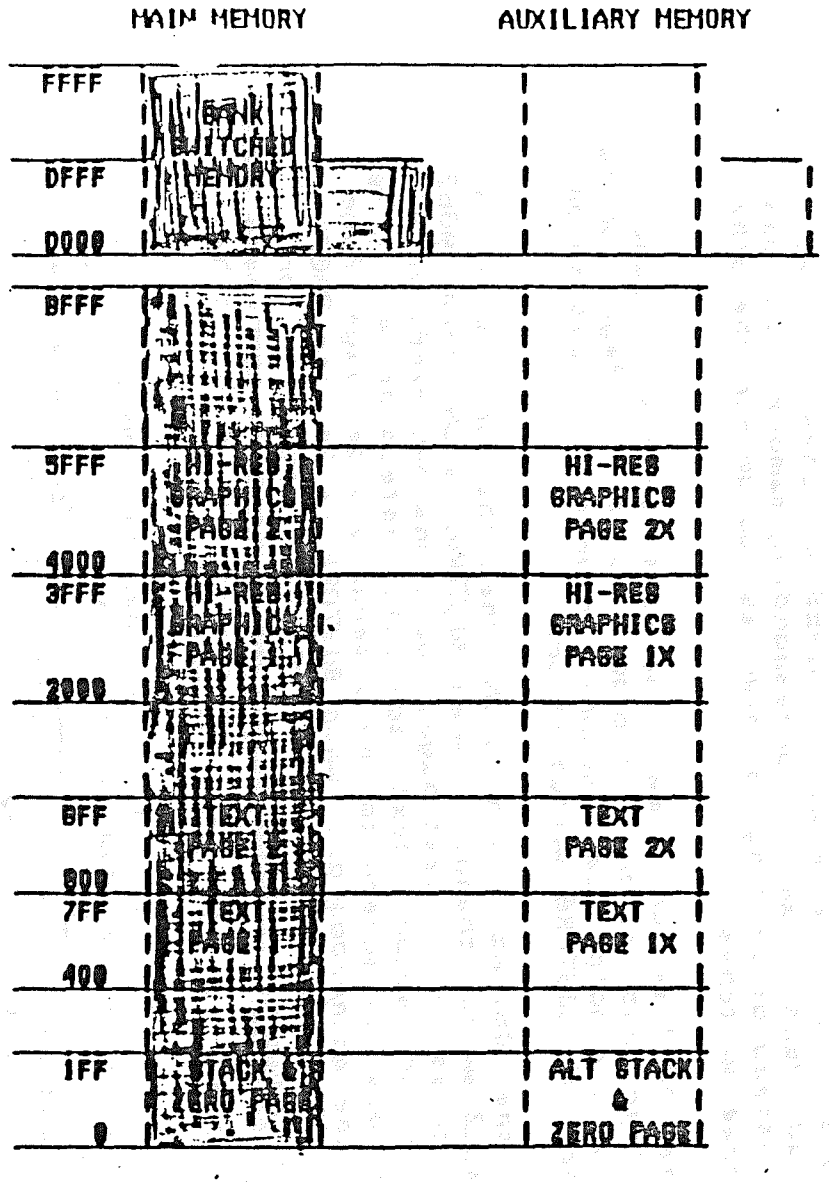
Accessing memory on the auxiliary card with the soft switches has the following characteristics. Memory maps, which help clarify the descriptions, are on the next page.

1). To activate the PAGE2 and HIRES switches, you need only read (PEEK) from the corresponding memory locations (instead of writing to them, as you do for the other three switches).

2). The PAGE2 switch normally selects the display page, in either graphics or text mode, from either Page 1 or Page 2 of the motherboard memory. However, it does so only when the 80STORE switch is OFF.

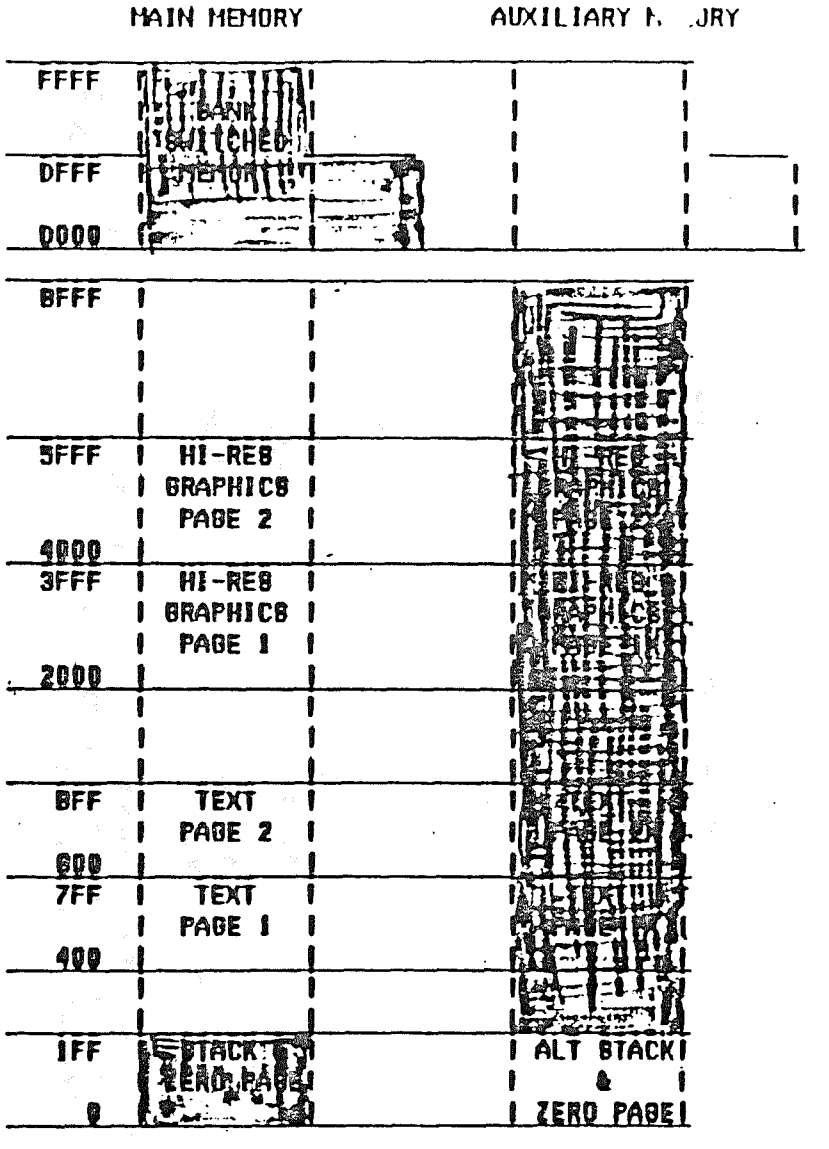
3). If the 80STORE switch is ON, then the function of the PAGE2 switch changes. When 80STORE is ON, then PAGE2 switches in the text page, locations $400-7FF, from auxiliary memory (text page 1X), instead of switching the display screen to the alternate video page (Page 2 on the motherboard). When 80STORE is ON, the PAGE2 switch determines which memory bank (auxiliary or motherboard) is used during any access to addresses $400 through 7FF. When the 80STORE switch is ON, it has priority over all other switches.

4). If the 80STORE switch is ON, then the PAGE2 switch only switches in the graphics page 1X from the auxiliary memory <u>if the HIRES switch is also ON</u>. (Note that this circumstance is slightly different from that described in Item 3 above.) When 80STORE is ON, and if the HIRES switch is also ON, then the PAGE2 switch selects the

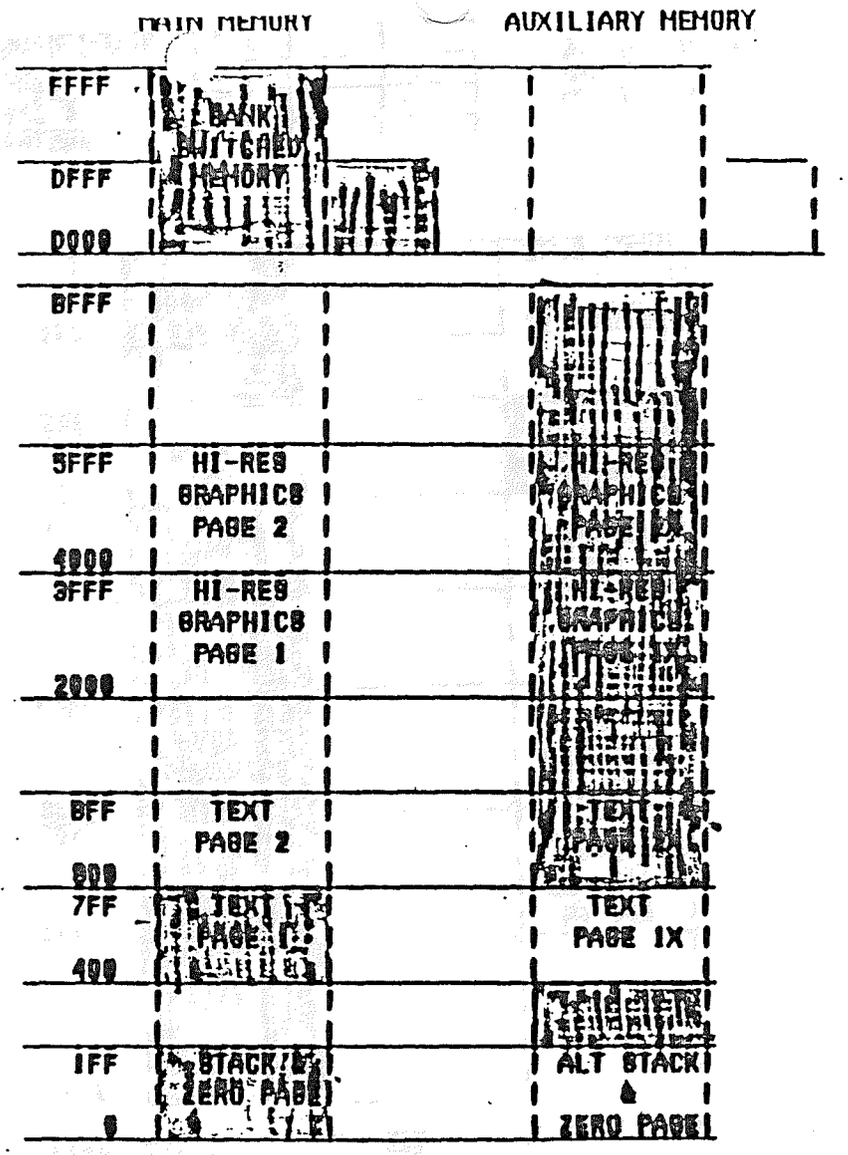


80STORE	OFF	ON	
PAGE 2	X	OFF	
HIRES	X	X	
RAMRD/RAMWRT	OFF	OFF	

ACTIVE MEMORY

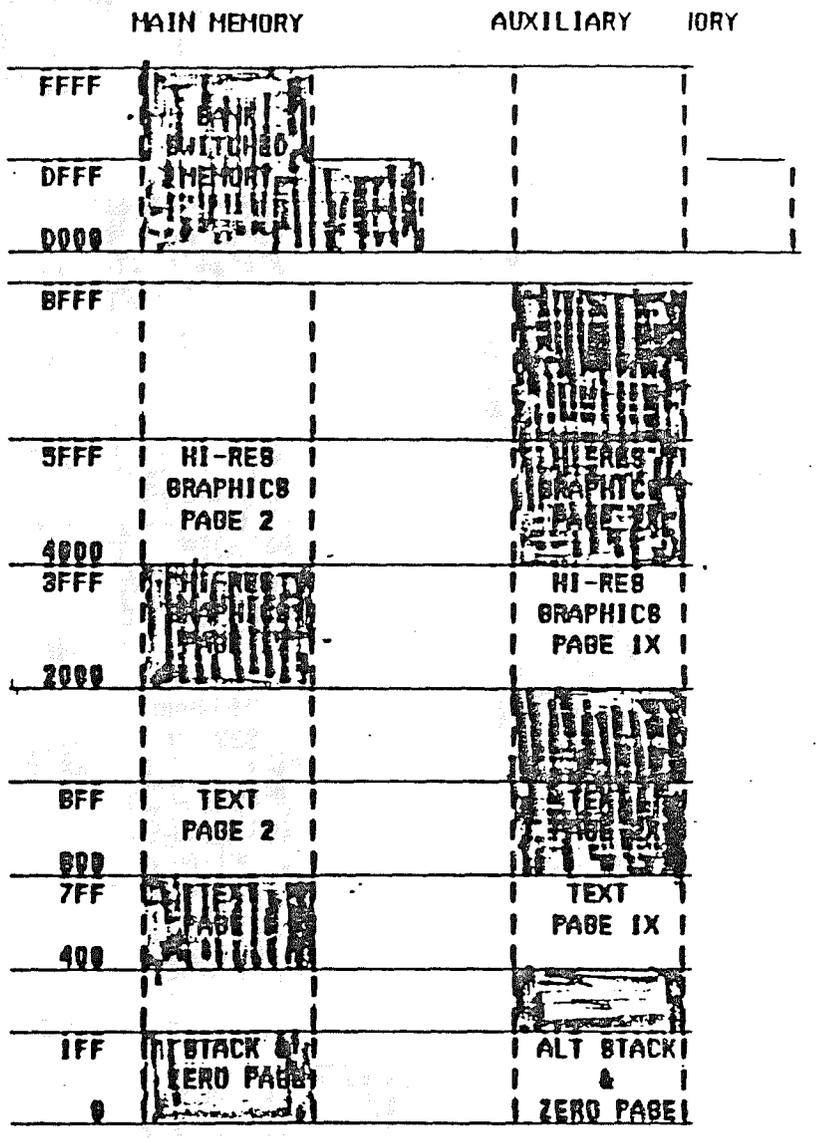


80STORE	OFF	ON	
PAGE 2	X	ON	
HIRES	X	X	
RAMRD/RAMWRT	ON	ON	

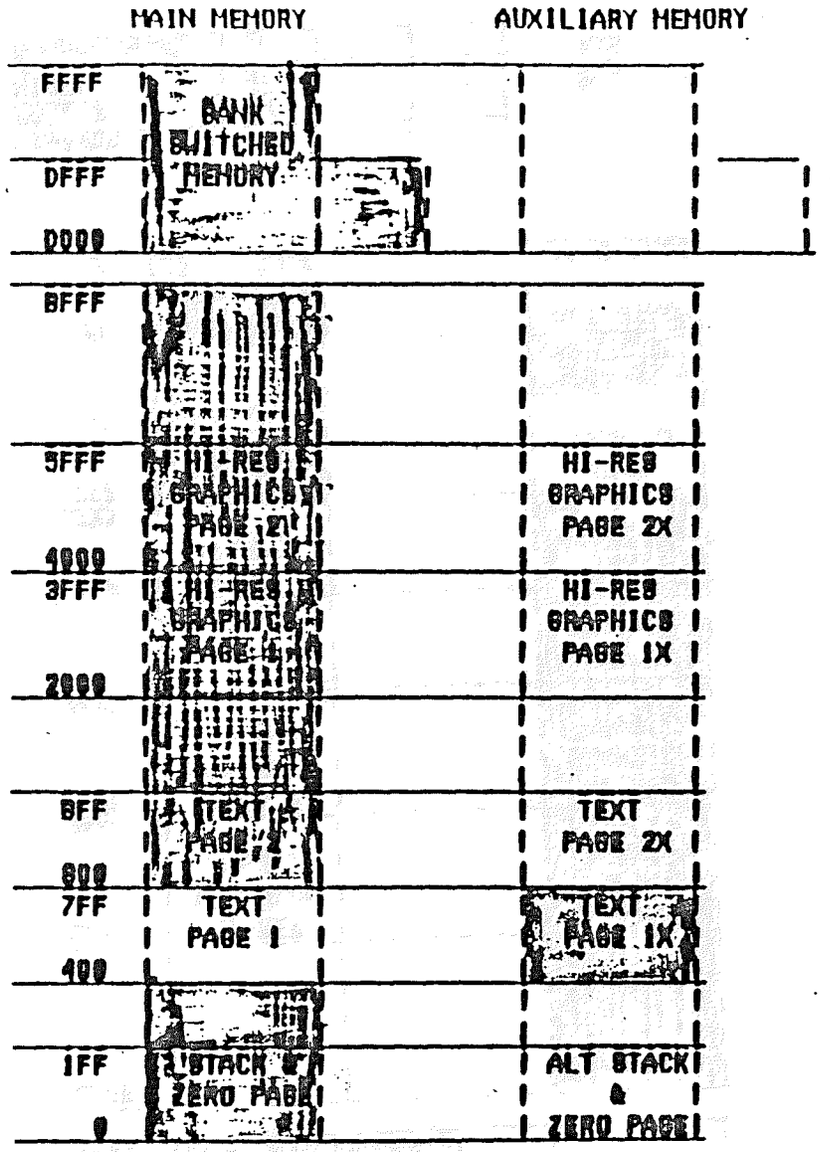


80STORE	ON			
PAGE 2	OFF			
HIRES	OFF			
RAMRD/RAMWRT	ON			

ACTIVE MEMORY

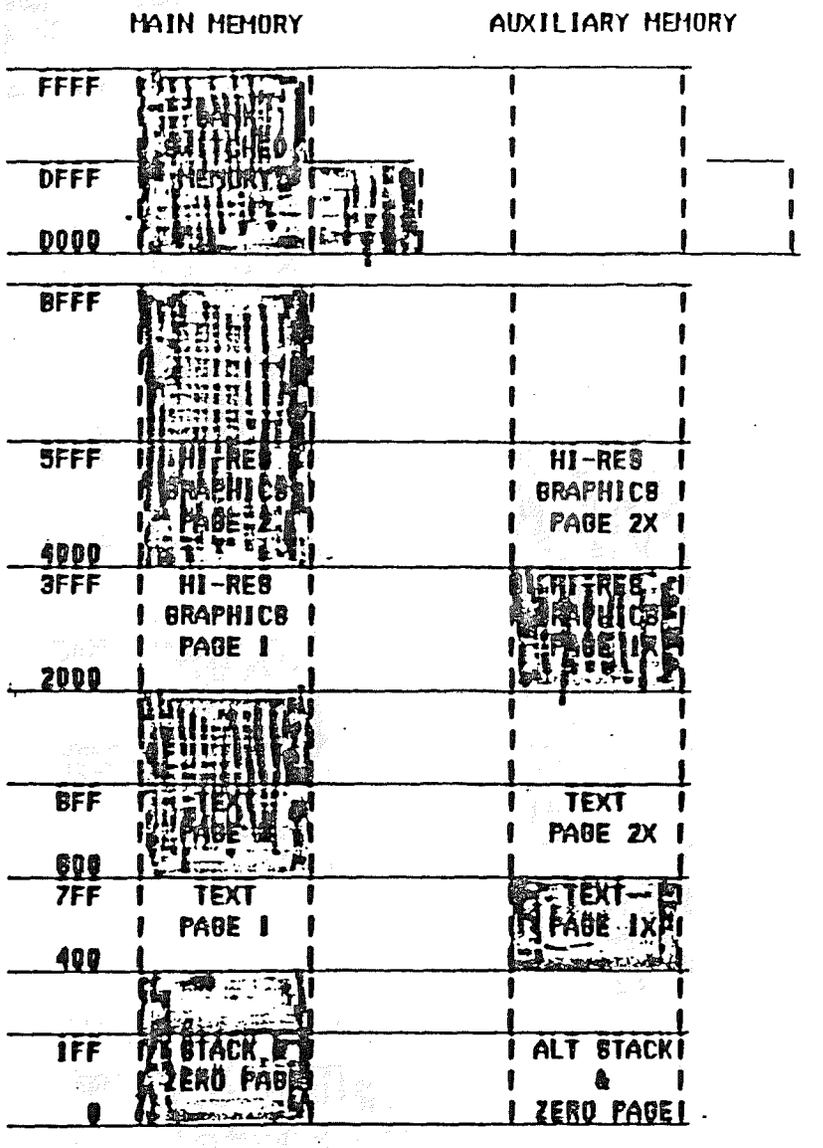


80STORE	ON			
PAGE 2	OFF			
HIRES	ON			
RAMRD/RAMWRT	ON			



80STORE	ON		
PAGE 2	ON		
HIRES	OFF		
RAMRD/RAMWRT	OFF		

ACTIVE MEMORY



80STORE	ON		
PAGE 2	ON		
HIRES	ON		
RAMRD/RAMWRT	OFF		

memory bank (auxiliary or motherboard) for accesses to a memory location within the range $2000 through 3FFF. If the HIRES switch is OFF, then any access to an memory location within the range $2000 through 3FFF uses the motherboard memory, regardless of the state of the PAGE2 switch.

5. If the 80STORE switch is OFF, and if the RAMRD and RAMWRT switches are ON, then any reading or writing to address space $200-$8FFF gains access to the auxiliary memory. If only one of the switches, for example RAMRD, is set then only the appropriate operation, in this case a read, will be performed on the auxiliary memory, while a write operation will access only the motherboard memory. If only RAMWRT is set then all write operations access the auxiliary memory. When The 80STORE switch is ON it has higher priority than the RAMRD and RAMWRT switches.

Shortcuts: Writing to Auxiliary Memory from the Keyboard
First, press CTRL-RESET. Next, type <CALL -151> (to get into the monitor). Then type the following hexadecimal addresses to turn on the double hi-res mode:

C057 (for Hi-res)

C050 (for Graphics)

C053 (for Mixed mode)

C05E Turns off AN3 for double hi-res

C00D:0 Turns on the 80COL switch

This procedure usually causes the display of a random-dot pattern at the top of the screen, while the bottom four lines on the screen contain text. To clear the screen, follow the steps listed below:

1). Type 3D0G to return to BASIC.

2). Type HGR to clear half of the screen. (The characters you type will probably appear in alternating columns. This is not a cause for alarm; as noted above, the firmware simply thinks you are working with a 40-column display.) Remember that hi-res graphics commands don't know about the half of the screen stored on page 1X in the auxiliary memory. Therefore, only page 1 (that is, the first half) of the graphics page on the motherboard is cleared. As a result, in the the screen display, only alternate 7-bit columns appear cleared.

On the other hand, if all of the screen columns were cleared after the HGR command, then chances are good that you're not in double hi-res mode. If your screen was cleared then to determine which mode you're in, type the following instructions:

CALL -151 back into monitor

2000:FF

2001<2000.2027M

If a solid line appears across the top of the screen, you're not in double hi-res mode. (The line that appears should be a dashed or intermittent line: - - - - - - - - across the screen.) If you're not in double hi-res mode, then make sure that you do have a Rev. B motherboard, and that the two Molex-type pins on the Extended 80-Column card are shorted together with the jumper block. Then re-type the instructions listed above.

If you're staring at a half-cleared screen, you can clear the non-blank columns by writing zeros to addresses $2000 through 3FFF on graphics page 1X of auxiliary memory. To do so, simply turn on the 80STORE switch, turn on the PAGE2 switch, and then write to locations $2000, $2001, $2002, and so on up through 3FFF. However: this procedure will not work if you try it from the monitor! The reason is that each time you invoke a monitor routine, the routine sets the PAGE2 switch back to page 1 so that it can display the most recent command that you entered. When you try to write to $2000, etc. on the auxiliary card, instead it will write to the motherboard memory.

Another way to obtain the desired result is to use the monitor's USER command, which forces a jump to memory location $3F8. You can place a JMP instruction starting at this memory location, so that the program will jump to a routine that writes into hi-res page 1X. Fortunately, the monitor already contains such a routine: AUXMOVE.

Using AUXMOVE

You use the AUXMOVE routine to move data blocks between main and auxiliary memory. But the task still remains of setting up the routine so that it knows which data to write, and where to write it. To use this routine, some byte pairs in the zero page must be set up with the data block addresses, and the carry bit must be fixed to indicate the direction of the move. You may not be surprised to learn that the byte pairs in the zero page used by AUXMOVE are also the scratch-pad registers used by the monitor during instruction execution. The result is that while you type the addresses for the monitor's move command, those addresses are being stored in the byte pairs used by AUXMOVE. Thereafter, you

can call the AUXMOVE command directly, using the USER (CTRL-Y) command.

In practice, then, enter the following instructions:

C00A:0	(turns on the 80-Column ROM, which contains the AUXMOVE routine)
C000:0	(reason explained below)
3F8: 4C 11 C3	(the jump to AUXMOVE)
2000<2000.3FFF ^Y	(where "^Y" indicates that you should type CTRL-Y.)

The syntax for this USER (CTRL-Y) command is:

(AUXdest)<(MBstart).(MBend)^Y	Copies the values in the range MBstart to MBend in the motherboard memory into the auxiliary memory beginning at AUXdest. This command is analogous to the MOVE command.

You can use this procedure to transfer any block of data from the motherboard memory to hi-res page 1X. Working directly from the keyboard, you can use a data block transferred this way to fill in any part of a double hi-res screen image. The image to be stored in hi-res page 1X (that is, the image that will be displayed in the even-numbered columns of the double hi-res picture) must first be stored in the motherboard memory. You can then use the CTRL-Y command to transfer the image to hi-res page 1X.

The AUXMOVE routine uses the RAMRD and RAMWRT switches to transfer the data blocks. Because the 80STORE switch overrides the RAMRD and RAMWRT switches, the 80STORE switch must be turned off -- otherwise it would keep the transfer from occurring properly (hence the write to $C000 above).

If the 80STORE and HIRES switches are ON and PAGE2 is off, when you execute AUXMOVE, then any access to an address located within the range from $2000 to $3FFF inclusive would use the motherboard memory, regardless of how RAMRD and RAMWRT are set. Entering the command C000:0 turns off 80STORE, thus letting the RAMRD and RAMWRT switches control the memory banking.

The CTRL-Y trick described above only works for transferring data blocks from the main (motherboard) memory to auxiliary memory (because the monitor always enters the AUXMOVE routine with the carry bit set). To move data blocks from

the auxiliary memory to the main memory, you must enter AUXMOVE with the carry bit clear. You can use the routine listed below to transfer data blocks in either direction:

301:AD 0 3	(loads the contents of address $300 into the accumulator)
304:2A	(rotates the most-significant bit into the carry flag)
305:4C 11 C3	(jump to $C311 <AUXMOVE>)
3F8:4C 1 3	(sets the CNTRL-Y command to jump to address $301)

Before using this routine, you must modify memory location $300, depending on the direction in which you want to transfer the data blocks. If the transfer is from the auxiliary memory to the motherboard, you must clear location $300 to zero. If the transfer is from the motherboard to the auxiliary memory, you must set location $300 to $FF.

Two Double Hi-Res Pages

So far, we've only discussed using graphics pages 1 and 1X to display double hi-res pictures. But -- analogous to the standard hi-res pages 1 and 2 -- two double hi-res pages exist: pages 1 and 1X, at locations $2000 through 3FFF, and pages 2 and 2X, at locations $4000 through 5FFF.The only trick in displaying the second double hi-res page is that you must turn off the 80STORE switch. If the 80STORE switch is ON, then only the first page (1 and 1X) is displayed. Go ahead and try it:

C000:0	to turn off the 80STORE switch
C055	to turn on the PAGE2 switch

The screen will fill up with another display of random bits. Clear the screen using the instructions listed above (in the section entitled "Using AUXMOVE"). However, this time, use addresses $4000 through 5FFF instead. (Don't be alarmed by the fact that the figures you're typing aren't displayed on the screen. They're being "displayed" on text Page 1.)

4000:0

4001<4000.5FFFM

4000<4000.5FFF ^Y

You'll be delighted to learn that you can also use this trick to display two 80-column text screens. The only problem here is that the 80-column firmware continually turns on the 80STORE switch, which prevents the display of the second 80-column screen. However, if you write your own 80-column display driver, then you can use both of the 80-column screens.

Color Madness

It should come as no surprise that color-display techniques in double hi-res are different from color-display techniques in standard hi-res. This is because the "half-dot shift" doesn't exist in double hi-res mode.

Instead of going into a disquisition on how a TV set decodes and displays a color signal, I'll simply explain how to generate color in double hi-res mode. In the following examples, the term "color monitor" refers to either an NTSC monitor or a color television set. Both work; however, the displays will be much harder to see on the color TV.The generation of color in double hi-res demands sacrifices. A 560x192-dot display is not possible in color. Instead, the horizontal resolution decreases by a factor of four (to 140 dots across the screen). Just as with a black-and-white monitor, a simple correspondence exists between memory and the pixels on the screen. The difference is that four bits are required to determine each color pixel. These four bits represent 16 different combinations: one for each of the colors available in double hi-res. (These are the same colors that are available in the low-resolution mode.)

Let's start by exploring the pattern that must be stored in memory to draw a single colored line across the screen. Start by pressing RESET; then load the program "COLOR TEST" from the DOS 3.3 sample programs disk (with the old Apple][+ DOS system master use the program "COLOR DEMOSOFT"). Use this program to adjust the colors displayed by your monitor. After you've adjusted the colors, exit from the color-demo program.

The instructions that appear below are divided into groups separated by blank lines. Because it's very difficult (and, on a TV set, almost impossible) to read the characters you're typing in as they appear on the screen, face it: you will make typing errors. If the instructions appear not to work, then start again from the beginning of a group of instructions.

CALL -151	(to get into the Monitor routine/program)
C050	(This set of instructions puts the
C057	computer into double hi-res mode.)
C05E	
C00D:0	

```
2000:0                  (This set of instructions clears first
2001<2000.3FFFM         one half of the screen, and then the
3F8: 4C 11 C3           other half of the screen.)
2000<2000.3FFF^Y

2100:11 4               (2 red dots appear on top left of screen)
2102<2100.2126M         (A dashed red line appears across screen)

2150:8 22               (Two green dots appear near bottom left)
2152<2150.2175M         (Dashed green line appears across screen)

2100<2150.2177^Y        (Fills in the red line)
```

In contrast to conditions in standard hi-res, no half-dot shift occurs, and the most-significant bit of each byte is not used.

As noted above, four bits determine a color. You can "paint" a single-color line across the screen simply by repeating a four-bit pattern across the screen. But it is much easier to write a whole byte rather than just change four bits at a time. Since only 7 bits of each byte are displayed (as noted earlier in our discussion of black-and-white double hi-res) and the pattern is four bits wide, it repeats itself every 28 bits or four bytes. Use the instructions listed below to draw a line of any color across the screen by repeating a four byte pattern for the color as shown in Table III below.

```
2200: mb1 mb2           (Colored dots appear at the left edge)
2202<2200.2226M         (A dashed, colored line appears)

2250: aux1 aux2
2250<2250.2276M

2200<2250.2276^Y        (Fills in line, using the selected color)
```

[see Table III on next page]

TABLE III. The Sixteen Colors

COLOR	aux1	mb1	aux2	mb2	REPEATED BINARY PATTERN
BLACK	00	00	00	00	0000
MAGENTA	08	11	22	44	0001
BROWN	44	08	11	22	0010
ORANGE	4C	19	33	66	0011
DARK GREEN	22	44	08	11	0100
GREY1	2A	55	2A	55	0101
GREEN	66	4C	19	33	0110
YELLOW	6E	5D	3B	77	0111
DARK BLUE	11	22	44	08	1000
VIOLET	19	33	66	4C	1001
GREY2	55	2A	55	2A	1010
PINK	5D	3B	77	6E	1011
MEDIUM BLUE	33	66	4C	19	1100
LIGHT BLUE	3B	77	6E	5D	1101
AQUA	77	6E	5D	3B	1110
WHITE	7F	7F	7F	7F	1111

In this table, "aux1" indicates the first, fifth, ninth, thirteenth, etc. byte of each line (i.e., every fourth byte, starting with the first byte). The heading "mb1" indicates the second, sixth, tenth, fourteenth, etc. byte of each line (i.e., every fourth byte, starting with the second byte). The "aux2" and "mb2" headings indicate every fourth byte, starting with the third and fourth bytes of each line, respectively. "Aux1" and "aux2" are always stored in auxiliary memory, while "mb1" and "mb2" are always stored in the motherboard memory.

As you'll infer from Table III, the absolute position of a byte also determines the color displayed. If you write an "8" into the first byte at the far left side of the screen (i.e., in the "aux1" column), then a red dot is displayed. But if you write an "8" into the third byte at the left side of the screen (the "aux2" column), then a dark green dot is displayed. Remember -- the color monitor decides which color to display based on the relative position of the bits on each line (i.e., on how far the bits are from the left edge of the screen).

So far, so good. But suppose you want to display more than one color on a single line. It's easy: just change the four-bit pattern that is stored in memory. For example, if you want the left half of the line to be red, and the right half to be purple, then store the "red" pattern (8, 11, 22, 44) in the first 40 bytes of the line, and then store the

"purple" pattern (19,33,66,4C) in the second 40 bytes of the line. Table III is a useful reference tool for switching from one color to another, provided you make the change on a byte boundary. In other words, you must start a new color at the same point in the pattern at which the old color ended. For example, if the old color stops after you write a byte from the "mbl" column, then you should start the new color by storing the next byte in memory with a byte from the "aux2" column. This procedure is illustrated below:

```
2028:11 44 11 44 11 44 11 77 5D 77 5D 77 5D
                (creates a dashed line that is red, then yellow)

2128: 8 22 8 22 8 22 8 22 6E 3B 6E 3B 6E

2028<2128.2134^Y          (fills in the rest of the colors)
```

Switching Colors in Mid-Byte

If you want a line to change color in the middle of a byte, you'll have to re-calculate the column, based on the information in Table III. Suppose you want to divide the screen into three vertical sections, each a different color. The left-hand third of the screen ends in the middle of the 27th character from the left edge -- that is, in an "aux2" column of the color table. (Dividing 27 by 4 gives a remainder of 3, which indicates the third column, or "aux2".) Your pattern should change from the first color to the second color after the 5th bit of the 27th byte. You can change the color in the middle of a byte by selecting the appropriate bytes from the "aux2" column of Table III, and concatenating two bits for the second color with five bits for the first color.

However, because the bits from each byte are shifted out in order from least significant to most significant, the two most significant bits (in this case I mean bits 5 and 6, because bit 7 is unused) for the second color are concatenated with the five least significant bits for the first color. For instance, if you want the color to change from orange (the first color) to green (the second color), then you must append the two most significant bits (5 and 6) of "green" to the five least significant bits (0-4) of "orange." In Table III, the "aux2" column byte for green is 19, and the two most significant bits are both clear. The "aux2" column byte for orange is 33, and the five least significant bits are equal to 10011. The new byte calculated from appending green (00) to orange (10011) yields 13 (0010011). Therefore, the first 26 byte of the line come from the table values for orange; the 27th byte is 13, and the next 26 bytes come from the table values for green.

```
2300: 19 66                (puts orange line on screen)
2302<2300.2310M
```

The double hires screen has 140 columns, numbered 0 through 139, and 192 rows, numbered 0 to 191. Just like the standard hi-res screen, the origin is in the upper left corner, while the point 139,191 is in the bottom right corner.

The color codes are the same as for lo-res graphics:

0:black
1:magenta
2:dark blue
3:violet
4:dark green
5:grey1
6:medium blue
7:light blue
8:brown
9:orange
10:grey2
11:pink
12:green
13:yellow
14:aqua
15:white

Some exercises you may want to try include painting the left half of the screen with grey1 and the right half with grey2 to see if they are different or moving a colored ball on different colored background. For the adventerous type, you may want to rewrite brickout (super brickout).

The following program shows off double hi-res. It starts with the color bar demo, except in this case the color bars can be much narrower than was possible in low resolution graphics. The next screen shows a simple picture of an orange line drawn diagonally on a green background. These two colors are also available in standard hi-res, but as you'll see in the next picture there are certain limitations.

[[RUN DEMO]]

In double hi-res the most significant bit is not used, and any color can appear next to any other color, anywhere on the screen (though "fringing" can occur where the colors join). In standard hi-res the most significant bit of each byte limits that byte to four of the six colors. If the MSB is set than than the only colors displayed by that byte are white, black, blue, and orange. Therefore since green and orange can't be displayed in the same byte, the whole byte becomes orange, and the stair step line appears.

By the way, if Annunciator 3 (AN3) is turned off when a jumpered extended 80-column card is present, then the most significant bit of standard hi-res isn't used either. This

means that any standard hi-res picture will display only black, white, violet or green. If the picture contains blue or orange, then those colors will be converted to violet or green. Go ahead and try it: pull out a game that uses all four colors, turn the AN3 off with PEEK (49246), and then, without pressing RESET (since that sets AN3 on), run the program (RUN HELLO sometimes works).

Now you've got the tools and the rules to the double hi-res mode. As you can see double hi-res has more color with higher resolution than standard hi-res. You can even develop games that do fancy animation or scroll orange objects across green backgrounds. In black and white, word processing programs that use different fonts or proportional character sets can be developed. Have fun playing with the this new mode and I hope I'll see some of your programs soon.

[[I've got two more demo programs if there is room:]]

```
RUN DOUBLENET     (Remember Brians's theme)
BRUN QIX
```

APPLE //e TECHNOTE #4

Revision of RDY TECHNOTE 1-April 83*
1-July 84

This article describes an input signal into the 6502 microprocessor called the RDY line. The RDY line allows a peripheral card to halt the microprocessor with the output address lines reflecting the current address being fetched. If a peripheral device can not get data on the bus fast enough to meet the set up time of the 6502 then the peripheral card can pull the RDY line low and tell the 6502 to wait. This allows the peripheral device enough time to get the proper data on the bus. This article describes the timing for as event such as this.

For further information contact:
PCS Developer Technical Support
M/S 22-W. Phone (408) 996-1010

PB

Peter Baum
20525 Mariani Ave
Cupertino, Ca. 95014

Apple Computer
M.S. 22-W

July 1, 1984

Using the RDY Line on the Apple //e and Apple][+

Though the 6502 was one of the first commercially successful microprocessors sold, the designers had forsight to include some very useful functions. Because many early peripherals products were very slow devices a microprocessor could not read from the device directly. To connect these slow devices onto the Apple peripheral bus, so that the 6502 can read data from them, requires either buffering the device or slowing down the processor. Though most people would try to buffer the device, sometimes it is not feasible. For example, the 2 ms. access time of a 1-megabit CMOS ROM makes buffering a nightmare, since both the address and data bus have to be buffered. When buffering isn't possible then a peripheral device can pull the RDY line to slow down the processor long enough to read a byte. This technique can be used by slow devices to communicate with the 6502.

The RDY line allows a peripheral card to halt the microprocessor with the output address lines reflecting the current address being fetched. If a peripheral device can not get data on the bus fast enough to meet the set up time of the 6502 then the peripheral card can pull the RDY line low and tell the 6502 to wait. This can not be done during a 6502 write cycle because the 6502 will not hold up.

In order for the 6502 to read a valid data byte from a peripheral card, the card has about 800 ns. from the time the addresses are valid to put the data on the bus. The data must be set up on the bus within approximately 400 ns. from the time that the I/O STROBE, I/O SELECT, or DEVICE SELECT signal on the peripheral slot goes true. If a device pulls the RDY line low for one cycle then the device will have 1.4 usec., instead of the 400 ns., to put out valid data. The RDY line can be pulled low for more than one cycle; in fact, there is no limit. A device that takes 100 us. to send data can just hold the RDY line low for 100 cycles. Hence, this technique will allow any slower device to get on the bus and send data to the 6502.

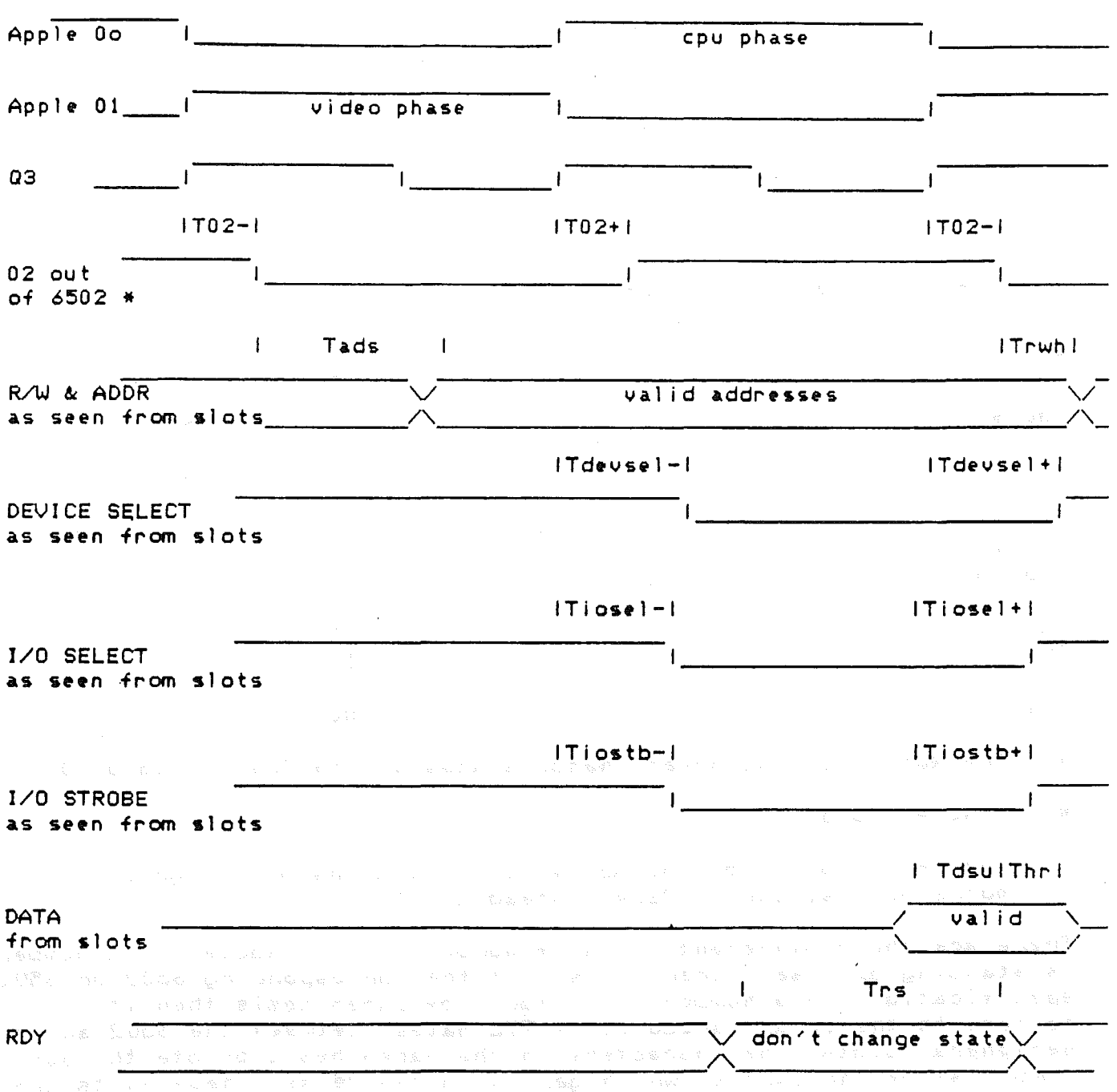
The RDY line is typically pulled low during 01, but the specification sheets for the 6502 show that it can be pulled anytime before the last 200 ns. of 02. The 02 line is not used by the Apple, but is an unused output from the 6502. It is basically the same as the 0o line with a little delay. Before I explain when to use (or not use in some cases) the RDY line, let us first look at some timing diagrams of the Apple system.

The timing diagram on the next page shows the relationship between the 6502 and Apple //e and Apple][+. The timing specifications have been adjusted to reflect the signals as they are seen from the peripheral slots. For example the 6502 (1 MHZ.) specification guarantees that the address bus will be valid within 225 ns. from 02 out. But the peripheral slots do not see these address lines directly; Instead the address lines go thru a buffer and then out to the peripheral slots. This adds a maximum delay of 13 ns. in the Apple][and 18 ns. in the Apple //e. The timing diagrams will show, in the case of an Apple][, that the address bus will be valid to the peripheral slots within 238 ns. (225+13) of 02 falling edge.

The major differences in timing between the Apple][+ and the Apple //e are due to the processor. The Apple][uses a 1 MHZ. 6502, while the Apple //e uses a 6502A, which is a 2 MHZ. part. This does not mean that the system clock in the Apple //e runs any faster, only that the 6502A is capable of running faster. This results in better timing margins. For example, the address and data busses are set up faster in the Apple //e by the 6502A than the 6502 sets them up in the Apple][. (This was done because the custom chips in the Apple //e are slower than the discrete logic in the Apple][and the 6502A was needed to compensate for this).

A peripheral card which uses the RDY line can only be used under certain circumstances. Because pulling the RDY line low halts the processor, any program with a software timing loop will not work properly. These programs assume that each instruction will take a fixed amount of time, which is not true when the processor stops in the middle of an instruction. An Apple][disk is an example of a peripheral which requires timing loops and won't run properly if the RDY line is used.

TIMING SIGNALS AS SEEN FROM PERIPHERAL SLOTS



* - 02 is an output signal from the 6502 which is not used by the Apple. It is a delayed 0o.

FIGURE 1

TIMING SPECIFICATIONS FOR FIGURE 1
(all times in ns.)

	Apple II 1 MHZ. 6502		Apple //e 2 MHZ. 6502A	
Symbol	min.	max.	min.	max.
T02- #	15	50+20 (LS08)	15	50+5 (S02)
T02+ #	30	80+15 (LS08)	30	80+5 (S02)
Tads		225+13 (8T97)		140+18 (LS244)
Trwh	30		30	
Tdevsel-		96 (3 x LS138)		65 (LS154+LS138)
Tiosel-		64 (2 x LS138)		38 (LS138)
Tiostb-		32 (LS138)		15 (LS10)
Tdevsel+		18 (LS138)		30 (LS154)
Tiosel+		36 (2 x LS138)		18 (LS138)
Tiostb+		18 (LS138)		15 (LS10)
Tdsu	100+17 (8T28) ^		50+12 (LS245)	
Thr	10		10	
Trs *	200		200	

* - The RDY line must never change states within Trs to end of 02.

- load = 100 pf.

^ - The RFI versions of the Apple II+, revisions A through D motherboards, use an 8304 instead an 8T28.

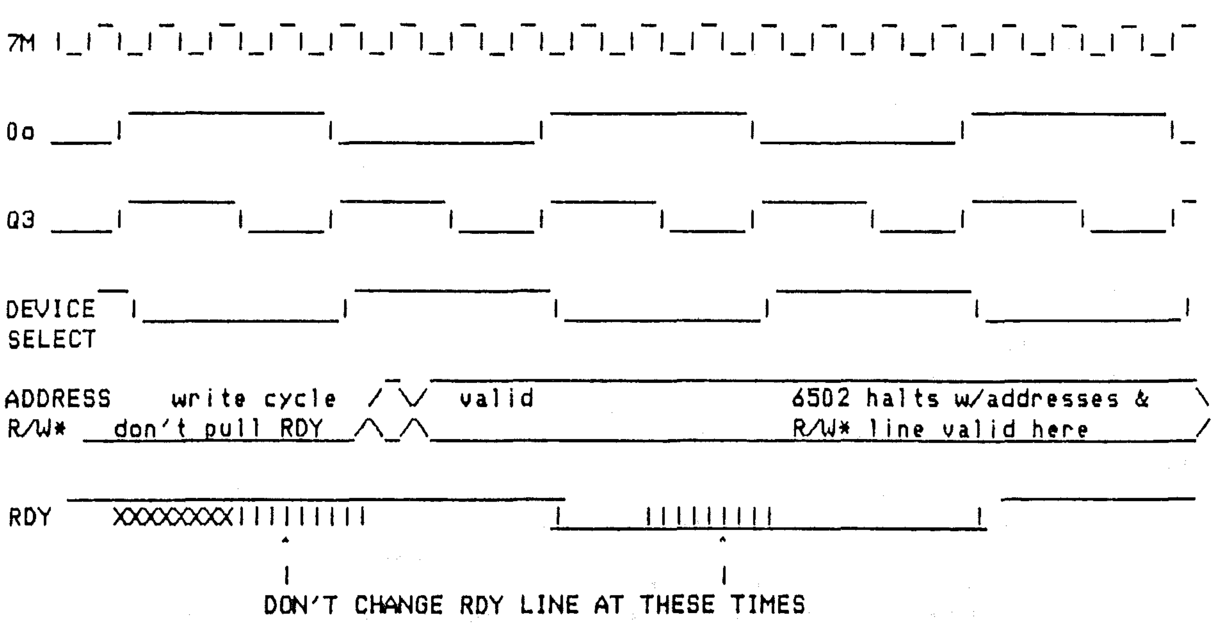
There are three different types of numbers listed above. If a number is standing by itself then it is just the corresponding 6502 or 6502A specification. If a number is followed by parenthesis then it represents the delay, produced by TTL gates, between the 6502 and the peripheral slots. The characters in the parenthesis denote the part number(s) of the part(s) which generated the delay. These parts are typically 74' series TTL except for the 8T28 and 8T97. If there are two numbers in a column with a "+" then the first number signifies the 6502 specification and the second the TTL delay, with the corresponding part number. Most of the TTL delay times are from the Texas Instrument data books. The 6502 specifications are from the Synertek 6502 data sheet and from Synertek application note AN2 - SY6500.

WHEN THE RDY LINE CAN BE CHANGED AND WHEN IT CAN'T

As can be seen from these diagrams, the RDY line should not be gated with the 0o trailing edge since this happens around the same time as the falling edge of 02. This would violate the Trs specification and probably force the 6502 to perform erratically. Gating the RDY line with the trailing edge of Q3 during 0o might work, but this could leave as little as 25 ns. for the signal to be valid. In other words Q3 must enable the RDY line low within 25 ns. of Q3 changing states. If this output cannot be guaranteed stable, then the RDY line might violate the Trs specification.

The safest time to pull the RDY line is using the 0o rising edge, but this edge occurs before I/O SELECT, I/O STROBE, or DEVICE SELECT is enabled. Therefore this scheme will not work if any of these three enables is used by the peripheral card. For example, many peripheral cards use memory mapped I/O to transfer data, with the cards registers designed to reside in the DEVICE SELECT memory space. Location C0n0 (where n = 8 + slot number of peripheral card) might hold the status of the card, and location C0n1 might be used to read a device such as a disk or an A/D converter. The card uses the DEVICE SELECT signal, pin 41 on the slot, and the 4 low order address lines to determine if the 6502 wants to read the status register or read from the A/D converter. Typically, the status register can put its data on the bus within 200 ns., easily meeting the set-up requirements of the 6502. But the A/D converter might take at least 100 us. before it can respond with data. The RDY line must be pulled low to allow time for the A/D converter to set up the data bus. Notice that the peripheral card doesn't know that it should pull the RDY line low until after the)EVICE SELECT signal has gone low. This signal doesn't go low until after 0o goes high, so the 0o rising edge can't be used to enable the RDY line for this peripheral card.

There are a few ways around this problem. One solution would be to decode the C0n0 address on the peripheral card and not use DEVICE SELECT. This also requires either putting user selectable switches on the card for setting the slot number, or making the card slot dependent. Another solution is to pull the RDY line low using one of the first three edges, trailing or leading, of the 7M clock. These edges occur at 70, 140, and 210 ns. into 0o and are trailing, leading, then trailing edges, respectively. The best solution is to use the DEVICE SELECT signal to enable the RDY line. The following timing diagram should help.



DON'T PULL RDY DURING WRITE CYCLES

Because there is no acknowledge response from the 6502, the peripheral card must do some of its own housekeeping and check if a write cycle is taking place. On write cycles the 6502 will not halt, but continue running until the next read cycle. After a slow peripheral pulls the RDY line and before it tries to get on the bus, it must make sure the 6502 is not in the middle of a write cycle. Otherwise there will be a bus crash, as both the peripheral card and 6502 try to drive the bus. One simple way to prevent this bus crash from occuring is to make sure the peripheral card doesn't pull the RDY line low during a write cycle. This can be guaranteed by checking the R/W* line when Oo goes high or DEVICE SELECT goes low. The R/W* line will be stable by this time.

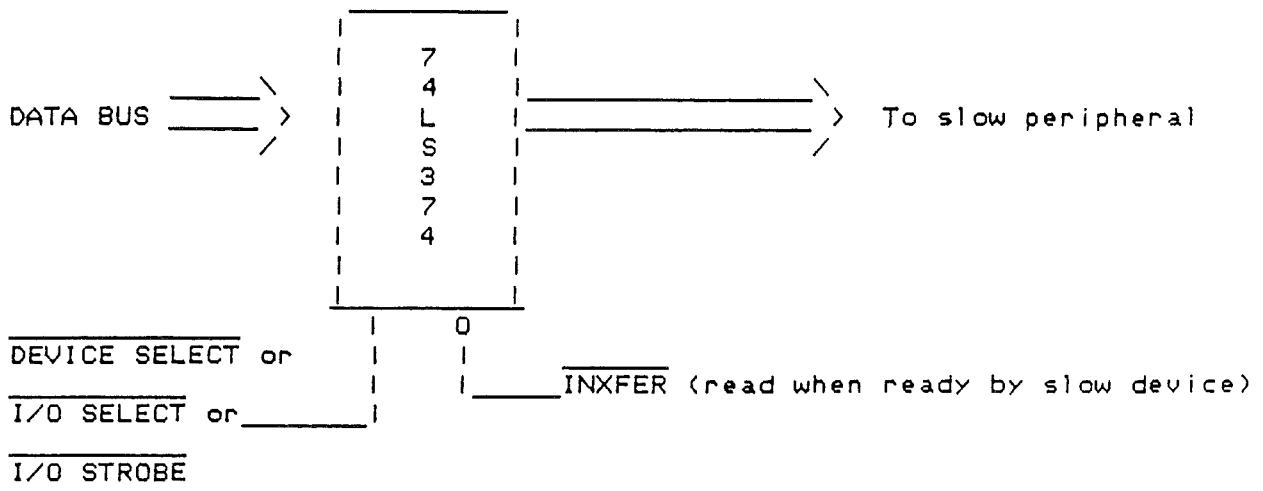
RELEASING THE RDY LINE

When the RDY line is released the 6502 will continue the cycle that was originally halted and allow the 6502 to read the data bus. Data will be read on the next trailing edge of 02 by the 6502, as long as RDY doesn't change within Trs of the end of 02. When the peripheral device has set the data bus up with the correct data it can release the RDY line to complete the read cycle. Releasing the RDY line has exactly the same constraints as pulling the line; Do not change RDY within 200 ns. of the end of 02.

The RDY line can be released before data has been set up, if the data will be valid within specification. This means that RDY can be released in the middle of 01 if the data bus will be valid 117 ns. before 02 trailing edge, for the Apple][(62 ns. for the Apple //e).

SLOW WRITES

Since the 6502 can't be halted during write cycles, if a device requires longer than one cycle to receive data then the data must be buffered. Here is an example of how to accomplish this:



NOTE: It is very easy to overrun the slow peripheral using this scheme, since it only buffers one byte at a time. Don't send data twice to the buffer within the maximum allowed time between slow peripheral reads.

APPLE //e TECHNOTE #5

5-JAN 84

One of the new features of the Apple //e is the ability to add more memory or overide existing memory from a peripheral card. This feature, which uses the INH (inhibit) line on the peripheral slots, has been expanded from its original purpose on the Apple II+ of disabling the onboard ROM and allowing the language card (RAM) to reside in the same address space. The Apple //e allows any part of memory to be replaced by memory on a peripheral card. This article explains how a peripheral card should use the INH line.

For further information contact:
PCS Developer Technical Support
M/S 22-W. Phone (408) 996-1010

PB

Peter Baum
Developer Technical Support
M.S. 22-W

Apple Computer
20525 Mariani
Cupertino, Ca. 95014

Jan. 5, 1984

Using the INH line on the Apple //e

One of the new features of the Apple //e is the ability to add more memory or overide existing memory from a peripheral card. This feature, which uses the INH (inhibit) line on the peripheral slots, has been expanded from its original purpose on the Apple][+ of disabling the onboard ROM and allowing the language card (RAM) to reside in the same address space. The Apple //e allows any part of memory to be replaced by memory on a peripheral card.

USES

Presently, only a few peripheral devices use the INH line in the //e for memory expansion. One type of card uses INH for RAM expansion by switching in extra language cards, while another class of cards uses it to extend the built-in 80-column ROM code by replacing it with their own ROM code. Other cards use INH so that they can have more than one stack and zero page. Future peripheral cards can take advantage of the INH line to do even fancier memory expansion, such as keeping multiple programs running in memory at the same time.

More memory, either ROM or RAM, can be added by mapping the memory into the same address space as existing memory. The processor can then select which memory, the onboard or the additional, it wants to use by setting a register (or softswitch). This technique of switching different blocks of memory into the same address space is called bank switching. An example of this technique for extending memory is found in the Apple][+ language card and in the bank switched memory on the //e.

HOW IT WORKS

When the INH line, pin 32 in slots 1-7, is pulled low, all memory on the motherboard and in the auxiliary slot is disabled (including memory on the 80-column and extended 80-column cards). This allows a peripheral card, in slots 1-7, to enable it's memory onto the bus.

When the 6502 reads a byte from memory the data typically comes from one of three places: motherboard ROM, motherboard RAM, or RAM on one of the 80-column cards in the auxiliary slot. When the INH line is pulled low, all of the above mentioned ROM anD RAM is disabled and will not drive the data bus. This allows the peripheral slots to drive the bus by enabling data onto it. The 6502 will then read data from the peripheral card instead of a location on the motherboard or auxiliary slot.

During a 6502 write cycle, if the INH line is pulled low, then motherboard and auxiliary card RAM are both disabled. A peripheral card can then read a byte off the data bus and store it away.

IMPLEMENTATION

Because pulling the INH line low disables all of memory, the peripheral card must be very careful when it does this. If only a small piece of memory is to be banked into a specific address space, then the INH line should only be pulled on memory references to that address space. Otherwise the motherboard memory will be disabled and the processor will read/write to the wrong memory and the program won't work properlyn For example, if a peripheral card wants to replace the zero page with memory on the card, then the INH line should be pulled low only on references to the address space between $0 and $FF. If the INH line is pulled during a processor instruction fetch from the monitor ROM at $F800, the 6502 will read the wrong instruction (or a floating bus) and probably crash the program.

Pulling the INH line at specific addresses is called select decoding. The hardware on the peripheral card does this by checking the address bus of the 6502, and if the address falls in the correct range the card pulls the INH line low. In the earlier example of a new zero page, if the address bus was in the range $0-$FF the card would pull INH low.

DIFFERENCES: //e vs. II+

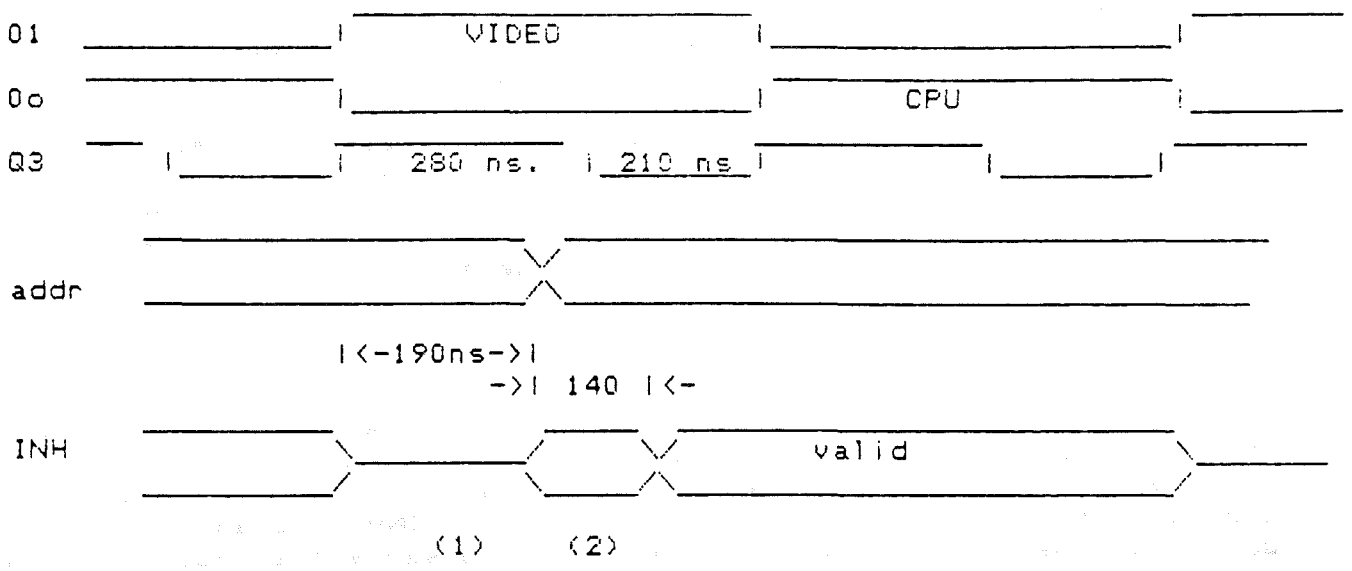
On the Apple II+, select decoding was not necessarily needed because the INH line only affected the ROM and not the RAM. If the Apple II+ peripheral card wanted to bank in extra language cards at 4he addresses $D000-$FFFF then it could pull the INH Line and keep it low during any memory access. This would just disable the onboard ROM and not any other memory accesses such as zero page or stack. This same card would not work in the //e, since the next instruction fetch to RAM after pulling INH low would read a floating bus because all the memory would be disabled.

ANOTHER FEATURE

For those of you who love to muck around in the guts of the Apple //e one more feature has been added to the INH function. The INH line will also overide DMA accesses to memory on the motherboard. This means that if a peripheral card uses DMA to read or write to memory, another peripheral card could pull the INH line and process the DMA access. An example of this would be a co-processor card using the memory on a RAM card in another slot. Rather than have the co-processor write to the memory on the motherboard and then have the 6502 write to the RAM card, the co-processor can write to an address that the RAM card recognizes. The RAM card could then pull the INH line and it would be free to read or write the data bus. This technique could also be used by a co-processor to write directly to a printer card in another slot.

TIMING

The peripheral card must wait for the address bus to settle, which occurs a maximum of 190 ns. after the falling edge of 0o, before pulling the INH line. (The 6502A maximum address setup time is 140 ns. from 02, with a worst case 6502A skew of 50 ns. from 0o to 02.) To guarantee that the RAM is disabled and a write doesn't accidentally take place to the motherboard, the INH line must be pulled low within 330 ns. of 0o.



(1) The INH line can be pulled high at this time.
(2) The INH line can be pulled low (or high) after the addresses are valid at 190ns, but before 330 ns. (from 0o).

CIRCUITS

A simple example of a circuit that can be used to implement the INH function is shown below.

```
                 ________
                |        |
A15 ----------- | select |
A14 ----------- | decode |                  _________
 .        .     | logic  |                |         |
 .        .     |        |_______________|D   7   Q|_______INH*  J[1-7]32
A0  ----------- |________|  PULLINH*     |    4    |
                                          |    L    |
                  |\                      |    S    |
Q3---------------| >o-------------------|>   3    |
                  |/                      |_   7    |
0o-------------------------------------|G   9    |
                                          |_________|
```

AN APPLICATION

The following circuit can be used to replace the code in the monitor ROM, from location $FC00 to $FFFF, with custom code. Any time the address space between $FC00-$FFFF is accessed the INH line is pulled low, the motherboard memory is disabled, and the circuit's 1K RAM is enabled instead. Part of this feature can be disabled and the motherboard memory can be read by keeping the switch connected to +5 volts (READDIS). Whenever the system writes to any location in the address space $FC00-$FFFF, the circuit will disable any RAM on the motherboard and instead write into the 1K RAM.

Here is a series of commands that can be used with the circuit to replace the reset vector at $FFFC and $FFFD. A new reset routine can be written that will print the screen or save the status of all the registers whenever the reset key is pressed.

Start the system with the circuit's switch connected to +5 (READDIS). This will enable the system to read the monitor ROM during power up, before the 1K RAM has been initialized.

Get into the monitor by typing CALL -151. The system prompt should now be a '*'.

Copy the monitor ROM into the 1K RAM with the command FC00<FC00.FFFFM <CR>

Change the reset vector so that it jumps to location $300 with this command, FFFC:0 <CR>. Copy your new reset routine into memory starting at location $300.

Set the switch to ground (READEN) so that all future read accesses to $FC00-$FFFF will read the 1K RAM.

For example if these instructions are stored in memory starting at location $300, then when reset is pressed the system will clear the screen and then continue execution in the monitor (prompt='*').

```
$300:20 58 FC    JSR HOME (clears screen)
$303:4C 65 FF    JMP to MON (resume execution in monitor)
```

One of the problems with this circuit is that it also overrides any accesses to the language card. Therefore any program that uses the language card will not work with this circuit. The circuit doesn't keep track of which memory is enabled, ROM or language card RAM, in the $FC00-$FFFF space.

APPLE //e TECHNOTE #6

6-May 84

This article describes the paddle circuit used in the Apple // family of computers. The article starts with a simple description of the circuit used and then takes the reader through a thorough example of a typical paddle read routine. Finally, a few of the anomalies of the paddle circuit are discussed.

For further information contact:
PCS Developer Technical Support
M/S 22-W. Phone (408) 996-1010

PB

Peter Baum
20525 Mariani Ave.
Cupertino, Ca. 95014

Apple Computer
MS 22-W

May 6, 1984

A Treatise on the Apple Paddles/Joysticks

This article describes the paddle circuit used in the Apple // family of computers. The article starts with a simple description of the circuit used and then takes the reader through a thorough example of a typical paddle read routine. Finally, a few of the anomalies of the paddle circuit are discussed.

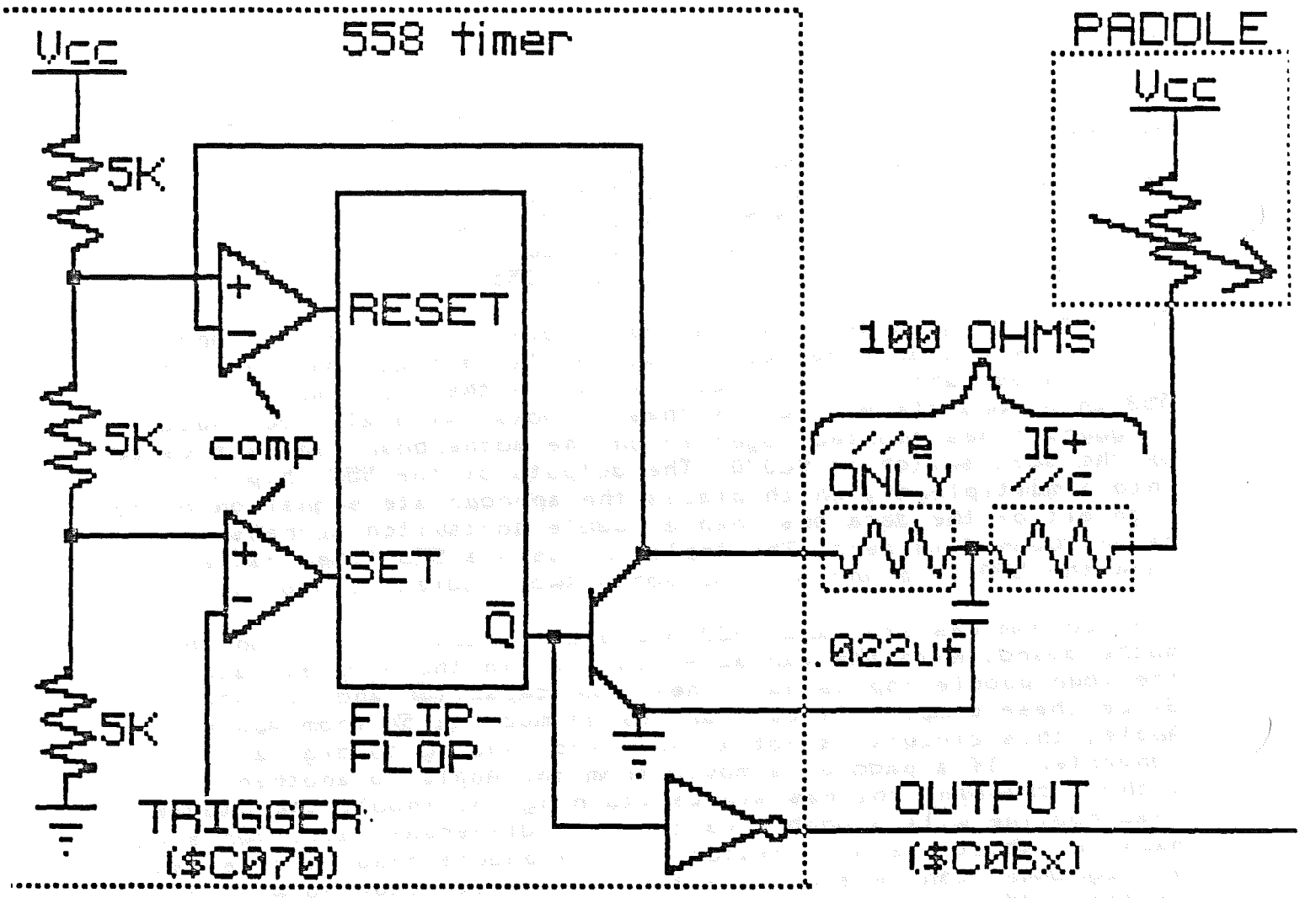
Circuit Description

The value of the Apple paddles (or joystick) is determined by a software timing loop reading a change of state in a timing circuit. The paddles consist of a variable resistor (from 0-150K ohms) which makes up part of the timing circuit. There is a routine in the Monitor ROM, called PREAD, which counts the time until a state change occurs in the paddle circuit. This time is translated into a value between 0 and 255.

The block diagram below shows the paddle circuit for the Apple][+, Apple //c and the Apple //e. The large block on the left illustrates part of the circuitry inside the 558 timer chip. The 558 chip consists of four of these blocks, with all four paddle triggers lines shorted together on the motherboard and activated by the soft switch at $C070. The outputs of the 558 chip run into a multiplexor, which places the appropriate signal onto the high bit of the data bus when a paddle softswitch address in the range $C064-7 is read. The Apple //c uses a 556 timer rather than the 558 chip and only supports two paddles, 0 and 1.

The 100 ohm resistor and .022 microfarad capacitor are on the motherboard, with the variable resistor in the paddle. Each of the four paddle inputs have their own capacitor and resistor. Since these components can vary by as much as 5% from Apple to Apple, this circuit is not a very exact analog to digital converter. If a paddle is moved from one Apple to another without changing the resistance (turning the knob), the paddle read routine will probably calculate a different value for each machine. About the only feature of the paddle read routine that a programmer can depend on is that the value returned will rise if the paddle resistance increases (or fall if the resistance decreases).

The paddle timing circuit on the Apple][+ and Apple //c is slightly different than the one on the Apple //e. On the Apple //e the 100 ohm fixed resistor is between the transistor and the capacitor, while the variable resistor in the paddle is connected directly to the capacitor. On the Apple][+ and //c the capacitor is connected directly to the transistor and the fixed resistor is in series with paddle resistor.



Paddle Circuit for Apple][+, //c and //e Showing 558 Timer

An Example of Typical Paddle Read Routine

The timing circuit works by discharging a capacitor through a transistor, then shutting the transistor off and letting the paddle charge the capacitor by supplying current through the variable resistor. The rate at which the capacitor charges is a function of the variable resistance; the lower the paddle resistance the greater the current and the faster the capacitor charges. When the capacitor reaches a predetermined value it changes the state of a flip-flop. The paddle read routine counts the time it takes for the capacitor to rise and change the flip-flop.

Let's step through an example of a typical paddle read operation. For now we'll assume the capacitor has already been discharged and in a few pages I'll explain when this assumption can be made and when it can't.

The software starts by reading the softswitch at location $C070, which strobes the trigger lines on the 558 timer. This causes two events to occur, the output signal (which is read at $C064-$C067 for paddle 0-3, respectively) goes high and the transistor turns off.

The software, after initially strobing the trigger line, executes a timing loop which reads the state of the output signal. When the output signal changes from high to low the the software jumps out of the timing loop and returns a value indicating the time. The monitor PREAD routine consists of a 11 usec. loop and will return a value between 0 and 255. (NOTE: The firmware listing is wrong and says the loop is 12 Usec.). The timing loop returns 255 if the circuit takes longer than 2.82 msec. for the state change to occur.

```
* PADDLE READ ROUTINE
* ENTER WITH PADDLE NUMBER (0-3) IN X-REG

FB1E:AD 70 C0   PREAD    4  LDA  PTRIG      ;TRIGGER PADDLES
FB21:A0 00               2  LDY  #0         ;INIT COUNTER
FB23:EA                  2  NOP             ;COMPENSATE FOR 1ST COUNT
FB24:EA                  2  NOP
FB25:BD 64 C0   PREAD2   4  LDA  PADDL0,X   ;COUNT EVERY 11 USEC.
FB28:10 04               2  BPL  RTS2D      ;BRANCH WHEN TIMED OUT
FB2A:C8                  2  INY             ;INCREMENT COUNTER
FB2B:D0 F8               3  BNE  PREAD2     ;CONTINUE COUNTING
FB2D:88                     DEY             ;COUNTER OVERFLOWED
FB2E:60                     RTS             ;RETURN W/VALUE 0-255
```

Inside the 558 timer chip, when the trigger is strobed low, the comparator that feeds the set input of the flip-flop is triggered, which in turn sets the output of the 558 timer. At the same time the transistor, which has held the capacitor near ground by sinking current from it, is shut off. The capacitor

can now charge up using the current supplied by the paddle. The smaller the paddle's resistance the more current the paddle will supply and the faster the capacitor charges. After some time, the capacitor will charge to the threshold value of 3.3 volts, which is set by the voltage divider network in the 558 timer, and the comparator that feeds the reset input on the flip-flop will trigger. This sets the output signal ($C06x) of the 558 timer low, which indicates to the software that the circuit has timed out.

```
                                                                  <=====THRESHOLD
TRIGGER $C070   __|__|_____________________________
                                  _.......-------|
FEEDBACK TO                     _.-'             |
RESET COMP     _____.-'                          \_________

                                _____________________
OUTPUT         _______|                          |___________

                        |<--TIMING VALUE-->|

                        0-2.82 MILLISECONDS
```

Resetting the flip-flop turns the transistor on, which discharges the capacitor very quickly (normally less than 250 ns). That paddle can then be read again.

A Closer Look at the Hardware

The First Anomaly

Notice that the last sentence states that the paddle can be read again and not the paddles. If another paddle is read immediately after the first, it may yield the wrong value. To show this I'll step through an example of reading a second paddle immediately after finishing the first.

In this example I'll assume that the first paddle has been set with a very low resistance, while the second paddle has a high resistance. The first paddle will time out very quickly and return with a small value, while the second paddle will take longer and yield a larger value.

We start reading the paddles by testing the paddle outputs to see if they're low, which indicates that the capacitor has been discharged. Assuming that the outputs are low, the next step is to trigger the 558 timer ($C070), which turns off the transistor and allows the capacitors to charge. Since all of the trigger input lines are shorted together all four of the capacitors will charge up, but at different rates since the paddle resistances have been set to different values. The voltage on the capacitor

for the first paddle will reach the threshold voltage very quickly since the paddle resistance has been set low, and therefore the timing loop will time out quickly.

At this point the capacitor for the second paddle is still charging and has not reached the threshold value yet, since the paddle resistance was set to a high value. The transistor for the second paddle is still turned off due to the initial trigger used for reading paddle one. This means that the capacitor for the second paddle has not been discharged.

Any attempts at reading the second paddle now will only yield false results. The capacitor is partly charged and therefore will reach the threshold value much faster than if the capacitor had been completely discharged. If the timing loop is used it will return with a smaller value than it would if the capacitor had been completely discharged. Notice that retriggering (reading location $C070) the 558 timer will not help, since that only keeps the transistor turned off and doesn't help discharge the capacitor. The only way for the capacitor to discharge is to let the circuit timeout completely by letting the capacitor charge until it resets the flip-flop.

To read the second paddle the capacitor must first be discharged, which is only done when the threshold value is reached and the 558 timer flip-flop is reset. The only way to guarantee that the capacitor is discharged is if the transistor is on. This condition is met when the paddle output is low. Therefore start every paddle read either by waiting for at least 3 ms. before strobing the trigger input or testing to make sure that the paddle output is low.

If after 4 ms. the paddle output is not low then there is a good chance that there is no paddle connected. It may also indicate that a peripheral with a larger maximum value resistor than the 150K ohms used by the Apple paddles is attached. Some peripheral devices use this technique of a larger variable resistor so that more than 256 points of resolution can be determined. Of course this requires a custom software driver and the Monitor PREAD routine can't be used.

The Apple //e Anomaly

The problem with Apple //e paddle input is that the capacitor may not be discharged by the transistor. Typically, the transistor will discharge the capacitor in less than 250 ns. on the Apple II+. But on the Apple //e if the paddle resistance is very low then the paddle may supply enough current to always keep the capacitor charged.

Because the fixed resistor (100 ohms) on the Apple //e motherboard is between the capacitor and the transistor, there will be a voltage drop across the resistor if the capacitor stays charged. When the transistor is shut off by the trigger

strobe, this voltage drop will disappear and the capacitor, which may be near the threshold voltage, will trigger the reset comparator earlier than it would if the capacitor had been discharged completely. The net affect of this is that the paddles will read zero on the Apple //e when they would read a small value on the Apple][+ or //c.

Other circuits which expect the capacitor to discharge completely may not work properly. A circuit which attempts to simulate a paddle through active components such as a digital to analog converter may be able to source enough current that the capacitor never discharges and the paddle always reads zero.

Hopefully, this article has given the reader a good feel for the paddle circuitry and the routines which determine the paddle values. To reinforce the material covered you should try writing your own paddle read routine. For example, you could write a read routine that would read two paddles at once. The software loop will not have the 11 usec. resolution of the PREAD routine, but you'll find it stills works just fine. Happy programming!

APPLE //e TECHNOTE #7

3-April 84

This article describes three different types of interfaces, serial, parallel, and IEEE-488, that are currently used to connect a printing device to an Apple //. The interface cards available from Apple and the protocol to connect to an Apple printer are briefly described. Pin out configuration and switch settings for these interfaces cards and printers is also included.

For further information contact:
PCS Developer Technical Support
M/S 22-W. Phone (408) 996-1010

PB

Peter Baum
Apple Computer
20525 Mariani Ave.
Cupertino, Ca. 95014

Apr. 3, 1984

Connecting a Printer to an Apple //

This article describes three different types of interfaces, serial, parallel, and IEEE-488, that are currently used to connect a printing device to an Apple //. The interface cards available from Apple and the protocol to connect to an Apple printer are briefly described. Pin out configuration and switch settings for these interfaces cards and printers is also included.

Serial

Currently, Apple sells a card, called the Super Serial Card (SSC), that can be used to connect an Apple printer to an Apple // (Apple sells a dot-matrix printer, called the Imagewriter, a daisy-wheel printer, and a plotter, which all use a serial interface). The SSC replaces both the Communications card and the Hi-speed Serial card. The SSC supports the firmware (Pascal 1.1) protocol except for the optional control and interrupt handling routines. For more information on the firmware protocols see Appendix A of the Super Serial Card manual (Part # A2L0044).

If the SSC is to be connected to an Imagewriter printer then the switches should be set as follows:

```
                SSC                                        Imagewriter
_____________________________________                    _______________
|     1234567 1234567  modem      |                    |   87654321  |
|ON     ' '' '  ''       v        |DB-25      DB-25|ON    ''       |
|OFF ,,, ,     ,,  ,, terminal    |================|OFF ,,  ,,,,   |
|      SW1      SW2               |   (590-0037)   |      SW1    |
|                                 |                |             |
|_________________                |                |     4321    |
                   |____________|                | ON     ''   |
                                                  | OFF  ,,     |
' = switch is in up (closed/on) position          |      SW2    |
, = switch is in down (open/off) position         |_____________|
v = The jumper block should point toward terminal
```

Note: Switch 1-5 on the printer must be in the on (up) position for the printer to work with the Apple //e.

These switch settings set the serial interface to use 9600 baud printer mode with 8 data bits, one stop bit, no delay after carriage return (CR), 80 column line width, no echo to screen, and automatic line feed sent after CR.

The Apple part number for a cable that connects the SSC to an Imagewriter is 590-0037. This cable consists of two male DB-25 connectors with pins 1-8, 12, 13, 19, 20, 23 wired pin to pin and shielded.

The SSC has a 10-pin header on it, but comes with a cable which connects the header to a female DB-25 connector. The DB-25 can be configured as either a modem (DCE) or as a terminal (DTE) using a jumper block (in the latter case the jumper block acts as a modem eliminator). Though the pin out configuration of the DB-25 connector is well defined, there is no standard use of the handshake signals. Different printers will use the handshake lines for different functions. The following table shows the pinout for the DB-25 on the SSC. Consult the printer manual for more specific information on which signals are used.

10-pin Header	Signal Name		Female DB-25 pinout Terminal	Modem	notes
1	Frame Ground	(FRMGND)	1	1	
2	Transmit Data	(TxD)	3	2	
3	Receive Data	(RxD)	2	3	
4	Request To Send	(RTS)	8	4	
5	Clear To Send	(CTS)	8	5	
6	Data Set Ready	(DSR)	20	6	
8	Signal Ground	(SGLGND)	7	7	
10	Data Carrier Detect	(DCD)	4,5	8	*1
7	Secondary Clear to Send	(SCTS)	19	19	*2
9	Data Terminal Ready	(DTR)	6	20	

notes:

*1 - only if SW1-7 is closed (on) with SSC
*2 - only if SW2-7 is closed (on) with SSC

```
 1 |\
   | ) 14
   | |
   | |     DB-25
   | |
   | ) 25
13 |/
```

To illustrate an example of a serial interface, I'll use the Imagewriter printer. Here is the pinout and interface specification:

Pin no.	Symbol	Descripition	Direction
1	FG	Frame Ground	
2	TxD	Send Data	Output
3	RxD	Receive Data	Input
4	RTS	Request to Send	Output
7	SG	Signal Ground	
14	FAULT	Fault	Output
20	DTR	Data Terminal Ready	Output

Functional Description:

FG = Grounding line for circuit protection
TxD = Serial transmission line from printer to computer
RxD = Serial transmission line from computer to printer
RTS = True when printer is turned on
Fault = False when printer deselected; true when selected
DTR = True if printer on and ready to receive

The printer uses a hardware handshaking scheme, called the Data Transfer Ready protocol, to receive data. Whenever the capacity of the input buffer is less than 30 characters, the printer sends a busy signal by setting the DTR line false. The computer must stop transmission within the next 27 characters or the printer will ignore the excess data. The DTR line is also set false when the printer is deselected, and when it receives a DC3 character. The DTR line is true whenever there is room for at least 100 characters in the input buffer, when the printer is turned on, selected, and has received a DC1 character.

Parallel

Apple currently ships a parallel card, called appropriately the Parallel Interface Card (PIC), which can be used to connect a parallel printer to an Apple // (Apple used to sell a dot-matrix printer called the DMP, but has discontinued shipping any printers with a parallel interface). The PIC replaces the Parallel Printer Interface Card and the Centronics Interface Card. The PIC doesn't support the firmware protocol, so Pascal identifies the card as a printer card (described in Pascal protocols).

Most commonly used printers will operate properly if the switches on the PIC are set as follows:

```
 _____________________
|        1234567      |
|  ON      ''         |
| OFF   ,,,   ,,      |
|_____________________|
```

This sets the parallel interface to transfer data using a 1 microsecond strobe pulse of negative polarity when sending data, while receiving a negative acknowledge signal, with interrupts disabled.

The PIC has a 26-pin header, but comes with a cable which connects the header to a female DB-25. The Parallel Printer Card and the Centronics Card used a 20-pin header. Most parallel printers (90%) use a 'microribbon 36' as the connector. The pinout varies from printer to printer, but the following table covers most printers (Apple DMP, Epson). For other printers refer to page 7 of the Parallel Interface Card manual (Part # A2L0045).

PIC Function	Printer Function	26-pin header	DB-25 conn.	36-pin microribbon	20-pin header
Ground	Ground	3	2	19	1
Ground	Ground	22	24	16	20
Ground	Ground	7	4		
Ground	Ground	14	20		
ACK	Acknowledge	6	16	10	2
Strobe	Strobe	4	15	1	8
DO 0	Data 1	9	5	2	10
DO 1	Data 2	11	6	3	11
DO 2	Data 3	15	8	4	12
DO 3	Data 4	18	22	5	13
DO 4	Data 5	20	23	6	14
DO 5	Data 6	21	11	7	15
DO 6	Data 7	23	12	8	16
DO 7	Data 8 (#2)	25	13	9	17
DI 3	Fault	24	25	32	6
DI 4	Busy	2	14	11	7
DI 5	Paper out	12	19	12	9
DI 6	Select	16	21	13	18
DI 7	Enable (#1)	10	18 7	35	19
Apple internal part # for cable		590-0049B		590-0042B	

(#1) - Pin 7 is blocked on the female DB-25 connector and omitted on the male DB-25 connector to prevent the insertion of serial connectors into parallel ports.

(#2) - This may be assigned a 'hard' value for some printers to distinguish between graphics and normal character sets.

<u>Functional Description of Signal for Typical Printer</u>

Strobe = Printer clocks data in on falling edge
ACK = Set low by printer to indicate it has processed last character and is ready for another
Fault = Set low if printer detects fault condition
Busy = Set high by printer to indicate not ready
Paper out= Used by printer to indicate out of paper
Select = Output from printer, set high if printer selected
Enable = Set high by printer to indicate printer active

Since the PIC can also be used to input parallel data and doesn't act as only a printer card, it is no longer referred to as a printer card, but instead as a general purpose parallel card.

IEEE-488

Though most printing instruments on the market today use either a serial or parallel interface, another standard interface, IEEE-488, is also available. These devices can be connected to the Apple // through the Apple IEEE-488 Interface Card. Currently Apple doesn't sell any printer devices that use the IEEE-488 interface, but other companies supply them. One of the advantages of the IEEE-488 bus over either the parallel or serial (RS-232) busses is that more than one type of printer can be attached to the bus at the same time. This means that both a fast dot-matrix and a daisy wheel printer can be hooked to the Apple with only one peripheral card.

The IEEE-488 bus standard is a well defined 8-bit parallel, byte serial, asynchronous data transfer interface. The standard has been thoroughly documented with the most complete description available from the Institute of Electrical and Electronic Engineers (IEEE) in New York. Standard cables are manufactured by many companies, and usually advertised as either IEEE-488, General Purpose Interface Bus (GPIB), or Hewlett-Packard Interface Bus (HPIB) cables.

The IEEE-488 card doesn't support the firmware protocols, so an assembly language driver must be used to access the card from Pascal (See Appendix F of the IEEE-488 Interface User's Guide, part number A2L0037).

Appendix A

Product	Order Part #
Super Serial Card	A2B0044
SSC to Imagewriter Accessory Kit *	A2C0352
SSC to Imagewriter external Cable	590-0037
Imagewriter	A9M0303
Apple Daisy Wheel Printer (DWP)	A3M0025
SSC to Apple DWP Accessory Kit *	A2C0351
Apple Color Plotter	A9M0302
SSC to Color Plotter Accessory Kit *	A2C0302
Parallel Card	A2B0021
IEEE-488 Interface Card	A2B0015
SSC manual	A2L0044
Parallel Interface Card manual	A2L0045
ProDOS Technical Reference Manual	A2W0010
Apple //e Reference Manual	A2L2005
Apple //e Design Guidelines	A2F2116

* The accessory kit includes a cable and manuals

Apple][Monitor Entry Points

2 August 1984

This document lists all supported entry points in the Apple][family $F800 Monitor ROM. This is NOT a programming guide. Since each member of the Apple][family has variations in the implementation of the Monitor, it is the individual programmer's responsibility to identify the machine type and take appropriate action when calling these routines. The only purpose of this document is to reassure software developers that the entry points for these routines will remain intact and that there is no committment to keep any other Monitor code in the same locations.

Disclaimer of all Warranties and Liabilities

Notice

Apple Computer, Inc. reserves the right to make improvements in the product described in this document at any time and without notice.

CJS

$F800 PLOT Plot on the low-resolution screen

PLOT puts a single block of the color value set by SETCOL on the low-resolution display screen. The block's vertical position is passed in the accumulator, its horizontal position in the Y register. PLOT returns with the accumulator scrambled, but X and Y intact.

$F819 HLINE Draw a horizontal line of blocks

HLINE draws a horizontal line of blocks of the color set by SETCOL on the low-resolution graphics display. Call HLINE with the vertical coordinate of the line in the accumulator, the leftmost horizontal coordinate in the Y register, and the rightmost horizontal coordinate in location $2C. HLINE returns with A and Y scrambled, X intact.

$F828 VLINE Draw a vertical line of blocks

VLINE draws a verticle line of blocks of the color set by SETCOL on the low-resolution display. You should call VLINE with the horizontal coordinate of the line in the Y register, the top vertical coordinate in the accumulator, and the bottom vertical coordinate in location $2D. VLINE will return with the accumulator scrambled.

$F832 CLRSCR Clear the low-resolution screen

CLRSCR clears the low-resolution graphics display to black. If CLRSCR is called while the video display is in text mode, it fills the screen with inverse at-sign (@) characters. CLRSCR destroys the contents of A and Y.

$F836 CLRTOP Clear the low-resolution screen

CLRTOP is the same as CLRSCR, except that it clears only the top 40 rows of the low-resolution display. (Mixed-mode)

$F847 GBASCALC Calculate base address for low resolution graphics

GBASCALC calculates the base address of the line on which a particular pixel is to be plotted. The accumulator is scrambled.

$F85F NXTCOL Increment color by 3

NXTCOL adds 3 to the current color (set by SETCOL) used for low-resolution graphics. The accumulator is scrambled.

$F864 SETCOL Set low-resolution graphics color

SETCOL sets the color used for plotting in low-resolution graphics to the value passed in the low nybble of accumulator. The colors and their values are listed in the technical reference manual. The accumulator is scrambled.

$F871 SCRN Read the low-resolution graphics screen

SCRN returns the color value of a single block on the low resolution graphics display. Call it with the vertical position of the block in the accumulator and horizontal position in the Y register. Call it as you would call PLOT (above). The color of the block will be returned in accumulator. No other registers are changed.

$F88E INSDS2 Set-up indexes for opcode in A register

INSDS2 expects to find the opcode in the accumulator. It then sets up formats, modes, and indexes into the mnemonic table. Upon entry, the X register must be zero. Upon exit the accumulator and X register are scrambled.

$F8D0 INSTDSP Display disassembled instruction

INSTDSP disassembles and displays one instruction pointed to by the program counter (PCL-PCH). None of the registers are preserved.

$F940 PRNTYX Print contents of Y and X registers as hex

PRNTYX prints the contents of the Y and X registers as a four-digit hexadecimal value. The Y register contains the first byte output, the X register contains the second. On return, the contents of the accumulator are scrambled.

$F941 PRNTAX Print A and X in hexadecimal

PRNTAX prints the contents of the A and X registers as a four-digit hexadecimal value. The accumulator contains the first byte output, the X register contains the second. On return, the contents of the accumulator are scrambled.

$F944 PRNTX Print contents of X register as hex

PRNTX prints the contents of the X register as a two digit hexadecimal value. On return, the contents of the accumulator are scrambled.

$F948 PRBLNK Print 3 spaces

PRBLNK outputs three blank spaces to the standard output device. On return, the accumulator usually contains $A0, the X register contains 0.

$F94A PRBL2 Print many blank spaces

PRBL2 outputs from 1 to 256 blanks to the standard output device. Upon entry, the X register should contain the number of blanks to be output. If X=$00, then PRBL2 will output 256 blanks. On return, the accumulator usually contains $A0, the X register contains 0.

$F953 PCADJ Adjust program counter

PCADJ increments the program counter by 1, 2, or 3 depending on the LENGTH byte stored at $2F, 0 = 1 byte, 1 = 2 bytes, 2 = 3 bytes. All registers are scrambled.

$FA40 IRQ IRO handler

IRQ first determines if the interrupt request was from a BRK instruction. If not, control is sent to IRQLOC ($3FE). The accumulator is stored (at $45 with the][,][+, and original //e monitors and pushed on the stack with the "ICON" //e, and //c monitors). When the $03FE interrupt handler terminates with an RTI, all registers are restored. (Generally called by operating system, not user.)

$FA4C BREAK BRK handler
BREAK saves the registers and JMPs to BRKV ($3F0).

$FA62 RESET Hardware reset handler
RESET sets normal video out and keyboard in, re-initialize system, set and clear various annunciators (depending on system type), clear keyboard, and falls through to NEWMON.

$FAA6 PWRUP System cold start
PWRUP prints system type at top of screen, sets page 3 vectors equal to cold start of current BASIC. It then falls through to SLOOP.

$FABA SLOOP Disk controller slot search loop
SLOOP is the disk controller search loop. It searches for a disk controller beginning at the peripheral ROM space pointed to by $00-$01. If a disk controller is found, it JMPs to the ROM code. Otherwise, it cold-starts BASIC. (Required to support the ProFile card boot code.)

$FAD7 REGDSP Display contents of registers
REGDSP sets location A3 ($40-$41) equal to $0045, then displays the contents of the registers (from locations $45 thru $49) with labels. (Setting A3 prepares the user for modifying memory beginning at $45.) The accumulator and X register are not preserved.

$FB19 RTBL Register names table
RTBL contains the ASCII codes for "AXYPS" (hi-bit set), the names of the registers.

$FB1E PREAD Read a hand controller
PREAD returns a number that represents the position of a hand control. You pass the number of the hand control in the X register. If this number is not valid (not equal to 0, 1, 2, or 3), stange things may happen. PREAD returns with a number from $00 to $FF in the Y register. The accumulator is scrambled.

$FB2F INIT Initialize system
Clears $48, the 6502 status register save locations, and sets softswitches to LO-RES, PAGE 1, TEXT, then falls through to SETTXT.

$FB39 SETTXT Set text mode
SETTXT sets text mode and LDA #0 to set window top then JMPs to SETWND.

$FB40 SETGR Set graphics mode
SETGR sets mixed graphics mode and clears the graphics portion of the screen then LDA #20 to set window top and falls through to SETWND.

$FB4B SETWND Set text window
SETWND sets a full width text window with the window top set to the value in the accumulator and bottom set to the bottom of the screen. It then VTABs to line 23.

$FB5B TABV Vertical tab
TABV merely stores the value in the accumulator in location CV ($25) and calls VTAB ($FC22).

$FB60 APPLEII Print machine type
APPLEII clears the screen and prints the machine type centered at the top of the screen. A and Y are scrambled.

$FB6F SETPWRC Create power-up byte
SETPWRC calculates the "funny" complement of the high byte of the RESET vector and stores it at PWREDUP ($3F5).

$FB78 VIDWAIT Check for a pause (CONTROL-S)
VIDWAIT checks the keyboard for a CONTROL-S if it is called with an $8D in the accumulator. If a CONTROL-S is found, it falls through to KBDWAIT. If not, control is sent on to VIDOUT where the character is printed and the cursor advanced.

$FB88 KBDWAIT Wait for keypress
KBDWAIT waits for a keypress. The keyboard is cleared unless the keypress is a control-C then control is sent on to VIDOUT where the character is printer and the cursor advanced.

$FBB3 VERSION Monitor ROM identification byte
VERSION is a byte used to aid in identifying which monitor ROM is installed.

$FBC0 ZIDBYTE Monitor ROM sub-identification byte
This byte provides more detailed identification of the monitor ROM.

$FBC1 BASCALC Text base address calculator
BASCALC calculates the base address of the line for the next text character on the forty column screen. The value is stored at BASH and BASL ($28-$29).

$FBDD BELL1 Beep the speaker
BELL1 toggles the speaker on and off at 1000 hz rate for 0.1 sec.

$FBF0 STORADV Place a printable character on the screen
STORADV stores the value in the accumulator at the next position in the text buffer and falls through to ADVANCE.

$FBF4 ADVANCE Increment the cursor position
ADVANCE advances the cursor by one position. If the cursor is at the window limit it branches to CR.

$FBFD VIDOUT Place a character on the screen
VIDOUT sends printable characters to STORADV. Return, linefeed, forward and reverse space, etc., are vectored to appropriate special routines. (NOTE: This routine does not work in 80-columns on][,][+, and original //e.)

$FC10 BS Back-space

BS decrements the cursor one position. If the cursor is at the beginning of the window, the horizontal cursor position is set to the right edge of the window and the routine falls through to UP. (NOTE: 40-columns only.)

$FC1A UP Move up a line

UP decrements the cursor vertical location by one line unless the cursor is currently on the first line. (NOTE: 40-columns only.)

$FC22 VTAB Vertical tab

VTAB loads the value at CV ($25) into the accumulator and falls through to VTABZ. (NOTE: This routine does not update OURCV in 80-columns.)

$FC24 VTABZ Vertical tab (alternate entry)

VTABZ uses the value in the accumulator to update the base address used for storing values in the text screen buffer.

$FC42 CLREOP Clear to end of page

CLREOP clears the text window from the cursor position to the bottom of the window. CLREOP destroys the contents of A and Y.

$FC58 HOME Home cursor and clear

HOME clears the current window and places the cursor in the home position: the upper left corner of the screen.

$FC62 CR Begin a new line

CR sets the cursor horizontal position back to the left edge of the window and increments the cursor vertical position. It then falls through to LF. (NOTE: 40-columns only.)

$FC66 LF Line-feed

If the cursor vertical position is not past the bottom line, the base address is updated, otherwise the routine falls through to SCROLL. (NOTE: 40-columns only.)

$FC70 SCROLL Scroll the screen

SCROLL moves all characters up one position within the current text window.

$FC9C CLREOL Clear to end of line

CLREOL clears a text line from the cursor position to the right edge of the window. CLREOL destroys the contents of A and Y.

$FC9E CLEOLZ Clear to end of line

CLEOLZ clears a text line to the right of the window, starting at the location given by base address BASL indexed by the contents of the Y register. CLEOLZ destroys the contents of A and Y.

$FCA8 WAIT Delay

WAIT delays for a specific amount of time, then returns to the program that called it. The amount of delay is specified by the contents of the accumulator. With A the contents of the accumulator, the delay is 1/2(26+27A+5A^2) microseconds. WAIT returns with the accumulator zeroed and the X and Y registers undisturbed.

$FCB4 NXTA4 Increment pointer A4

NXTA4 increments the 16 bit pointer, A4 ($42-$43) and then falls through to NXTA1.

$FCBA NXTA1 Compare A1 with A2 and increment A1

NXTA1 does a 16 bit compare of A1 ($3C-$3D) with A2 ($3E-$3F) and then increments pointer A1.

$FCC9 HEADR Write a header to cassette tape (][,][+, //e only)

HEADR writes a header to cassette tape.

$FD0C RDKEY Get an input character

RDKEY is the character input subroutine. It places an appropriate cursor on the display at the cursor position and jumps to the subroutine whose address is stored in KSW (locations $38-$39), usually the standard input subroutine KEYIN, which returns with a character in the accumulator.

$FD1B KEYIN Read the keyboard

KEYIN is the keyboard input subroutine. It reads the Apple's keyboard, waits for a keypress, and randomizes the random number seed at $4E-$4F. When a key is pressed, KEYIN removed the cursor from the display and returns with the keycode in the accumulator. (NOTE: On //e with 80-columns, it interprets escape codes and forward arrows.)

$FD35 RDCHAR Get an input character or ESC code

RDCHAR is an alternate input subroutine that gets characters from the standard input subroutine, and also intreprets the escape codes listed in the technical reference manual.

$FD67 GETLNZ Get an input line

GETLNZ is an alternate entry point for GETLN that sends a carriage return to the standard output, then continues into GETLN.

$FD6A GETLN Get an input line with prompt

GETLN is the standard input subroutine for entire lines of characters, as described in the various technical reference manuals. Your program calls GETLN with the prompt character in location $33; GETLN then falls through to GETLNO.

$FD6C GETLNO Get an input line with prompt (alternate entry)

GETLNO outputs the contents of the accumulator as the prompt. The routine will return with the input line in the input buffer ($200-$2FF) and the X register holding the length of the input line. If the user cancels the input line, either with too many backspaces or a CONTROL-X, then the contents of PROMPT ($33) will be issued as the prompt when it gets another line.

$FD6F GETLN1 Get an input line, no prompt

GETLN1 is an alternate entry point for GETLN that does not issue a prompt before it accepts the input line. If, however, the input line is cancelled, with too long a line, with too many backspaces or with a CONTROL-X, then GETLN1 will issue the contents of PROMPT ($33) as a prompt when it gets another line.

$FD8B CROUT1 RETURN with clear to end-of-line

CROUT1 clears the screen from the current cursor position to the edge of the text window, then falls through to CROUT.

$FD8E CROUT Carriage return output

CROUT sends a RETURN character to the current output device.

$FD92 PRA1 Print RETURN and A1 in HEX

PRA1 sends out a RETURN character followed by the contents of the 16 bit pointer, A1 ($3C-$3D) in hex follwed by a hyphen.

$FDDA PRBYTE Print a hexadecimal byte

PRBYTE outputs the contents of the accumulator in hexadecimal on the current output device. The contents of the accumulator are scrambled.

$FDE3 PRHEX Print a hexadecimal digit

PRHEX prints the lower nybble of the accumulator as a single hexadecimal value. On return, the contents of the accumulator are scrambled.

$FDED COUT Output a character

COUT calls the current output subroutine. The character to be output should be in the accumulator. COUT calls the subroutine whose address is stored in CSW (locations $36 and $37), which is usually the standard character output COUT1.

$FDF0 COUT1 Output to screen

COUT1 displays the character in the accumulator on the Apple's screen at the current output cursor position and advances the output cursor. It places the character using the setting of the Normal/Inverse location. It handles the control charcters [RETURN], linefeed, backspace, and bell. It returns with all registers intact.

$FE2C MOVE Move a block of memory

MOVE copies the contents of memory from one range of locations to another. This subroutine is the same as the MOVE commands in the Monitor, except it takes its arguments from pairs of locations in memory, low-byte first. The destination address must be in A4 ($42-$43), the starting source address in A1 ($3C-$3D), and the ending source address in A2 ($3E-$3F) when your program calls MOVE.

$FE5E LIST Disassemble and list 20 instructions
LIST will disassemble and list to the current output device, 20 assembly language instructions beginning at the location pointed to by A1 ($3C-$3D).

$FE80 SETINV Set inverse text mode
SETINV sets INVFLG so that subsequent text output to the screen will appear in inverse mode.

$FE84 SETNORM Set normal text mode
SETNORM sets INVFLG such that subsequent text output to the screen will appear in normal mode.

$FE89 SETKBD Re-set input to keyboard
SETKBD re-sets the the input hooks ($38-$39) to point to the Keyboard.

$FE8B INPORT Re-set input to a slot
INPORT re-sets the input hooks ($38-$39) to point to the ROM space reserved for a perphireal card (or port) in the slot (or port) designated by the value in the accumulator. (NOTE: In new //e and //c monitor, does a quit if the video firmware was on.)

$FE93 SETVID Re-set output to screen
SETVID re-sets the output hooks ($36-$37) to the screen display routines.

$FE95 OUTPORT Re-set output to a slot
OUTPORT re-sets the output hooks ($36-$37) to point to the ROM space reserved for a peripheral card (or port) in the slot (or port) designated by the value in the accumulator.

$FEB6 GO Begin code execution
GO begins execution of the code pointed to by A1 ($3C-$3D).

$FECD WRITE Write a record on a cassette tape (][,][+, and //e only)
WRITE converts the data in a range of memory to a series of tones at the cassette output port. Before calling WRITE, the address of the first data byte must be in A1 ($3C-$3D) and the address of the last byte in A2 ($3E-$3F). The subroutine writes a ten-second continuous tone as a header, then writes the data followed by a one byte checksum.

$FEFD READ Read data from a cassette tape (][,][+, and //e only)
READ reads a series of tones at the cassette input port, converts them to bytes, and stores the data in a specified range of memory locations. Before calling READ, the address of the first byte must be in A1 ($3C-$3D) and the address of the last byte in A2 ($3E-$3F).

$FF2D PRERR Print ERR
PRERR sends the word ERR, and falls through to BELL. On return, the accumulator contains $87.

$FF3A BELL Output a bell character
BFLL writes a bell [CONTROL]-G character to the current output device. It leaves the accumulator holding $87.

$FF3F RESTORE Restore all registers
RESTORE loads the 6502's internal registers with the contents of memory locations $45 through $48, as saved by BREAK.

$FF4A SAVE Save all registers
SAVE stores the contents of the 6502's internal registers in locations $45 through $49 in the order A, X, Y, P, S. The contents of A and X are changed and the decimal mode is cleared.

$FF58 = $60 Known RTS instruction (IORTS)
This byte must always contain $60.

$FF65 MON Standard Monitor entry with beep
MON beeps the speaker and falls through to MONZ.

$FF69 MONZ Standard Monitor entry point (CALL -151)
MONZ displays the "*" prompt and sends control to GETLNZ.

$FF8A DIG Shift hex digit into A2
DIG shifts an ASCII representation of a hex digit in the accumulator into A2 ($3E-$3F).

$FFA7 GETNUM Transfer hex input into A2
GETNUM scans input buffer starting at position Y. Shifts hex digits into A2 ($3E-$3F). Stops when non-hex digit encountered.

$FFAD NXTCHR Translate next character
NXTCHR is the loop used by GETNUM to parse each character in the input buffer and convert it to a value in A2 ($3E-$3F).

$FFFA NMI Non-maskable interrupt vector
NMI is a two byte pointer to the non-maskable interrupt handler.

$FFFC RESET Reset vector
RESET is a two byte pointer to the RESET handler routine.

$FFFE IRQVECT IRQ vector
IRQVECT is a two byte pointer to the interrupt request handler.

```
2350: 4C 33
2352<2350.2360M
2300<2350.2360^Y

230D: 33 4C 33 4C 33 4C 33 4C      (puts green line next to it)
235D: 13 66 19 66 19 66 19 66      (note first byte)
230D<235D.2363^Y
```

There you have it: a basic explanation of how double hi-res works -- except for one or two anomalies. The first anomaly is that NTSC monitors have a limited display range. The second anomaly shows one of the features of double hi-res versus a limitation of standard hi-res.

An NTSC color monitor decides what color to display based on its "view" of four bit "windows" in each line, starting from the left edge of the screen. The monitor looks at the first four bits, determines which color is called for, and then shifts one bit to the right and determines the color for this new four-bit window. But remember the color depends not only on the pattern, but also the positon of the pattern. To compensate for relative position from the left edge of the screen, the monitor keeps track of where on each line each of these window starts. (For those of you of the technical persuasion, this is done through the use of the color burst signal, which is a 3.58 MHz. clock).

Try this example:

```
2000:0                    Clears screen
2001<2000.3FFFM
2000<2000.3FFF^Y

2001:66                   Draws orange box in uppper left
2401:66
2801:66
2C01:66
3001:66


2050:33                   Draws blue box below and
3402<2050.2050^Y              to the right of the orange
3802<2050.2050^Y
3C02<2050.2050^Y
```

Notice that if the blue box was drawn at the top of the screen, next to the orange box, they would overlap. Yet, the boxes were drawn on two different columns, orange on mb2 and blue on aux1. This can be explained by the previous paragraph, and the sliding windows. The monitor will detect the pattern for orange slightly after the mb2 column, while the pattern for blue shows up before column aux1.

orange pattern:

```
00000001011001110000000        look at four-bit windows and you'll see
  aux2 |  mb2  |  aux1         an orange pattern overlaps on both sides
```

If a pattern is repeated on a line, this overlap doesn't cause a problem, since the same color just overlaps itself. But watch what happens when a new pattern is started next to a different pattern:

```
3002<2050.2050^Y          Puts blue pattern next to orange
2C02<2050.2050^Y
2802<2050.2050^Y
```

Where the blue overlaps the orange, you'll see a white dot. This is because one of the four-bit windows the monitor sees is all 1's. If two colors are placed right next to each other, the monitor will sometimes display a third color, or fringe, right at the boundary. "Fringing" is especially noticeable when there are a lot of narrow columns of different colors next to each other. (Next time you run COLOR TEST take a look at the boundaries between the colors).

```
        orange    blue

00000001011001111001100        note the four 1's in a row
  aux2 |  mb2  |  aux1         at the boundary between
                                orange and blue
```

THE DOUBLE HI-RES ROUTINES

The second anomaly presents a good lead in to the last part of this series, the double hi-res routines, which plot lines. These routines work like the standard hi-res Applesoft commands, HGR, HCOLOR , and HPLOT, except they use the Applesoft ampersand function.

[[At this point BRUN COLOR DBL HIRES]]

There are four ampersand functions:

&H	Clears double hi-res screen
&Cn	Sets the double hires color to n
&Px,y	Plots a point at x,y
&Lx,y	Draws a line from the last point to x,y

TEXT	Returns to 40-column text mode
POKE 49164,0	
POKE 49247,0	

Apple //c Technical Note #1

Revision of Apple //c Technical Note #1, 8 February 1984*
25 February 1984

There are differences between how the mouse works on the Apple //e and how it works on the Apple //c. This technical note explains what is causing these differences and how to write programs that work the same on both machines.

For further information contact:
PCS Developer Technical Support
M/S 22-W, Phone (408) 996-1010

* A clarification of the effects disabling interrupts has on mouse data has been added

BG

INTRODUCTION

As advertised, if you use the mouse firmware routines such as SETMOUSE to control the mouse then these routines will perform the same function in the Apple //c as they do in the Apple //e. This does not mean that a program which uses the mouse will behave the same in both computers. There are two reasons for this. One is that if a program has not properly set the environment prior to calling these routines it is possible for the program to work in one machine and not in the other. The second reason is that there are differences in the machines and although the ROM rouines perform the same functions there may be noticable differences in the 'behaviour' of the mouse. This technical note will explain the fundamental differences between the way the mice in the two machines work. It will then point out precautions that need to be taken to make sure that your machine language program will work on both machines. With the exception of mouse movement scaling described below BASIC and Pascal programs do not need to be concerned about setting the proper environment.

The Apple //e mouse card has a microprocessor on it which constantly polls the mouse to get status and position information. This data is then kept on the card and is available whenever the program requests it through the READMOUSE routine. If the mouse is in passive mode this information will be 'picked up' by the main program whenever it gets around to it. The SETMOUSE routine can set the mouse card to issue interrupts under certain conditions. When the mouse card determines that such conditions exist it issues an interrupt. This stops the main computer and goes to what ever interrupt handling routine has been set up. This routines will then read the information from where the card processor saved it and puts it in the screen holes. When using a mouse on an Apple with a mouse card your program is only interrupted if your program has requested it. And the data in the screen holes is being changed only when the program's interrupt handler or polling routine has called READMOUSE. Also enabling and inhibiting interrupts does not affect the updating of mouse information by the card's microprocessor.

The Apple //c mouse does not have a card microprocessor and so mouse information is collected by interrupting the Apple //c's microprocessor. When the interrupt happens the firmware captures it and processes it which includes updating the screen holes. The interrupt is passed on only if SETMOUSF set up the conditions to do so. However, having the mouse interrupt the computer's microprocessor means that your program is being constantly interrupted. This will affect program timming. It also means that the screen holes are constantly being updated with X and Y information even in passive mode since this information must be kept somewhere and there is no card to keep it on. Also, if you have disabled interrupts then the mouse can never interrupt the processor and so the X adn Y values are never updated and calling READMOUSE will indicate that there has been no mouse movement.

Since the Apple //c is constantly being interrupted while the mouse is on, the program's performance may be affected. To minimize this affect the Apple //c responds one-half as frequently to mouse movements as does the mouse card. The noticable result of this is that the mouse must be moved twice as far to create the same effect. If you want the same behaviour on both machines then multiply the Apple //c X and Y values by two and clamping to 1/2 the //e value before using them.

With the exception of having to double the Apple //c mouse movement your program can ignore which machine it is running on by following the precautions listed below. If you are working from BASIC or Pascal these conditions are taken care of for you.

THE FOLLOWING CONDITIONS MUST BE TAKEN INTO ACCOUNT BY MACHINE LANGUAGE PROGRAMMERS IF THE PROGRAM IS TOP RUN SIMILARLY IN ALL THF APPLE // FAMILY OF COMPUTERS:

* Do not disable interrupts unless you must. Then be sure to re-enable them.

* Disable interrupts when calling any mouse routine (SEI).

* Do not re-enable interrupts (CLI) or (PLP if previously had done a PHP) after READMOUSE until X & Y data have been removed from the screen holes.

* Be sure to disable interrupts (SEI) before placing position information in the screen holes (POSMOUSE or CLAMPMOUSE).

* Enter all mouse routines (not required for SERVEMOUSF) with the X register set to $Cn and Y register set to $n0 where n = slot number.

* Some programs may need to turn off interrupts for purposes other then reading the mouse. This is sometimes done on the Apple //e to keep from having to handle interrupts while in auxiliary memory. If interrupts are turned off and then back on, the first call to READMOUSE may give incorrect values. Subsequent calls to READMOUSE will return correct values until interrupts are turned off and on again. Turning off interrupts for mouse calls does not create this problem. If you are watching numbers coming form the mouse while moving it in a direction that would increase values you might see the following: 6, 7, 8, 9, 8, 9, 10. In practice this momentary 'glitch' in the stream of mouse data has little importance and would probably only be noticed by a programmer testing his/her program - no one's hand is that steady. If you must keep this 'glitch' from happening then do not keep interrupts off for more then 40 microseconds or be sure that at least one mouse interrupts has taken place since interupts were turned back on.

Apple //c Technical Note #2

Using 40 Column text with Double High Resolution Graphics
22 March 1984

This technical note describes how to properly handle the 40 column screen while using double high-resolution graphics on the Apple //c.

CJS

Many developers using double high resolution (dbl-hi-res) graphics may wish to use 40 column text displays so that the text can be read on a television set. There are a couple of possibilities:

1.) You can define your own dbl-hi-res character set with any size characters you desire and then plot them on the dbl-hi-res screen.

2.) You can print text to the Apple //c text screen and toggle the screen on to display it.

To use the second method, however, does require some special considerations.

The firmware in the Apple //c implements the scroll routine differently that the Apple //e 80 column firmware. The Apple //c scroll routine continues to use the window parameters when scrolling, but uses the 80COL softswitch to determine if it should scroll the 80 or 40 column screen. Since the firmware has initialized a 40-column window, the scroll routines will move only the first 40 columns. But, the 80COL flag has been turned on for dbl-hi-res! Therefore, the scrolling routine takes every even column from auxiliary memory and every odd column from main memory. As a result, only the first 40 columns get scrolled, 20 columns from auxiliary memory and 20 columns from main memory.

One possible solution to the problem is to write your own scroll routines. Another might be to write to the screen so that scrolling will not occur. But these is yet another solution. Turn on the full 80 column mode with a "PR#3" or the equivalent. Now print your text to COUT in the normal manner being careful not to exceed 40 characters per line. The 80 column firmware will scroll everything properly.When you are ready to display text, send a CONTROL-Q through COUT to switch to 40 columns. When you are ready to return to dbl-hi-res mode, send a CONTROL-R to COUT.

When making this switch, a momentary "glitch" may occur. If you send the CONTROL-Q to COUT while still in graphics mode the screen will go to regular "single" hi-res mode before finally going to text mode. If you switch to text mode first, the text will be in 80 column mode (with 40 columns displayed on the left half of the screen) before ultimately going to 40 column mode. The same potential glitch may occur goint back to dbl-hi-res. The "glitch" will be only momentary and may not present any problem for you. If it does, you may wish to make your change-over coincide with the video's verticle blanking interval. (See the Apple //c Reference Manual.)

NOTE: There is no way to display 4 lines of 40 column text at the bottom of the dbl-hi-res screen in mixed mode since the 80 column hardware must be active while dbl-hi-res is being displayed.

Apple //c Technical Note #3

Foreign Language Keyboard Layouts
1 March 1984

There are differences between the keyboard layout on the North American Apple //c and Apple //c's in other coutries. This technical note documents the layouts, along with the ASCII codes for each key, for the French, Italian, German, and United Kingdom systems.

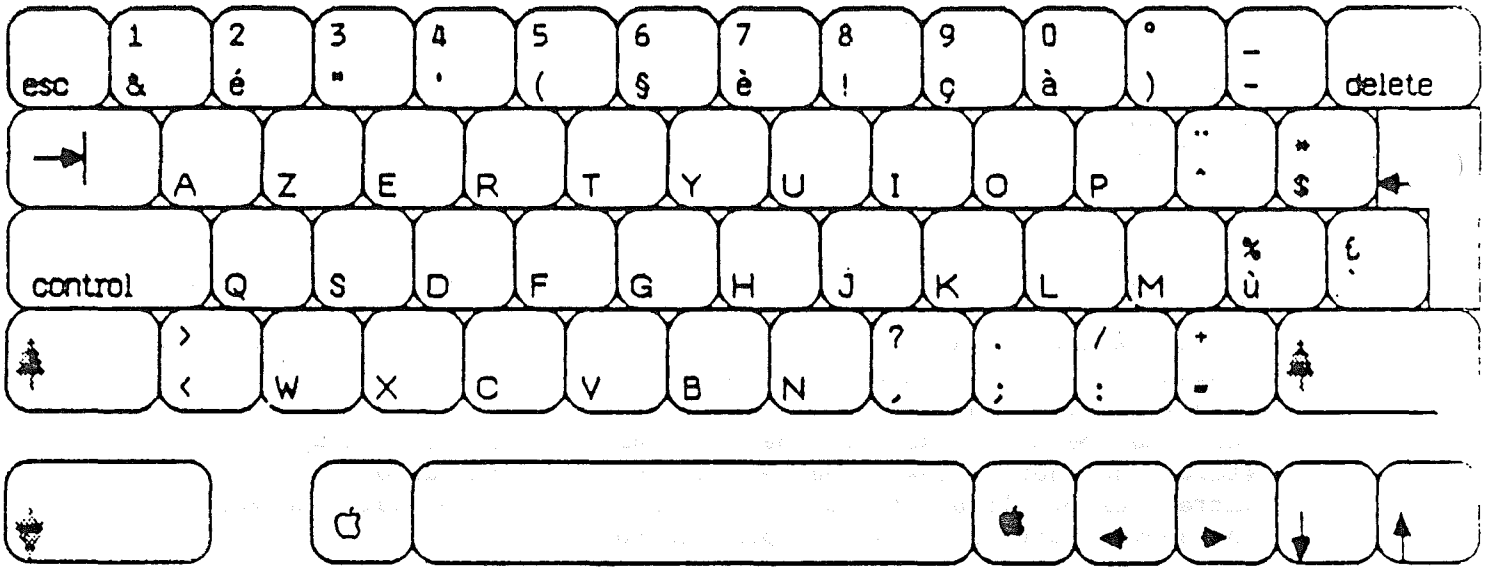
For further information contact:
PCS Developer Technical Support
M/S 22-W Phone (408) 996-1010

PB

Apple //c

Standard France Keyboard Layout

October 24, 1983



Notes: 1) Uses "Shift lock" instead of "Caps lock" -- All keys are shifted.

2) When "Shift Lock" is depressed, "Shift" keys <u>unshift</u> all keys.

Row	Key#	Rom Ad	Char C-S	Code C-S	Char C	Code C	Char S	Code S	Char	Code
1	01	000	ESC	1B	ESC	1B	ESC	1B	ESC	1B
1	02	004	&	26	1	31	&	26	1	31
1	03	008	é	7B	2	32	é	7B	2	32
1	04	00C	"	22	3	33	"	22	3	33
1	05	010	'	27	4	34	'	27	4	34
1	06	018	(	28	5	35	(	28	5	35
1	07	014	GS	1D	GS	1D	§	5D	6	36
1	08	01C	è	7D	7	37	è	7D	7	37
1	09	020	!	21	8	38	!	21	8	38
1	10	024	FS	1C	FS	1C	ç	5C	9	39
1	11	0C0	NUL	00	NUL	00	à	40	0	30
1	12	0C4	ESC	1B	ESC	1B	)	29	°	5B
1	13	0BC	US	1F	US	1F	-	2D	_	5F
1	14	130	DEL	7F	DEL	7F	DEL	7F	DEL	7F
2	16	028	HT	09	HT	09	HT	09	HT	09
2	17	02C	SOH	01	SOH	01	a	61	A	41
2	18	030	SUB	1A	SUB	1A	z	7A	Z	5A
2	19	034	ENQ	05	ENQ	05	e	65	E	45
2	20	038	DC2	12	DC2	12	r	72	R	52
2	21	040	DC4	14	DC4	14	t	74	T	54
2	22	03C	EM	19	EM	19	y	79	Y	59
2	23	044	NAK	15	NAK	15	u	75	U	55
2	24	048	HT	09	HT	09	i	69	I	49
2	25	04C	SI	0F	SI	0F	o	6F	O	4F
2	26	0E4	DLE	10	DLE	10	p	70	P	50
2	27	0E8	RS	1E	RS	1E	^	5E	¨	7E
2	28	0EC	$	24	*	2A	$	24	*	2A
3	31	050	DC1	11	DC1	11	q	71	Q	51
3	32	058	DC3	13	DC3	13	s	73	S	53
3	33	054	EOT	04	EOT	04	d	64	D	44
3	34	060	ACK	06	ACK	06	f	66	F	46
3	35	064	BEL	07	BEL	07	g	67	G	47
3	36	05C	BS	08	BS	08	h	68	H	48
3	37	068	LF	0A	LF	0A	j	6A	J	4A
3	38	06C	VT	0B	VT	0B	k	6B	K	4B
3	39	074	FF	0C	FF	0C	l	6C	L	4C
3	40	070	CR	0D	CR	0D	m	6D	M	4D
3	41	114	ù	7C	%	25	ù	7C	%	25
3	41A	0B8	`	60	£	23	`	60	£	23
3	42	108	CR	0D	CR	0D	CR	0D	CR	0D
4	43A	0E0	DEL	7E	DEL	7E	<	3C	>	3E
4	44	078	ETB	17	ETB	17	w	77	W	57
4	45	07C	CAN	18	CAN	18	x	78	X	58
4	46	080	ETX	03	ETX	03	c	63	C	43
4	47	084	SYN	16	SYN	16	v	76	V	56
4	48	088	STX	02	STX	02	b	62	B	42
4	49	08C	SO	0E	SO	0E	n	6E	N	4E
4	50	090	,	2C	?	3F	,	2C	?	3F
4	51	094	;	3B	.	2E	;	3B	.	2E
4	52	098	:	3A	/	2F	:	3A	/	2F
4	53	09C	=	3D	+	2B	=	3D	+	2B
5	58	110	SP	20	SP	20	SP	20	SP	20
5	60	138	BS	08	BS	08	BS	08	BS	08
5	61	13C	NAK	15	NAK	15	NAK	15	NAK	15

Row	Key#	Rom Ad	Char C-S	Code C-S	Char C	Code C	Char S	Code S	Char	Code
5	62	134	LF	0A	LF	0A	LF	0A	LF	0A
5	63	10C	VT	0B	VT	0B	VT	0B	VT	0B

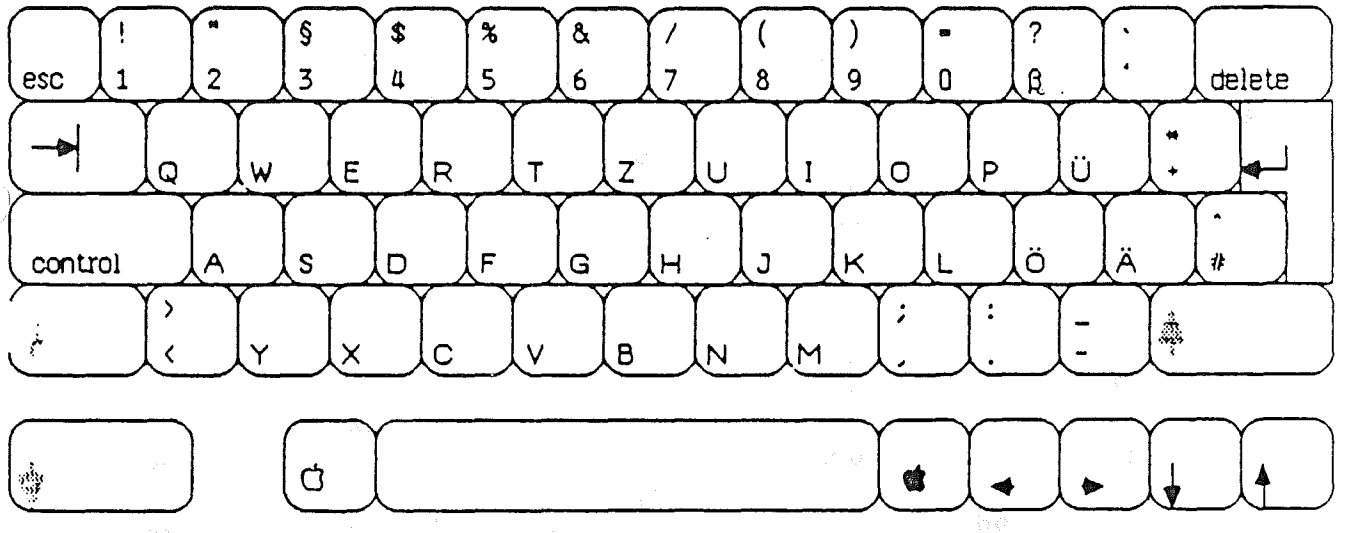
Row	Key#	Rom Ad	Char C-S	Code C-S	Char C	Code C	Char S	Code S	Char	Code
1	01	200	ESC	1B	ESC	1B	ESC	1B	ESC	1B
1	02	204	1	31	&	26	1	31	&	26
1	03	208	2	32	é	7B	2	32	é	7B
1	04	20C	3	33	"	22	3	33	"	22
1	05	210	4	34	'	27	4	34	'	27
1	06	218	5	35	(	28	5	35	(	28
1	07	214	GS	1D	GS	1D	6	36	§	5D
1	08	21C	7	37	è	7D	7	37	è	7D
1	09	220	8	38	!	21	8	38	!	21
1	10	224	FS	1C	FS	1C	9	39	ç	5C
1	11	2C0	NUL	00	NUL	00	0	30	à	40
1	12	2C4	ESC	1B	ESC	1B	°	5B	)	29
1	13	2BC	US	1F	US	1F	_	5F	-	2D
1	14	330	DEL	7F	DEL	7F	DEL	7F	DEL	7F
2	16	228	HT	09	HT	09	HT	09	HT	09
2	17	22C	SOH	01	SOH	01	A	41	a	61
2	18	230	SUB	1A	SUB	1A	Z	5A	z	7A
2	19	234	ENQ	05	ENQ	05	E	45	e	65
2	20	238	DC2	12	DC2	12	R	52	r	72
2	21	240	DC4	14	DC4	14	T	54	t	74
2	22	23c	EM	19	EM	19	Y	59	y	79
2	23	244	NAK	15	NAK	15	U	55	u	75
2	24	248	HT	09	HT	09	I	49	i	69
2	25	24C	SI	0F	SI	0F	O	4F	o	6F
2	26	2E4	DLE	10	DLE	10	P	50	p	70
2	27	2E8	RS	1E	RS	1E	¨	7E	^	5E
2	28	2EC	*	2A	$	24	*	2A	$	24
3	31	250	DC1	11	DC1	11	Q	51	q	71
3	32	258	DC3	13	DC3	13	S	53	s	73
3	33	254	EOT	04	EOT	04	D	44	d	64
3	34	260	ACK	06	ACK	06	F	46	f	66
3	35	264	BEL	07	BEL	07	G	47	g	67
3	36	25C	BS	08	BS	08	H	48	h	68
3	37	268	LF	0A	LF	0A	J	4A	j	6A
3	38	26C	VT	0B	VT	0B	K	4B	k	6B
3	39	274	FF	0C	FF	0C	L	4C	l	6C
3	40	270	CR	0D	CR	0D	M	4D	m	6D
3	41	314	%	25	ù	7C	%	25	ù	7C
3	41A	2B8	£	23	`	60	£	23	`	60
3	42	308	CR	0D	CR	0D	CR	0D	CR	0D
4	43A	2E0	>	3E	<	3C	>	3E	<	3C
4	44	278	ETB	17	ETB	17	W	57	w	77
4	45	27C	CAN	18	CAN	18	X	58	x	78
4	46	280	ETX	03	ETX	03	C	43	c	63
4	47	284	SYN	16	SYN	16	V	56	v	76
4	48	288	STX	02	STX	02	B	42	b	62
4	49	28C	SO	0E	SO	0E	N	4E	n	6E
4	50	290	?	3F	,	2C	?	3F	,	2C
4	51	294	.	2E	;	3B	.	2E	;	3B
4	52	298	/	2F	:	3A	/	2F	:	3A
4	53	29C	+	2B	=	3D	+	2B	=	3D
5	58	310	SP	20	SP	20	SP	20	SP	20
5	60	338	BS	08	BS	08	BS	08	BS	08
5	61	33C	NAK	15	NAK	15	NAK	15	NAK	15

Row	Key#	Rom Ad	Char C-S	Code C-S	Char C	Code C	Char S	Code S	Char	Code
5	62	334	LF	0A	LF	0A	LF	0A	LF	0A
5	63	30C	VT	0B	VT	0B	VT	0B	VT	0B

Apple //c

Standard German Keyboard Layout

October 24, 1983



Super][German Keyboard ROM Map — Alpha Lock

Key Num.	Key Cap	Matrix Number	ROM Addr.	Cntl/Shft Char	Cntl/Shft Code	Control Char	Control Code	Shift Char	Shift Code	Normal Char	Normal Code
01	ESC	00	000:	ESC	1B	ESC	1B	ESC	1B	ESC	1B
02	1!	01	004:	!	21	1	31	!	21	1	31
03	2"	02	008:	"	22	2	32	"	22	2	32
04	3§	03	00C:	NUL	00	NUL	00	§	40	3	33
05	4$	04	010:	$	24	4	34	$	24	4	34
07	6&	05	014:	&	26	6	36	&	26	6	36
06	5%	06	018:	%	25	5	35	%	25	5	35
08	7/	07	01C:	/	2F	7	37	/	2F	7	37
09	8(	08	020:	(	28	8	38	(	28	8	38
10	9)	09	024:	)	29	9	39	)	29	9	39
16	TAB	10	028:	HT	09	HT	09	HT	09	HT	09
17	Q	11	02C:	DC1	11	DC1	11	Q	51	Q	51
18	W	12	030:	ETB	17	ETB	17	W	57	W	57
19	E	13	034:	ENQ	05	ENQ	05	E	45	E	45
20	R	14	038:	DC2	12	DC2	12	R	52	R	52
22	Z	15	03C:	SUB	1A	SUB	1A	Z	5A	Z	5A
21	T	16	040:	DC4	14	DC4	14	T	54	T	54
23	U	17	044:	NAK	15	NAK	15	U	55	U	55
24	I	18	048:	HT	09	HT	09	I	49	I	49
25	O	19	04C:	SI	0F	SI	0F	O	4F	O	4F
31	A	20	050:	SOH	01	SOH	01	A	41	A	41
33	D	21	054:	EOT	04	EOT	04	D	44	D	44
32	S	22	058:	DC3	13	DC3	13	S	53	S	53
36	H	23	05C:	BS	08	BS	08	H	48	H	48
34	F	24	060:	ACK	06	ACK	06	F	46	F	46
35	G	25	064:	BEL	07	BEL	07	G	47	G	47
37	J	26	068:	LF	0A	LF	0A	J	4A	J	4A
38	K	27	06C:	VT	0B	VT	0B	K	4B	K	4B
40	Ö	28	070:	FS	1C	FS	1C	Ö	5C	Ö	5C
39	L	29	074:	FF	0C	FF	0C	L	4C	L	4C
44	Y	30	078:	EM	19	EM	19	Y	59	Y	59
45	X	31	07C:	CAN	18	CAN	18	X	58	X	58
46	C	32	080:	ETX	03	ETX	03	C	43	C	43
47	V	33	084:	SYN	16	SYN	16	V	56	V	56
48	B	34	088:	STX	02	STX	02	B	42	B	42
49	N	35	08C:	SO	0E	SO	0E	N	4E	N	4E
50	M	36	090:	CR	0D	CR	0D	M	4D	M	4D
51	,;	37	094:	;	3B	,	2C	;	3B	,	2C
52	.:	38	098:	:	3A	.	2E	:	3A	.	2E
53	-_	39	09C:	US	1F	US	1F	_	5F	-	2D
E1		40	0A0:	/	2F	/	2F	\	2F	/	2F
E2		41	0A4:	BS	08	BS	08	BS	08	BS	08
E3		42	0A8:	0	30	0	30	0	30	0	30
E4		43	0AC:	1	31	1	31	1	31	1	31
E5		44	0B0:	2	32	2	32	2	32	2	32
E6		45	0B4:	3	33	3	33	3	33	3	33
29	#^	46	0B8:	RS	1E	RS	1E	^	5E	#	23
13	´`	47	0BC:	`	60	´	27	`	60	´	27
11	0=	48	0C0:	=	3D	0	30	=	3D	0	30
12	ß ?	49	0C4:	?	3F	ß	7E	?	3F	ß	7E
E7		50	0C8:	)	29	)	29	)	29	)	29
E8		51	0CC:	ESC	1B	ESC	1B	ESC	1B	ESC	1B

E9	52	0D0:	4	34	4	34	4	34	4	34
E10	53	0D4:	5	35	5	35	5	35	5	35
E11	54	0D8:	6	36	6	36	6	36	6	36
E12	55	0DC:	7	37	7	37	7	37	7	37
56 <>	56	0E0:	>	3E	<	3C	>	3E	<	3C
26 P̤	57	0E4:	DLE	10	DLE	10	P̤	50	P̤	50
27 Ü	58	0E8:	GS	1D	GS	1D	Ü	5D	Ü	5D
28 +*	59	0EC:	*	2A	+	2B	*	2A	+	2B
E13	60	0F0:	*	2A	*	2A	*	2A	*	2A
E14	61	0F4:	NAK	15	NAK	15	NAK	15	NAK	15
E15	62	0F8:	8	38	8	38	8	38	8	38
E16	63	0FC:	9	39	9	39	9	39	9	39
E17	64	100:	.	2E	.	2E	.	2E	.	2E
E18	65	104:	+	2B	+	2B	+	2B	+	2B
42 RETURN	66	108:	CR	0D	CR	0D	CR	0D	CR	0D
63 up	67	10C:	VT	0B	VT	0B	VT	0B	VT	0B
58 space	68	110:	SP	20	SP	20	SP	20	SP	20
41 Ä	69	114:	ESC	1B	ESC	1B	Ä	5B	Ä	5B
E19	70	118:	?	3F	?	3F	?	3F	?	3F
E20	71	11C:	SP	20	SP	20	SP	20	SP	20
E21	72	120:	(	28	(	28	(	28	(	28
E22	73	124:	-	2D	-	2D	-	2D	-	2D
E23	74	128:	CR	0D	CR	0D	CR	0D	CR	0D
E24	75	12C:	,	2C	,	2C	,	2C	,	2C
14 delete	76	130:	DEL	7F	DEL	7F	DEL	7F	DEL	7F
62 down	77	134:	LF	0A	LF	0A	LF	0A	LF	0A
60 left	78	138:	BS	08	BS	08	BS	08	BS	08
61 right	79	13C:	NAK	15	NAK	15	NAK	15	NAK	15

Fill all unused locations with A0.

Super][German Keyboard ROM Map -- Upper/Lower Case

Key Num.	Key Cap	Matrix Number	ROM Addr.	Cntl/Shft Char	Cntl/Shft Code	Control Char	Control Code	Shift Char	Shift Code	Normal Char	Normal Code
01	ESC	00	200:	ESC	1B	ESC	1B	ESC	1B	ESC	1B
02	1!	01	204:	!	21	1	31	!	21	1	31
03	2"	02	208:	"	22	2	32	"	22	2	32
04	3§	03	20C:	NUL	00	NUL	00	§	40	3	33
05	4$	04	210:	$	24	4	34	$	24	4	34
07	6&	05	214:	&	26	6	36	&	26	6	36
06	5%	06	218:	%	25	5	35	%	25	5	35
08	7/	07	21C:	/	2F	7	37	/	2F	7	37
09	8(	08	220:	(	28	8	38	(	28	8	38
10	9)	09	224:	)	29	9	39	)	29	9	39
16	TAB	10	228:	HT	09	HT	09	HT	09	HT	09
17	Q	11	22C:	DC1	11	DC1	11	Q	51	q	71
18	W	12	230:	ETB	17	ETB	17	W	57	w	77
19	E	13	234:	ENQ	05	ENQ	05	E	45	e	65
20	R	14	238:	DC2	12	DC2	12	R	52	r	72
22	Z	15	23C:	SUB	1A	SUB	1A	Z	5A	z	7A
21	T	16	240:	DC4	14	DC4	14	T	54	t	74
23	U	17	244:	NAK	15	NAK	15	U	55	u	75
24	I	18	248:	HT	09	HT	09	I	49	i	69
25	O	19	24C:	SI	0F	SI	0F	O	4F	o	6F
31	A	20	250:	SOH	01	SOH	01	A	41	a	61
33	D	21	254:	EOT	04	EOT	04	D	44	d	64
32	S	22	258:	DC3	13	DC3	13	S	53	s	73
36	H	23	25C:	BS	08	BS	08	H	48	h	68
34	F	24	260:	ACK	06	ACK	06	F	46	f	66
35	G	25	264:	BEL	07	BEL	07	G	47	g	67
37	J	26	268:	LF	0A	LF	0A	J	4A	j	6A
38	K	27	26C:	VT	0B	VT	0B	K	4B	k	6B
40	Ö Ö	28	270:	FS	1C	FS	1C	Ö	5C	ö	7C
39	L	29	274:	FF	0C	FF	0C	L	4C	l	6C
44	Y	30	278:	EM	19	EM	19	Y	59	y	79
45	X	31	27C:	CAN	18	CAN	18	X	58	x	78
46	C	32	280:	ETX	03	ETX	03	C	43	c	63
47	V	33	284:	SYN	16	SYN	16	V	56	v	76
48	B	34	288:	STX	02	STX	02	B	42	b	62
49	N	35	28C:	SO	0E	SO	0E	N	4E	n	6E
50	M	36	290:	CR	0D	CR	0D	M	4D	m	6D
51	,;	37	294:	;	3B	,	2C	;	3B	,	2C
52	.:	38	298:	:	3A	.	2E	:	3A	.	2E
53	-_	39	29C:	US	1F	US	1F	_	5F	-	2D
E1		40	2A0:	/	2F	/	2F	\	2F	/	2F
E2		41	2A4:	BS	08	BS	08	BS	08	BS	08
E3		42	2A8:	0	30	0	30	0	30	0	30
E4		43	2AC:	1	31	1	31	1	31	1	31
E5		44	2B0:	2	32	2	32	2	32	2	32
E6		45	2B4:	3	33	3	33	3	33	3	33
29	#^	46	2B8:	RS	1E	RS	1E	^	5E	#	23
13	´`	47	2BC:	`	60	´	27	`	60	´	27
11	0=	48	2C0:	=	3D	0	30	=	3D	0	30
12	ß ?	49	2C4:	?	3F	ß	7E	?	3F	ß	7E
E7		50	2C8:	)	29	)	29	)	29	)	29
E8		51	2CC:	ESC	1B	ESC	1B	ESC	1B	ESC	1B

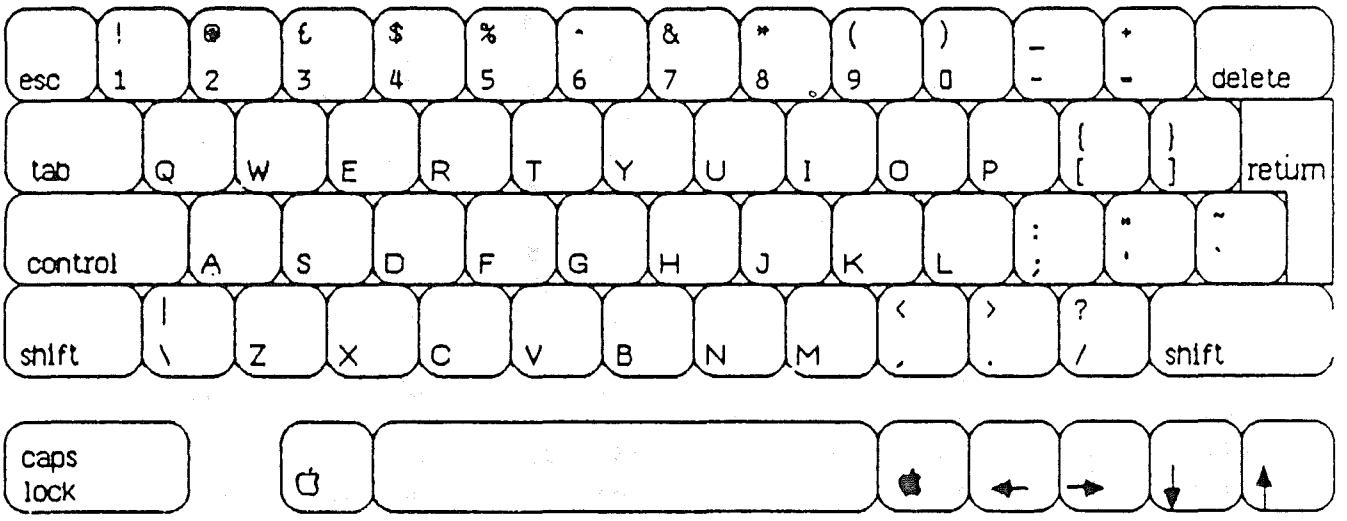
E9	52	2D0:	4	34	4	34	4	34	4	34
E10	53	2D4:	5	35	5	35	5	35	5	35
E11	54	2D8:	6	36	6	36	6	36	6	36
E12	55	2DC:	7	37	7	37	7	37	7	37
56 <>	56	2E0:	>	3E	<	3C	>	3E	<	3C
26 P	57	2E4:	DLE	10	DLE	10	P	50	p	70
27 ü Ü	58	2E8:	GS	1D	GS	1D	Ü	5D	ü	7D
28 +*	59	2EC:	*	2A	+	2B	*	2A	+	2B
E13	60	2F0:	*	2A	*	2A	*	2A	*	2A
E14	61	2F4:	NAK	15	NAK	15	NAK	15	NAK	15
E15	62	2F8:	8	38	8	38	8	38	8	38
E16	63	2FC:	9	39	9	39	9	39	9	39
E17	64	300:	.	2E	.	2E	.	2E	.	2E
E18	65	304:	+	2B	+	2B	+	2B	+	2B
42 RETURN	66	308:	CR	0D	CR	0D	CR	0D	CR	0D
63 up	67	30C:	VT	0B	VT	0B	VT	0B	VT	0B
58 space	68	310:	SP	20	SP	20	SP	20	SP	20
41 ä Ä	69	314:	ESC	1B	ESC	1B	Ä	5B	ä	7B
E19	70	318:	?	3F	?	3F	?	3F	?	3F
E20	71	31C:	SP	20	SP	20	SP	20	SP	20
E21	72	320:	(	28	(	28	(	28	(	28
E22	73	324:	-	2D	-	2D	-	2D	-	2D
E23	74	328:	CR	0D	CR	0D	CR	0D	CR	0D
E24	75	32C:	,	2C	,	2C	,	2C	,	2C
14 delete	76	330:	DEL	7F	DEL	7F	DEL	7F	DEL	7F
62 down	77	334:	LF	0A	LF	0A	LF	0A	LF	0A
60 left	78	338:	BS	08	BS	08	BS	08	BS	08
61 right	79	33C:	NAK	15	NAK	15	NAK	15	NAK	15

Fill all unused locations with A0.

Apple //c

Standard UK Keyboard Layout

October 24, 1983



Notes: Per Neil Davison (October 18, 1983)
Use no symbols on keycaps; instead use:

"shift"
"return"
"caps lock"

Super][British Keyboard ROM Map — Alpha Lock

Key Num.	Key Cap	Matrix Number	ROM Addr.	Cntl/Shft Char	Cntl/Shft Code	Control Char	Control Code	Shift Char	Shift Code	Normal Char	Normal Code
01	ESC	00	000:	ESC	1B	ESC	1B	ESC	1B	ESC	1B
02	1!	01	004:	!	21	1	31	!	21	1	31
03	2@	02	008:	NUL	00	NUL	00	@	40	2	32
04	3£	03	00C:	£	23	3	33	£	23	3	33
05	4$	04	010:	$	24	4	34	$	24	4	34
07	6&	05	014:	RS	1E	RS	1E	^	5E	6	36
06	5%	06	018:	%	25	5	35	%	25	5	35
08	7&	07	01C:	&	26	7	37	&	26	7	37
09	8*	08	020:	*	2A	8	38	*	2A	8	38
10	9(	09	024:	(	28	9	39	(	28	9	39
16	TAB	10	028:	HT	09	HT	09	HT	09	HT	09
17	Q	11	02C:	DC1	11	DC1	11	Q	51	Q	51
18	W	12	030:	ETB	17	ETB	17	W	57	W	57
19	E	13	034:	ENQ	05	ENQ	05	E	45	E	45
20	R	14	038:	DC2	12	DC2	12	R	52	R	52
22	Y	15	03C:	EM	19	EM	19	Y	59	Y	59
21	T	16	040:	DC4	14	DC4	14	T	54	T	54
23	U	17	044:	NAK	15	NAK	15	U	55	U	55
24	I	18	048:	HT	09	HT	09	I	49	I	49
25	O	19	04C:	SI	0F	SI	0F	O	4F	O	4F
31	A	20	050:	SOH	01	SOH	01	A	41	A	41
33	D	21	054:	EOT	04	EOT	04	D	44	D	44
32	S	22	058:	DC3	13	DC3	13	S	53	S	53
36	H	23	05C:	BS	08	BS	08	H	48	H	48
34	F	24	060:	ACK	06	ACK	06	F	46	F	46
35	G	25	064:	BEL	07	BEL	07	G	47	G	47
37	J	26	068:	LF	0A	LF	0A	J	4A	J	4A
38	K	27	06C:	VT	0B	VT	0B	K	4B	K	4B
40	;:	28	070:	:	3A	;	3B	:	3A	;	3B
39	L	29	074:	FF	0C	FF	0C	L	4C	L	4C
44	Z	30	078:	SUB	1A	SUB	1A	Z	5A	Z	5A
45	X	31	07C:	CAN	18	CAN	18	X	58	X	58
46	C	32	080:	ETX	03	ETX	03	C	43	C	43
47	V	33	084:	SYN	16	SYN	16	V	56	V	56
48	B	34	088:	STX	02	STX	02	B	42	B	42
49	N	35	08C:	SO	0E	SO	0E	N	4E	N	4E
50	M	36	090:	CR	0D	CR	0D	M	4D	M	4D
51	,<	37	094:	<	3C	,	2C	<	3C	,	2C
52	.>	38	098:	>	3E	.	2E	>	3E	.	2E
53	/?	39	09C:	?	3F	/	2F	?	3F	/	2F
E1		40	0A0:	/	2F	/	2F	/	2F	/	2F
E2		41	0A4:	BS	08	BS	08	BS	08	BS	08
E3		42	0A8:	0	30	0	30	0	30	0	30
E4		43	0AC:	1	31	1	31	1	31	1	31
E5		44	0B0:	2	32	2	32	2	32	2	32
E6		45	0B4:	3	33	3	33	3	33	3	33
29	`~	46	0B8:	~	7E	`	60	~	7E	`	60
13	=+	47	0BC:	+	2B	=	3D	+	2B	=	3D
11	0)	48	0C0:	)	29	0	30	)	29	0	30
12	-_	49	0C4:	US	1F	US	1F	_	5F	-	2D
E7		50	0C8:	)	29	)	29	)	29	)	29
E8		51	0CC:	ESC	1B	ESC	1B	ESC	1B	ESC	1B

E9	52	0D0:	4	34	4	34	4	34	4	34
E10	53	0D4:	5	35	5	35	5	35	5	35
E11	54	0D8:	6	36	6	36	6	36	6	36
E12	55	0DC:	7	37	7	37	7	37	7	37
56 |	56	0E0:	FS	1C	FS	1C	|	7C	\	5C
26 P	57	0E4:	DLE	10	DLE	10	P	50	P	50
27 [{	58	0E8:	ESC	1B	ESC	1B	{	7B	[	5B
28]}	59	0EC:	GS	1D	GS	1D	}	7D	]	5D
E13	60	0F0:	*	2A	*	2A	*	2A	*	2A
E14	61	0F4:	NAK	15	NAK	15	NAK	15	NAK	15
E15	62	0F8:	8	38	8	38	8	38	8	38
E16	63	0FC:	9	39	9	39	9	39	9	39
E17	64	100:	.	2E	.	2E	.	2E	.	2E
E18	65	104:	+	2B	+	2B	+	2B	+	2B
42 RETURN	66	108:	CR	0D	CR	0D	CR	0D	CR	0D
63 up	67	10C:	VT	0B	VT	0B	VT	0B	VT	0B
58 space	68	110:	SP	20	SP	20	SP	20	SP	20
41 '"	69	114:	"	22	'	27	"	22	'	27
E19	70	118:	?	3F	?	3F	?	3F	?	3F
E20	71	11C:	SP	20	SP	20	SP	20	SP	20
E21	72	120:	(	28	(	28	(	28	(	28
E22	73	124:	-	2D	-	2D	-	2D	-	2D
E23	74	128:	CR	0D	CR	0D	CR	0B	CR	0D
E24	75	12C:	,	2C	,	2C	,	2C	,	2C
14 delete	76	130:	DEL	7F	DEL	7F	DEL	7F	DEL	7F
62 down	77	134:	LF	0A	LF	0A	LF	0A	LF	0A
60 left	78	138:	BS	08	BS	08	BS	08	BS	08
61 right	79	13C:	NAK	15	NAK	15	NAK	15	NAK	15

Fill all unused locations with A0.

Super][British Keyboard ROM Map -- Upper/Lower Case

Key Num.	Key Cap	Matrix Number	ROM Addr.	Cntl/Shft Char	Cntl/Shft Code	Control Char	Control Code	Shift Char	Shift Code	Normal Char	Normal Code
01	ESC	00	200:	ESC	1B	ESC	1B	ESC	1B	ESC	1B
02	1!	01	004:	!	21	1	31	!	21	1	31
03	2@	02	008:	NUL	00	NUL	00	@	40	2	32
04	3£	03	00C:	£	23	3	33	£	23	3	33
05	4$	04	010:	$	24	4	34	$	24	4	34
07	6&	05	014:	RS	1E	RS	1E	^	5E	6	36
06	5%	06	018:	%	25	5	35	%	25	5	35
08	7&	07	01C:	&	26	7	37	&	26	7	37
09	8*	08	020:	*	2A	8	38	*	2A	8	38
10	9(	09	024:	(	28	9	39	(	28	9	39
16	TAB	10	228:	HT	09	HT	09	HT	09	HT	09
17	Q	11	22C:	DC1	11	DC1	11	Q	51	q	71
18	W	12	230:	ETB	17	ETB	17	W	57	w	77
19	E	13	234:	ENQ	05	ENQ	05	E	45	e	65
20	R	14	238:	DC2	12	DC2	12	R	52	r	72
22	Y	15	03C:	EM	19	EM	19	Y	59	y	79
21	T	16	240:	DC4	14	DC4	14	T	54	t	74
23	U	17	244:	NAK	15	NAK	15	U	55	u	75
24	I	18	248:	HT	09	HT	09	I	49	i	69
25	O	19	24C:	SI	0F	SI	0F	O	4F	o	6F
31	A	20	250:	SOH	01	SOH	01	A	41	a	61
33	D	21	254:	EOT	04	EOT	04	D	44	d	64
32	S	22	258:	DC3	13	DC3	13	S	53	s	73
36	H	23	25C:	BS	08	BS	08	H	48	h	68
34	F	24	260:	ACK	06	ACK	06	F	46	f	66
35	G	25	264:	BEL	07	BEL	07	G	47	g	67
37	J	26	268:	LF	0A	LF	0A	J	4A	j	6A
38	K	27	26C:	VT	0B	VT	0B	K	4B	k	6B
40	;:	28	070:	:	3A	;	3B	:	3A	;	3B
39	L	29	274:	FF	0C	FF	0C	L	4C	l	6C
44	Z	30	278:	SUB	1A	SUB	1A	Z	5A	z	7A
45	X	31	27C:	CAN	18	CAN	18	X	58	x	78
46	C	32	280:	ETX	03	ETX	03	C	43	c	63
47	V	33	284:	SYN	16	SYN	16	V	56	v	76
48	B	34	288:	STX	02	STX	02	B	42	b	62
49	N	35	28C:	SO	0E	SO	0E	N	4E	n	6E
50	M	36	290:	CR	0D	CR	0D	M	4D	m	6D
51	,<	37	094:	<	3C	,	2C	<	3C	,	2C
52	.>	38	098:	>	3E	.	2E	>	3E	.	2E
53	/?	39	09C:	?	3F	/	2F	?	3F	/	2F
E1		40	2A0:	/	2F	/	2F	/	2F	/	2F
E2		41	2A4:	BS	08	BS	08	BS	08	BS	08
E3		42	2A8:	0	30	0	30	0	30	0	30
E4		43	2AC:	1	31	1	31	1	31	1	31
E5		44	2B0:	2	32	2	32	2	32	2	32
E6		45	2B4:	3	33	3	33	3	33	3	33
29	`~	46	0B8:	~	7E	`	60	~	7E	`	60
13	=+	47	0BC:	+	2B	=	3D	+	2B	=	3D
11	0)	48	0C0:	)	29	0	30	)	29	0	30
12	-_	49	0C4:	US	1F	US	1F	_	5F	-	2D
E7		50	2C8:	)	29	)	29	)	29	)	29
E8		51	2CC:	ESC	1B	ESC	1B	ESC	1B	ESC	1B

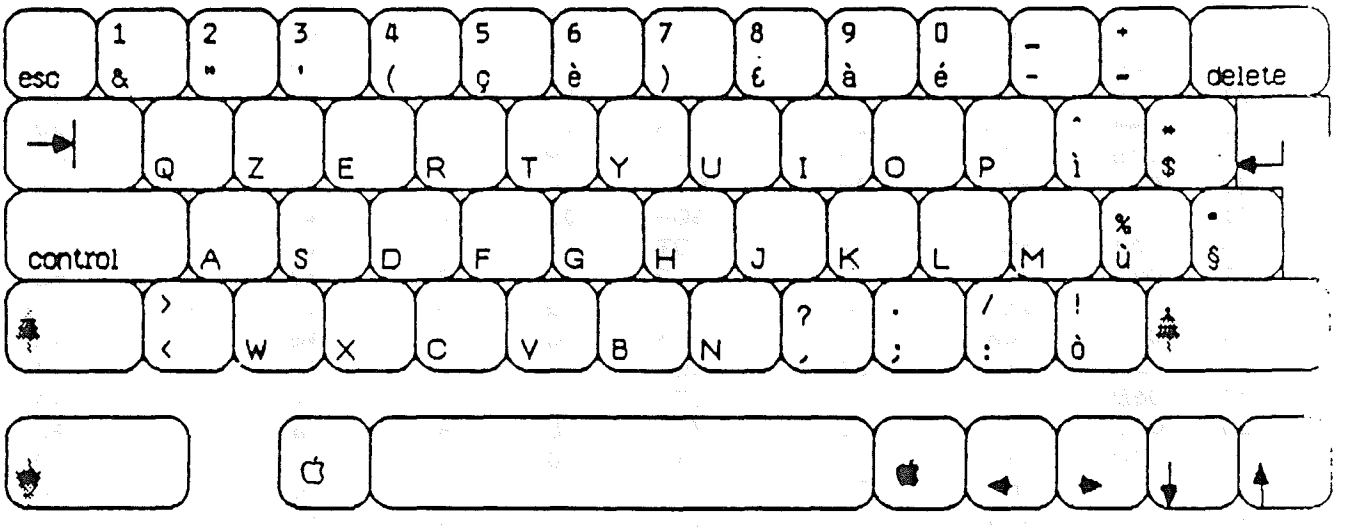
E9	52	2D0:	4	34	4	34	4	34	4	34
E10	53	2D4:	5	35	5	35	5	35	5	35
E11	54	2D8:	6	36	6	36	6	36	6	36
E12	55	2DC:	7	37	7	37	7	37	7	37
56 |	56	0E0:	FS	1C	FS	1C	|	7C	\	5C
26 P	57	0E4:	DLE	10	DLE	10	P	50	p	70
27 [{	58	0E8:	ESC	1B	ESC	1B	{	7B	[	5B
28]}	59	0EC:	GS	1D	GS	1D	}	7D	]	5D
E13	60	2F0:	*	2A	*	2A	*	2A	*	2A
E14	61	2F4:	NAK	15	NAK	15	NAK	15	NAK	15
E15	62	2F8:	8	38	8	38	8	38	8	38
E16	63	2FC:	9	39	9	39	9	39	9	39
E17	64	300:	.	2E	.	2E	.	2E	.	2E
E18	65	304:	+	2B	+	2B	+	2B	+	2B
42 RETURN	66	308:	CR	0D	CR	0D	CR	0D	CR	0D
63 up	67	30C:	VT	0B	VT	0B	VT	0B	VT	0B
58 space	68	310:	SP	20	SP	20	SP	20	SP	20
41 '"	69	314:	"	22	'	27	"	22	'	27
E19	70	318:	?	3F	?	3F	?	3F	?	3F
E20	71	31C:	SP	20	SP	20	SP	20	SP	20
E21	72	320:	(	28	(	28	(	28	(	28
E22	73	324:	-	2D	-	2D	-	2D	-	2D
E23	74	328:	CR	0D	CR	0D	CR	0D	CR	0D
E24	75	32C:	,	2C	,	2C	,	2C	,	2C
14 delete	76	330:	DEL	7F	DEL	7F	DEL	7F	DEL	7F
62 down	77	334:	LF	0A	LF	0A	LF	0A	LF	0A
60 left	78	338:	BS	08	BS	08	BS	08	BS	08
61 right	79	33C:	NAK	15	NAK	15	NAK	15	NAK	15

Fill all unused locations with A0.

Apple //c

Standard Italy Keyboard Layout

October 24, 1983



Notes: 1) Uses "Shift lock" not "Caps Lock" -- All keys are shifted.

2) Alternate character set is U.S. but kbd layout is identical to the Italian -- only characters which are not common to both character sets change.

3) The following characters change to their US equivalents:

Hex	Italian	US	Hex	Italian	US
23	£	#	60	ù	`
40	§	@	7B	à	{
5B	°	[	7C	ò	|
5C	ç	\	7D	è	}
5D	é	]	7E	ì	~

File: ITALIANUC

Report: ROMCODE

Row	Key#	Rom Ad	Char C-S	Code C-S	Char C	Code C	Char S	Code S	Char	Code
1	01	000	ESC	1B	ESC	1B	ESC	1B	ESC	1B
1	02	004	1	31	1	31	1	31	1	31
1	03	008	2	32	2	32	2	32	2	32
1	04	00C	3	33	3	33	3	33	3	33
1	05	010	4	34	4	34	4	34	4	34
1	07	014	6	36	6	36	6	36	6	36
1	06	018	FS	1C	FS	1C	5	35	5	35
1	08	01C	7	37	7	37	7	37	7	37
1	09	020	8	38	8	38	8	38	8	38
1	10	024	9	39	9	39	9	39	9	39
2	16	028	HT	09	HT	09	HT	09	HT	09
2	17	02C	DC1	11	DC1	11	Q	51	Q	51
2	18	030	SUB	1A	SUB	1A	Z	5A	Z	5A
2	19	034	ENQ	05	ENQ	05	E	45	E	45
2	20	038	DC2	12	DC2	12	R	52	R	52
2	22	03C	EM	19	EM	19	Y	59	Y	59
2	21	040	DC4	14	DC4	14	T	54	T	54
2	23	044	NAK	15	NAK	15	U	55	U	55
2	24	048	HT	09	HT	09	I	49	I	49
2	25	04C	SI	0F	SI	0F	O	4F	O	4F
3	31	050	SOH	01	SOH	01	A	41	A	41
3	33	054	EOT	04	EOT	04	D	44	D	44
3	32	058	DC3	13	DC3	13	S	53	S	53
3	36	05C	BS	08	BS	08	H	48	H	48
3	34	060	ACK	06	ACK	06	F	46	F	46
3	35	064	BEL	07	BEL	07	G	47	G	47
3	37	068	LF	0A	LF	0A	J	4A	J	4A
3	38	06C	VT	0B	VT	0B	K	4B	K	4B
3	40	070	CR	0D	CR	0D	M	4D	M	4D
3	39	074	FF	0C	FF	0C	L	4C	L	4C
4	44	078	ETB	17	ETB	17	W	57	W	57
4	45	07C	CAN	18	CAN	18	X	58	X	58
4	46	080	ETX	03	ETX	03	C	43	C	43
4	47	084	SYN	16	SYN	16	V	56	V	56
4	48	088	STX	02	STX	02	B	42	B	42
4	49	08C	SO	0E	SO	0E	N	4E	N	4E
4	50	090	?	3F	?	3F	?	3F	?	3F
4	51	094	.	2E	.	2E	.	2E	.	2E
4	52	098	/	2F	/	2F	/	2F	/	2F
4	53	09C	!	21	!	21	!	21	!	21
3	41A	0B8	NUL	00	ESC	1B	°	5B	°	5B
1	13	0BC	+	2B	+	2B	+	2B	+	2B
1	11	0C0	GS	1D	GS	1D	0	30	0	30
1	12	0C4	US	1F	US	1F	_	5F	_	5F
4	43A	0E0	>	3E	>	3E	>	3E	>	3E
2	26	0E4	DLE	10	DLE	10	P	50	P	50
2	27	0E8	RS	1E	RS	1E	^	5E	^	5E
2	28	0EC	*	2A	*	2A	*	2A	*	2A
3	42	108	CR	0D	CR	0D	CR	0D	CR	0D
5	63	10C	VT	0B	VT	0B	VT	0B	VT	0B

File: ITALIANUC

Report: ROMCODE

Row	Key#	Rom Ad	Char C-S	Code C-S	Char C	Code C	Char S	Code S	Char	Code
5	58	110	SP	20	SP	20	SP	20	SP	20
^	41	114	%	25	%	25	%	25	%	25
	14	130	DEL	7F	DEL	7F	DEL	7F	DEL	7F
5	62	134	LF	0A	LF	0A	LF	0A	LF	0A
5	60	138	BS	08	BS	08	BS	08	BS	08
5	61	13C	NAK	15	NAK	15	NAK	15	NAK	15

File: ITALIANLC

Report: ROMCODE

Row	Key#	Rom Ad	Char C-S	Code C-S	Char C	Code C	Char S	Code S	Char	Code
1	01	200	ESC	1B	ESC	1B	ESC	1B	ESC	1B
1	02	204	1	31	%	26	1	31	%	26
1	03	208	2	32	"	22	2	32	"	22
1	04	20C	3	33	'	27	3	33	'	27
1	05	210	4	34	(	28	4	34	(	28
1	07	214	6	36	è	7D	6	36	è	7D
1	06	218	FS	1C	FS	1C	5	35	ç	5C
1	08	21C	7	37	)	29	7	37	)	29
1	09	220	8	38	£	23	8	38	£	23
1	10	224	9	39	à	7B	9	39	à	7B
2	16	228	HT	09	HT	09	HT	09	HT	09
2	17	22C	DC1	11	DC1	11	Q	51	q	71
2	18	230	SUB	1A	SUB	1A	Z	5A	z	7A
2	19	234	ENQ	05	ENQ	05	E	45	e	65
2	20	238	DC2	12	DC2	12	R	52	r	72
2	22	23c	EM	19	EM	19	Y	59	y	79
2	21	240	DC4	14	DC4	14	T	54	t	74
2	23	244	NAK	15	NAK	15	U	55	u	75
2	24	248	HT	09	HT	09	I	49	i	69
2	25	24C	SI	0F	SI	0F	O	4F	o	6F
3	31	250	SOH	01	SOH	01	A	41	a	61
3	33	254	EOT	04	EOT	04	D	44	d	64
3	32	258	DC3	13	DC3	13	S	53	s	73
3	36	25C	BS	08	BS	08	H	48	h	68
3	34	260	ACK	06	ACK	06	F	46	f	66
3	35	264	BEL	07	BEL	07	G	47	g	67
3	37	268	LF	0A	LF	0A	J	4A	j	6A
3	38	26C	VT	0B	VT	0B	K	4B	k	6B
3	40	270	CR	0D	CR	0D	M	4D	m	6D
3	39	274	FF	0C	FF	0C	L	4C	l	6C
4	44	278	ETB	17	ETB	17	W	57	w	77
4	45	27C	CAN	18	CAN	18	X	58	x	78
4	46	280	ETX	03	ETX	03	C	43	c	63
4	47	284	SYN	16	SYN	16	V	56	v	76
4	48	288	STX	02	STX	02	B	42	b	62
4	49	28C	SO	0E	SO	0E	N	4E	n	6E
4	50	290	?	3F	,	2C	?	3F	,	2C
4	51	294	.	2E	;	3B	.	2E	;	3B
4	52	298	/	2F	:	3A	/	2F	:	3A
4	53	29C	!	21	ò	7C	!	21	ò	7C
3	41A	2B8	ESC	1B	NUL	00	°	5B	§	40
1	13	2BC	+	2B	=	3D	+	2B	=	3D
1	11	2C0	GS	1D	GS	1D	Ø	30	é	5D
1	12	2C4	US	1F	US	1F	_	5F	-	2D
4	43A	2E0	>	3E	<	3C	>	3E	<	3C
2	26	2E4	DLE	10	DLE	10	P	50	p	70
2	27	2E8	RS	1E	RS	1E	^	5E	ì	7E
2	28	2EC	*	2A	$	24	*	2A	$	24
3	42	308	CR	0D	CR	0D	CR	0D	CR	0D
5	63	30C	VT	0B	VT	0B	VT	0B	VT	0B

File: ITALIANLC

Report: ROMCODE

Row	Key#	Rom Ad	Char C-S	Code C-S	Char C	Code C	Char S	Code S	Char	Code
5	58	310	SP	20	SP	20	SP	20	SP	20
'	41	314	%	25	ù	60	%	25	ù	60
.	14	330	DEL	7F	DEL	7F	DEL	7F	DEL	7F
5	62	334	LF	0A	LF	0A	LF	0A	LF	0A
5	60	338	BS	08	BS	08	BS	08	BS	08
5	61	33C	NAK	15	NAK	15	NAK	15	NAK	15

Apple //c Technical Note #4

Corrected DVORAK Keyboard Layout

esc	! 1	@ 2	# 3	$ 4	% 5	^ 6	& 7	* 8	(9	) 0	([	)]	delete	
tab	? /	< ,	> .	P p	Y y	F f	G g	C c	R r	L l	: ;	+ =	\	
control	A a	O o	E e	U u	I i	D d	H h	T t	N n	S s	_ -	return		
shift	" '	Q q	J j	K k	X x	B b	M m	W w	V v	Z z	shift			
caps lock	~ `													
										<--	-->	↓	↑	

This is the Dvorak keyboard layout as implemented on the Apple //c computer when the keyboard switch is depressed. There has only been one version of the keyboard layout ROM. All Apple //c's, pre-release and final production machines, have the above layout and no other. The pre-release documentation was in error as is the final Apple //c Reference Manual.

Apple IIc Delta Guide

Table of Contents

Introduction 1

Apple IIc Firmware 51

Introduction

This document compares the Apple IIc to the Apple IIe, but it also reiterates most of differences between the Apple IIe and the Apple II Plus that were originally noted in the *Guide to the New Features of the Apple IIe* (Apple Product Number A2F2114). In addition, it points out differences between the Apple II and II Plus.

This draft does not include a list of the keyboard and video character sets and other large tables of information. Unless otherwise noted, this information can be found in the *Apple IIe Reference Manual*.

The keyboard and character set differences between different countries' models of the Apple IIc are the same as for the IIe. The *International Supplement to the Apple IIe Owner's Manual* (Part number 030-0525) contains tables and illustrations describing these differences. Note, however, that the Apple IIc has NTSC video circuitry inside the case for all countries; external PAL (and presumably SECAM) video adapters will get their signals from the video expansion connector.

Categories

The characteristics that vary from one machine to another fall under a handful of categories, starting with concrete physical elements and ending with more abstract and technical items:

- Equations

  each machine equals its predecessor plus or minus certain overall characteristics—merely an overview

- External physical

  keyboard layout and front of machine
  sides (yes, sides)
  top (removable or no)
  back panel

- Internal physical

  slots/no slots
  game I/O, aux video pins, LEDs, etc.
  power supply
  disk drive
  speaker

- Input and output

  keyboard character sets
  display character sets
  display modes
  cassette I/O
  disk I/O
  game I/O
  mouse input

- Hardware in general

  type of central processing unit
  amount and address ranges of RAM
  amount and address ranges or ROM
  power supplies

- Firmware in general

  monitor
  video firmware
  diagnostic firmware
  slot/port firmware

- Software in general

  languages
  operating systems

- Hardware specifics

  important RAM locations
  hardware locations
  important ROM locations
  use of ICs (customs, hybrids, sockets)
  signals available to the outside world

Equations

These equations are merely an overview of what each model of Apple II is with respect to its predecessor. The remainder of this guide spells out these differences in detail.

Note: These equations are in terms of functional equivalence, not strict equality. For example,

Apple IIe = Apple II Plus + Language Card

does not mean there is an actual language card or slot—just that the one machine functions as if it were the other with such a card in a slot.

There is a related document (a *Configuration Guide*) that describes how to configure an Apple IIe to make it (almost) equivalent to an Apple IIc..

Apple II Plus	=	Apple II + Autostart ROM + Applesoft firmware + 48K RAM standard
		Apple II - Integer BASIC firmware - Old Monitor ROM

Apple IIe	=	Apple II Plus + language card + additional 16K RAM + 80-column firmware + built-in diagnostics + full ASCII keyboard + internal power-on light + FCC EMC approval + improved back panel + 9-pin back panel game connector + auxiliary slot (with possibility of 80-column card + extra 64K RAM)
		Apple II Plus - slot 0
Apple IIc	=	Apple IIe + extended 80-column text card + 40/80 column switch + language switch + disk light + disk controller port + disk drive + mouse port + serial printer port + serial communication port + built-in port firmware + video expansion connector
		Apple IIe - removable cover - slots 1 to 7 - auxiliary slot - internal power-on light - cassette I/O connectors - internal game I/O connector (hence no game output) - RF modulator connector - auxiliary video pin - diagnostic firmware - miniassembler - monitor cassette support

External Physical

The Apple II and II Plus were identical in external appearance. The Apple IIe and Apple IIc differ from the earlier machines in their keyboard layouts and back panels.

Keyboard

The Apple II and II Plus have identical 52-key keyboards. The Apple IIe and Apple IIc keyboards have the same 63-key, full ASCII keyboard layouts, with new and repositioned keys and characters compared to the Apple II and II Plus. While the Apple II and II Plus have a REPT key, the Apple IIe and IIc have an auto-repeat feature built into each character key.

The Apple IIc has additional switches near its keyboard: one for changing between 40-column and 80-column displays, the other for selecting keyboard layouts (Sholes versus Dvorak on USA models) or keyboard layout and character set (on international models).

The power-on light position differs for the Apple II/II Plus, Apple IIe and Apple IIc. The RESET key also appears in different positions.

Some Apple II and II Plusses have a slide switch inside the case, near the edge of the cover, for selecting whether or not RESET has to be accompanied by CONTROL to work. On the Apple IIe and Apple IIc, there is no choice: CONTROL-RESET works, and RESET alone does not.

Some notable differences in key captions:

	ESCAPE	TAB	CONTROL	SHIFT	CAPS LOCK	DELETE	RETURN	RESET	Other
Apple II	ESC	n/a	CTRL	SHIFT	n/a	n/a	RETURN	RESET	REPT
Apple II +	ESC	n/a	CTRL	SHIFT	n/a	n/a	RETURN	RESET	REPT
Early IIe†	ESC	TAB	CONTROL	SHIFT	CAPS LOCK	DELETE	RETURN	RESET	Apple keys
Later IIe†	Esc	Tab	Control	Shift	Caps Lock	Delete	Return	Reset	Apple keys
Europe IIe	Esc	→|	Control	↑	↓	Delete	←	Reset	Apple keys

†Early Apple IIe's had "two-shot" injection-molded keys (until about June, 1983). After that, manufacturing switched over to a "sublimation" process for applying captions to keys, and changed the captions.

Back Panel

The Apple II and II Plus have three deep notches and two shallow ones on their back panels. The Apple IIe has a metal back panel with 12 numbered rectangular openings with pop-out inserts.

The Apple II, II Plus, and IIe have a video-output phono jack and mini-phono jacks for cassette input and cassette output. The Apple IIe has a DB-9 game input connector that the Apple II and II Plus do not have.

The Apple IIc has the following back-panel connectors, moving from left to right as viewed from the back:

- a game input DB-9 (like the IIe) that is also for the mouse
- a 5-pin DIN connector for serial input and output (Port 2)
- a video expansion output DB-15 for RGB monitor adapter, etc.
- a video output phono jack (same as on all other Apple II's)
- a DB-19 connector for connecting a second disk drive (like IIe)

- a 5-pin DIN connector for serial input and output (Port 1)
- a special recessed male 7-pin DIN connector for 12-volt DC power input (unlike any of the other Apple II's)

The power switch is in the same position (left rear corner) and same orientation (push in top to turn on) for all Apple II's.

Internal Physical

The internal layout of the Apple IIc is irrelevant to this discussion: the user is not to open the Apple IIc case.

The Apple IIe internal layout differs from that of the Apple II and II Plus in several general ways. There are, of course, far fewer components:

- Component layout is different.
- There is no place for plug-in ROMs (like the Programmer's Aid ROM).
- Cards that had a connection on the main logic board on the II and II Plus will not work on the IIe.
- There is a power-on light near the back panel.
- Slot 0 is gone.
- The auxiliary slot is set away from the back panel.

Slots/No Slots

The Apple II and II Plus have 8 identical slots; the IIe has 7 identical slots plus a 60-pin auxiliary slot for video, add-on memory, and test cards. The Apple IIc has no slots; instead, it has built-in hardware and firmware equivalents to slots with cards in them. These are called ports on the Apple IIc.

Game I/O and Other Connectors

The Apple II, II Plus, and IIe have a 16-pin game I/O connector inside the case that supports 3 switch inputs, 4 analog (paddle) inputs, and 4 annunciator outputs. The Apple IIe and IIc have a DB-9 back-panel connector that supports the 3 switch inputs and 4 paddle inputs (2 on the Apple IIc). The Apple IIc does not support the 4 annunciator outputs.

Power Supply

The power supplies for the Apple II, II Plus, and IIe are basically identical; the one for the Apple IIc is quite different from the rest. For further comparisons, see the section under "Hardware in General."

Disk Drive

All of the Apple II series computers are designed to operate with a Disk II drive or its equivalent: 16 sectors, 35 tracks, and so on.

Speaker

The Apple IIe has the same size speaker as the II and II Plus, although it is face down and baffled better. The Apple IIc has a smaller speaker, and, in addition, has a 2-channel (but monaural) mini-phone jack for headphones (which disconnects the internal speaker when something is plugged into it) and a volume control.

Input and Output

This section describes the variations in character sets and other I/O among the Apple II models.

Keyboard Character Sets

The Apple II and II Plus keyboard character sets are the same. They are described in the *Apple II Reference Manual*.

The Apple IIe and IIc keyboard character sets are the same: full ASCII. The standard (Sholes) layout and key assignments are described in the *Apple IIe Reference Manual*. The Dvorak layout and key assignments will be described in the *Apple IIc Reference Manual*.

Display Character Sets

The Apple II and II Plus display character sets are the same: 64 characters of uppercase ASCII (see the *Apple II Reference Manual*). Both the Apple IIe and IIc make available this character set with the addition of lowercase (called the primary set) and an alternate character set (which has inverse lowercase at the expense of flashing characters). Both these sets are described in the *Apple IIe Reference Manual*.

Display Modes

All models have 40-column text mode, low-resolution graphics mode, mixed low-res and 40-column text mode, and high-resolution graphics mode. The Apple IIe (Rev B motherboard) with 80-column text card, and the Apple IIc also have double-high-resolution graphics mode.

Cassette I/O

The Apple II, II Plus and IIe all have cassette input and output jacks, memory locations, and monitor support. The Apple IIc does not.

Disk I/O

The Apple II, II Plus, and IIe can support up to 6 (4 is recommended maximum) disk drives attached to controller cards plugged into slots 6, 5 and 4. The Apple IIc supports its built-in drive (treated as slot 6 drive 1) and one external disk drive (treated as slot 6 drive 2, or as slot 7 drive 1 for external-drive startup purposes.

Game I/O

The Apple II, II Plus, and IIe support game input and output via a 16-pin Dual Inline Pin (DIP) connector inside the case. The Apple IIe and IIc both support game input via a DB-9 connector on their back panels.

Mouse Input

The Apple IIc provides built-in firmware support for a mouse connected to the DB-9 game/mouse connector. The Apple IIe will provide interface card firmware support for a mouse connected to a DB-9 connector that the user installs with the card.

Hardware in General

Type of CPU

The Apple II and II Plus CPU is the 6502. The Apple IIe uses a 6502A, which is capable of a faster clock speed than 1 megahertz (because it is hand-selected from 6502 production), but in fact is not clocked faster than that in the Apple IIe.

The Apple IIc uses the 65C02 as its CPU: this is a redesigned CMOS CPU that has 27 new instructions, new addressing modes, and for some instructions a differing execution scheme. Programs written for the Apple IIc will run on the earlier machines only if they do not contain instructions unique to the 65C02.

Amount and Address Ranges of RAM

Apple II's had as little as 4K of RAM at the time of purchase, but could be upgraded to as much as 48K of RAM by replacing one or more rows of 4 kilobit chips with the (then) newer and noticeably costlier 16 kilobit chips. Changing a matched set of jumper blocks completed the address mapping portion of the conversion. This process is described in the *Apple II Reference Manual*.

The Apple II Plus has 48K of RAM ($0000 through $BFFF) as a standard feature. Addresses $C000 through $FFFF are occupied by ROM only.

Installing an Apple Language Card in an Apple II or II Plus adds the 16K of RAM from $C000 through $FFFF.

The Apple IIe has a full 64K of RAM. The top 12K addresses overlap with the ROM addresses $D000 through $FFFF. There is an additional area of 4K from $D000 through $DFFF. This arrangement is equivalent to an Apple II Plus with an Apple Language Card installed. A program selects between the RAM and

ROM address spaces and between the $Dxxx banks by changing soft switches located in memory. (This process is often called "bank switching.")

With an Apple 80-column Text Card installed in its auxiliary slot, an Apple IIe has an additional 1K of RAM available, for displaying the other 40 columns of 80-column text.

With an Apple Extended 80-Column Text Card installed in its auxiliary slot, an Apple IIe has an additional 64K of RAM available, although no more than half of the 128K of RAM space is available at any given time. Soft switches located in memory control these address space selections.

The RAM in the Apple IIc is equivalent to the RAM in an Apple IIe with an Extended 80-column Card (in other words, with 64K + 64K).

Amount and Address Ranges of ROM

The Apple II and II Plus have from 2K to 12K of firmware in ROM. The uppermost addresses ($F800 through $FFFF) are always used, while other address ranges are optional. Users can plug their own ROMs into the sockets provided. The ROM address range is from $D000 through $FFFF.

The Apple IIe has 16K of ROM (addresses $C100 through $FFFF; page $C0 addresses are for I/O hardware). ROM addresses $C300 through $C3FF (normally assigned to the ROM in a card in slot 3) and $C800 through $CFFF contain 80-column video firmware; ROM addresses $C100 through $C2FF and $C400 through $C7FF (normally assigned to the ROM on cards in slots 1, 2, 4, 5, 6 and 7) contain built-in self-test routines.

A soft switch controls whether the video firmware or slot 3 card ROM is active. Invoking the self-tests with ◆-CONTROL-RESET causes the self-test firmware to take over the slot ROM address spaces.

The Apple IIc ROM also uses the 16K from $C100 through $FFFF, and its 80-column video firmware occupies the same addresses as on the IIe. However, there are no built-in self-tests. Instead, addresses $C100 through $C2FF and $C400 through $C7FF contain the firmware supporting the four built-in I/O ports (printer, communication, mouse, and disk).

Power Supplies

The power supplies for the Apple II, II Plus, and IIe are essentially the same: they convert 110 VAC (220 VAC on most international models) to the voltages required by the circuitry. The Apple IIc, on the other hand, has an external floor transformer that converts 110 VAC (or 220 VAC) to 12 VDC (nominal); the internal power supply then derives the required voltages.

Firmware in General

This section discusses overall blocks of firmware, not about individual routines and their entry points. A full description of those will appear in the *Apple IIc Reference Manual*.

Monitor

The Apple II comes with the so-called Old Monitor ROM, which would put the user into the monitor (* prompt) at startup. The resident interpreter is for Integer BASIC, with ROM space left over for other firmware (such as programmer's aids).

The Apple II Plus, IIe, and IIc come with the Autostart ROM, which tries to load software from the highest slot containing a Disk II controller card or its equivalent. If this attempt fails, the autostart monitor puts the user in the resident Applesoft interpreter (] prompt).

Video Firmware

The video firmware for the Apple IIc is identical to that for the IIe. Because the Apple IIc has no slots, the 80-column video firmware is always present (switched in); there is no possibility of conflict with firmware on a card in slot 3. Also note that there is only one $C800-$CFFF address space: this, too, belongs to 80-column video firmware.

Diagnostic Firmware

Apple II and II Plus do not have built-in diagnostics. The IIe does; it is invoked by pressing ◆-CONTROL-RESET. The Apple IIc has a ◆ key, too, but no built-in diagnostics.

Slot/Port Firmware

The Apple IIc is the only Apple II of the four that has built-in firmware for slots other than "slot 3" (80-column video). In fact, the Apple IIc has hardware, firmware and back-panel connectors that provide the equivalent of:

- a subset of Super Serial Card hardware and firmware, preconfigured for a 1200-baud (maybe 9600-baud) printer in slot 1, with a 5-pin DIN back-panel connector;
- a subset of Super Serial Card hardware and firmware, preconfigured for a 300-baud modem in slot 2, with a 5-pin DIN back-panel connector;
- mouse-interface hardware, firmware in "slot 4" addresses, and a DB-9 back-panel connector shared with game input;
- an enhanced set of disk controller card hardware and firmware, designed to run the built-in drive as Slot 6 Drive 1 (and its equivalents in other operating systems), and the external drive as Slot 6 Drive 2, or even as Slot 7 Drive 1 (PR#7) for system startup from the external drive.

These equivalents of slot-card-firmware-connector are called ports 1, 2, 4, 6 and 7, respectively. By extension, the 80-column video firmware can be called port 3, but only with caution. The Apple IIe and IIc Reference Manuals discuss how to turn the 80-column firmware on and off correctly.

Software in General

This section points out differences to watch out for with respect to programming languages and operating systems that can (or can't) run on the four machines.

Languages

The Apple IIc does not support Pascal 1.0 firmware (I/O) protocols, because its required fixed entry points are impossible to match with the new firmware. Pascal 1.1 is more flexible, and so the Apple IIc can and does support it. Here the entry points are addressed indirectly via a jump table.

The Apple IIc as shipped will not support Integer BASIC because that interpreter does not work under ProDOS. To use Integer BASIC, start the system using the *DOS 3.3 System Master* disk, and invoke Integer BASIC from the keyboard or program.

Former cassette I/O locations now belong to the 40/80-column switch ($C060; was cassette input) and firmware functions ($C02x; was cassette output).

Operating Systems

The Apple IIc will be a ProDOS, rather than a DOS, machine. That does not mean that DOS will not run on it. Rather, we will describe ProDOS as the operating system, ship it and not DOS unless otherwise requested.

CP/M will not currently run on the Apple IIc because it requires plugging a Z80 card into a slot. (Slot? What slot?) Some day there may be another way to make CP/M available, but there isn't right now.

Operating system cassette I/O commands will cause error messages or unpredictable weirdness, depending on how fail-safe the OS is.

Hardware Specifics

The specifics of firmware and I/O storage assignments will be presented in the *Apple IIc Reference Manual*. The sections here discuss the use of integrated and hybrid circuits, and the hard-wired I/O locations in the $C0xx address range.

Use of ICs

The IIe custom chips (Memory Management Unit and Input/Output Unit) replaced more than 50 chips, and added the functionality of dozens more. The IIe PAL replaced several logic chips. The Apple IIc has custom MMU and IOU chips, too, but they have differenct "bonding options"; that is, some of the pins are attached to different parts of the logic inside for the IIe and Apple IIc versions.

In addition, the Apple IIc has a custom General Logic Unit (GLU), Timing Generator (TMG), and Disk Controller Unit (IWM, Integrated Woz (or Wendell) Machine). The Apple IIc has two

hybrid units (AUD and VID) for audio and video amplification; these save space on the PC board and consume less power that the seperate components ("discretes") that they replace.

The trend as one moves from Apple II and II Plus to Apple IIe and IIc is toward fewer and fewer chip sockets. Directly soldering ICs to the circuit board saves money and increases reliability. However, certain key parts (like character generator ROMs) still have sockets. The Apple IIc, in fact, is not intended to be opened by the user—only by Apple manufacturing and service—so for most people, sockets/no sockets is not important.

Hardware Locations

The following table compares the functions that have been hard-wired into the Apple IIe and IIc. Those hard-wired into the Apple II and II Plus are explained in the *Apple II Reference Manual*.

		Apple IIe	Apple IIc
C000	KBD	Keyboard data (0-6) & strobe (read)	Same as on IIe
C000	80STORE	Store in main memory (write)	Same as on IIe
C001		Store in aux memory (write)	Same as on IIe
C002	RAMRD	Read main memory (write)	Same as on IIe
C003		Read aux memory (write)	Same as on IIe
C004	RAMWRT	Write main memory (write)	Same as on IIe
C005		Write aux memory (write)	Same as on IIe
C006	SLOTCXROM	Slot ROMs at Cx00 (write)	Reserved (write)
C007		Internal ROM at Cx00 (write)†	Reserved (write)
C008	ALTZP	Main stack & zero page (write)	Same as on IIe
C009		Aux stack & zero page (write)	Same as on IIe
C00A	SLOTC3ROM	Internal ROM at C300 (write)	Reserved (write)
C00B		Slot ROM at C300 (write)	Reserved (write)
C00C	80COL	80-column display off (write)	Same as on IIe
C00D		80-column display on (write)	Same as on IIe
C00E	ALTCHARSET	Alt. char. set off (write)	Same as on IIe
C00F		Alt. char. set on (write)	Same as on IIe
C01x	KBDSTRB	Clear keyboard strobe (write)	Same as on IIe
C010	RDAKD	Any key down (bit 7)	Same as on IIe
C011	RDBANK	Read bank 1,2 (bit 7 = 1 = bank 2)	Same as on IIe
C012	RDRAM	Read RAM protect/enable (C08x)	Same as on IIe
C013	RDRAMRD	Read RAMRD switch (C002, C003)	Same as on IIe
C014	RDRAMWRT	Read RAMWRT switch (C004, C005)	Same as on IIe
C015	RDSLOTCXROM	Read SLOTCXROM switch (C006, C007)	Reset XINT (read)
C016	RDALTZP	Read ALTZP switch (C008, C009)	Same as on IIe
C017	RDSLOTC3ROM	Read SLOTC3ROM switch (C00A, C00B)	Reset YINT (read)
C018	RD80STORE	Read switch (C000, C001)	Same as on IIe
C019	RDVBL	Read vertical blanking (VBL)	Reset VBLINT (read)‡
C01A	RDTEXT	Read TEXT switch (C050, C051)	Same as on IIe
C01B	RDMIXED	Read MIXED switch (C052, C053)	Same as on IIe
C01C	RDPAGE2	Read PAGE2 switch (C054, C055)	Same as on IIe
C01D	RDHIRES	Read HIRES switch (C057, C058)	Same as on IIe
C01E	RDALTCHARSET	Read ALTCHARSET switch (C00E, C00F)	Same as on IIe
C01F	RD80COL	Read 80COL switch (C00C, C00D)	Same as on IIe
C020			
C021			
C022			
C023			
C024			
C025			
C026			Same as on IIe
C027		Toggle cassette output (read only)	
C028			Reserved (write)
C029			
C02A			
C02B			
C02C			
C02D			
C02E			
C02F			

† This would be more appropriately called INTCXROM
‡ Use $C07x to reset VBLINT and also trigger paddle timers

	Apple IIe	Apple IIc
C030		
C031		
C032		
C033		
C034		
C035		
C036		Same as on IIe
C037	—Toggle speaker (read only)	
C038		Reserved (write)
C039		
C03A		
C03B		
C03C		
C03D		
C03E		
C03F		
C040	Read annunciator 0 (bit 7)	Read X0/Y0 mask status (bit 7)
C041	Read annunciator 1 (bit 7)	Read VBL mask status (bit 7)
C042	Read annunciator 2 (bit 7)	Read X0 edge (1 = falling)
C043	Read annunciator 3 (bit 7)	Read Y0 edge (1 = falling)
C044		Reserved
C045		Reserved
C046		Reserved
C047		Reserved
C048		Read or write resets XINT & YINT
C049		Read or write resets XINT & YINT
C04A		Read or write resets XINT & YINT
C04B		Read or write resets XINT & YINT
C04C		Read or write resets XINT & YINT
C04D		Read or write resets XINT & YINT
C04E		Read or write resets XINT & YINT
C04F		Read or write resets XINT & YINT
C050	Text mode off	Same as on IIe
C051	Text mode on	Same as on IIe
C052	Mixed mode off (if text mode off)	Same as on IIe
C053	Mixed mode on (if text mode off)	Same as on IIe
C054	Page 2 off (depends on 80STORE)	Same as on IIe
C055	Page 2 on (depends on 80STORE)	Same as on IIe
C056	Hi-res clear: use RAMRD and RAMWRT	Same as on IIe
C057	Hi-res set: access hi-res page	Same as on IIe
C058	Annunciator 0 off	Disable mouse X0 & Y0 interrupts†
C059	Annunciator 0 on	Enable mouse X0 & Y0 interrupts†
C05A	Annunciator 1 off	Disable VBL interrupts*
C05B	Annunciator 1 on	Enable VBL interrupts*
C05C	Annunciator 2 off	Interrupt on rising edge of X0†
C05D	Annunciator 2 on	Interrupt on falling edge of X0†
C05E	Annunciator 3 off; set dbl-hi-res	If IOUDIS off: interrupt on rising edge of Y0 If IOUDIS on: set dbl-hi-res
C05F	Annunciator 3 on; clear dbl-hi-res	If IOUDIS off: interrupt on falling edge of Y0 If IOUDIS on: clear dbl-hi-res

† IOUDIS must be off for all these to work; all are R/W reserved if IOUDIS on.

	Apple IIe	Apple IIc
C06x		Reserved (write)
C060	Cassette in (read)	Read 80/40 colume switch (bit 7) (1 = 40 Col {switch down})
C061	Switch input 0 & d key	Same as on IIe (bit 7 = 1 = pressed)
C062	Switch input 1 & CLOSED-APPLE key	Same as on IIe (bit 7 = 1 = pressed)
C063	Switch input 2 (read)†	Read mouse switch (bit 7)
C064	Read analog input 0 (bit 7)	Same as on IIe
C065	Read analog input 1 (bit 7)	Same as on IIe
C066	Read analog input 2 (bit 7)	Read mouse X1 (direction) on bit 7
C067	Read analog input 3 (bit 7)	Read mouse Y1 (direction) on bit 7
C068		Reserved (read)
C069		Reserved (read)
C06A		Reserved (read)
C06B		Reserved (read)
C06C		Reserved (read)
C06D		Reserved (read)
C06E		Reserved (read)
C06F		Reserved (read)
C07x	Analog input reset (paddle trigger)	
C070		Read or write: trigger paddle timer; reset VBLINT
C071		Reserved
C072		Reserved
C073		Reserved
C074		Reserved
C075		Reserved
C076		Reserved
C077		Read: bit 7 = GR (1 = current line is graphics; 0 = it is text)
C078		Reserved
C079		Reserved
C07A		Reserved
C07B		Reserved
C07C		Reserved
C07D		Reserved
C07E		Read: bit 7 = IOUDIS; trigger paddle timer; reset VBLINT Write: set IOUDIS (that is, disable C058-F IOU access & enable DHIRES switch); trigger paddle timer; reset VBLINT
C07F		Read: bit 7 = DHIRES; trigger paddle timer; reset VBLINT Write: clear IOUDIS (that is, enable C058-F IOU access & disable DHIRES switch); trigger paddle timer; reset VBLINT

† Commonly used as shift-key mod on II/II Plus

	Apple IIe	Apple IIc	
C080	Protect RAM Read RAM 2nd D000 Bank	Same as on IIe	
C081	Write RAM Read ROM 2nd D000 Bank†	Same as on IIe	
C082	Protect RAM Read ROM 2nd D000 Bank	Same as on IIe	
C083	Write RAM Read RAM 2nd D000 Bank†	Same as on IIe	
C084	Protect RAM Read RAM 2nd D000 Bank	Reserved	
C085	Write RAM Read ROM 2nd D000 Bank†	Reserved	
C086	Protect RAM Read ROM 2nd D000 Bank	Reserved	
C087	Write RAM Read RAM 2nd D000 Bank†	Reserved	
C088	Protect RAM Read RAM 1st D000 Bank	Same as on IIe	
C089	Write RAM Read ROM 1st D000 Bank†	Same as on IIe	
C08A	Protect RAM Read ROM 1st D000 Bank	Same as on IIe	
C08B	Write RAM Read RAM 1st D000 Bank†	Same as on IIe	
C08C	Protect RAM Read RAM 1st D000 Bank	Reserved	
C08D	Write Ram Read ROM 1st D000 Bank†	Reserved	
C08E	Protect RAM Read ROM 1st D000 Bank	Reserved	
C08F	Write RAM Read RAM 1st D000 Bank†	Reserved	
C090		Reserved (Serial port 1)	
C091		Reserved	
C092		Reserved	
C093		Reserved	
C094		Reserved	
C095		Reserved	
C096		Reserved	
C097		Reserved	
C098	Slot 1 perphieral card I/O	Transmit/Receive Reg	ACIA
C099		Status Register	ACIA
C09A		Command Register	ACIA
C09B		Control Register	ACIA
C09C		Reserved	
C09D		Reserved	
C09E		Reserved	
C09F		Reserved	
C0A0		Reserved (Serial port 2)	
C0A1		Reserved	
C0A2		Reserved	
C0A3		Reserved	
C0A4		Reserved	
C0A5		Reserved	
C0A6		Reserved	
C0A7		Reserved	
C0A8	Slot 2 perphieral card I/O	Transmit/Receive Reg	ACIA
C0A9		Status Register	ACIA
C0AA		Command Register	ACIA
C0AB		Control Register	ACIA
C0AC		Reserved	
C0AD		Reserved	
C0AE		Reserved	
C0AF		Reserved	

† Write RAM requires 2 consecutive read accesses; protect RAM does not.

	Apple IIe	Apple IIc
C0B0		Reserved
C0B1		Reserved
C0B2		Reserved
C0B3		Reserved
C0B4		Reserved
C0B5		Reserved
C0B6		Reserved
C0B7		Reserved
C0B8	Slot 3 perphieral card I/O	Reserved
C0B9		Reserved
C0BA		Reserved
C0BB		Reserved
C0BC		Reserved
C0BD		Reserved
C0BE		Reserved
C0BF		Reserved
C0C0		Reserved
C0C1		Reserved
C0C2		Reserved
C0C3		Reserved
C0C4		Reserved
C0C5		Reserved
C0C6		Reserved
C0C8	Slot 4 perphieral card I/O	Reserved
C0C9		Reserved
C0CA		Reserved
C0CB		Reserved
C0CC		Reserved
C0CD		Reserved
C0CE		Reserved
C0CF		Reserved
C0D0		Reserved
C0D1		Reserved
C0D2		Reserved
C0D3		Reserved
C0D4		Reserved
C0D5		Reserved
C0D6		Reserved
C0D7		Reserved
C0D8	Slot 5 perphieral card I/O	Reserved
C0D9		Reserved
C0DA		Reserved
C0DB		Reserved
C0DC		Reserved
C0DD		Reserved
C0DE		Reserved
C0DF		Reserved

	Apple IIe	Apple IIc
C0E0	Slot 6 perphieral card I/O	Phase 0 = 0 (Disk Controller)
C0E1		Phase 0 = 1
C0E2		Phase 1 = 0
C0E3		Phase 1 = 1
C0E4		Phase 2 = 0
C0E5		Phase 2 = 1
C0E6		Phase 3 = 0
C0E7		Phase 3 = 1
C0E8		Motor off
C0E9		Motor on
C0EA		Drive 1
C0EB		Drive 2
C0EC		L6 = 0
C0ED		L6 = 1
C0EE		L7 = 0
C0EF		L7 = 1
C0F0	Slot 7 perphieral card I/O	Reserved
C0F1		Reserved
C0F2		Reserved
C0F3		Reserved
C0F4		Reserved
C0F5		Reserved
C0F6		Reserved
C0F7		Reserved
C0F8		Reserved
C0F9		Reserved
C0FA		Reserved
C0FB		Reserved
C0FC		Reserved
C0FD		Reserved
C0FE		Reserved
C0FF		Reserved

Monitor Entry Point Labels

This section presents a complete compilation of all $F800 Monitor ROM label occurrences in the various source file listings and the various lists of built-in subroutines. An "X" indicates that the label appears in the source code listing and a "supported" indicates it was found in the list of built-in subroutines.

Sources for this information were:

- *Apple II Reference Manual*

  Page 61 - Some Useful Monitor Subroutines
  Page 155 - Monitor ROM Listing
  Page 136 - Autostart ROM Listing

- *Apple IIe Reference Manual*

  Appendix C - Directory of Built-in Subroutines

- *Apple II Reference Manual Addendum:Monitor ROM Listings*

  Page 3 - Monitor Firmware Listing

- *Apple IIc Reference Manual*

  Appendix C - Important Firmware Locations
  C.5 Monitor Addresses

- *Apple IIc Firmware Assembly List*

Address	Label	Apple II	Apple II Plus	Apple IIe	Apple IIc
$F800	PLOT	X supported	X supported	X	X supported
$F80C	RTMASK	X	X	X	X
$F80E	PLOT1	X	X	X	X
$F819	HLINE	X supported	X supported	X supported	X supported
$F81C	HLINE1	X	X	X	X
$F826	VLINEZ	X	X	X	X
$F828	VLINE	X supported	X supported	X supported	X supported
$F831	RTS1	X	X	X	X
$F832	CLRSCR	X supported	X supported	X supported	X supported
$F836	CLRTOP	X supported	X supported	X supporter	X supported
$F838	CLRSC2	X	X	X	X
$F83C	CLRSC3	X	X	X	X
$F847	GBASCALC	X	X	X	X
$F856	GBCALC	X	X	X	
$F85F	NEXTCOL	supported	supported	supported	
$F85F	NXTCOL	X	X	X	
$F864	SETCOL	X supported	X supported	X supported	X supported
$F871	SCRN	X supported	X supported	X supported	X supported
$F879	SCRN2	X	X	X	X
$F87F	RTMSKZ	X	X	X	X
$F882	INSDS1	X	X	X	X
$F88C	INSDS2				X
$F88E	INSDS2	X	X	X	
$F897	IEVEN				X
$F89B	IEVEN	X	X	X	
$F8A1	ERR				X
$F8A5	ERR	X	X	X	
$F8A5	GETFMT				X
$F8A9	GETFMT	X	X	X	
$F8BE	MNNDX1	X	X	X	X
$F8C2	MNNDX2	X	X	X	X
$F8C9	MNNDX3	X	X	X	X
$F8CD	GOTONE				X
$F8D0	INSTDSP	X	X	X	X
$F8D4	PRNTOP	X	X	X	X
$F8DB	PRNTBL	X	X	X	X
$F8F5	NXTCOL		X		
$F8F5	PRMN1	X		X	X
$F8F9	PRMN2	X	X	X	X
$F910	PRADR1	X	X	X	X
$F914	PRADR2	X	X	X	X
$F926	PRADR3	X	X	X	X
$F92A	PRADR4	X	X	X	X
$F930	PRADR5	X	X	X	X
$F938	RELADR	X	X	X	X
$F940	PRNTYX	X	X	X	X
$F941	PRNTAX	X supported	X supported	X supported	X supported
$F944	PRNTX	X	X	X	X
$F948	PRBLNK	X supported	X supported	X supported	X
$F94A	PRBL2	X supported	X supported	X supported	X supported

"X" = Label appears in source listing
"supported" = Documented as a supported built-in subroutine

Address	Label	Apple II	Apple II Plus	Apple IIe	Apple IIc
$F94C	PRBL3	X	X	X	X
$F953	PCADJ	X	X	X	X
$F954	PCADJ2	X	X	X	X
$F956	PCADJ3	X	X	X	X
$F95C	PCADJ4	X	X	X	X
$F961	RTS2	X	X	X	X
$F962	FMT1	X	X	X	X
$F9A6	FMT2	X	X	X	X
$F9B4	CHAR2				X
$F9B4	CHAR1	X	X	X	
$F9BA	CHAR1				X
$F9BA	CHAR2	X	X	X	
$F9C0	MNEML	X	X	X	X
$FA00	MNEMR	X	X	X	X
$FA40	IRQ		X	X	X
$FA47	NEWBREAK				X
$FA43	STEP	X			
$FA4C	BREAK		X	X	X
$FA4E	XQINIT	X			
$FA59	OLDBRK		X	X	X
$FA62	RESET		X	X	X
$FA6F	INITAN		X	X	
$FA78	XQ1	X			
$FA7A	XQ2	X			
$FA81	NEWMON		X	X	X
$FA85	IRQ	X			
$FA92	BREAK	X			
$FA98	FIXSEV		X	X	X
$FA9C	XBRK	X			
$FAA3	BEEPFIX				X
$FAA3	NOFIX		X	X	X
$FAA5	XRTI	X			
$FAA6	PWRUP		X	X	X
$FAA6	SETPG3		X	X	X
$FAA9	XRTS	X			
$FAAB	SETPLP		X	X	X
$FAAD	PCINC2	X			
$FAAF	PCINC3	X			
$FAB9	XJSR	X			
$FABA	SLOOP		X	X	
$FABD	RESET.X				
$FAC4	XJMP	X			
$FAC5	XJMPAT	X			
$FAC7	NXTBYT		X	X	
$FACD	NEWPCL	X			
$FACF	NOFIX				X
$FAD1	RTNJMP	X			
$FAD2	RTBL				X
$FAD7	REGDSP	X	X	X	X
$FADA	RGDSP1	X	X	X	X

"X" = Label appears in source listing
"supported" = Documented as a supported built-in subroutine

Address	Label	Apple II	Apple II Plus	Apple IIe	Apple IIc
$FAE4	RDSP1	X	X	X	X
$FAFD	PWRCON		X	X	X
$FAFD	BRANCH	X			
$FB02	RGDSP2				X
$FB05	DISKID		X	X	
$FB09	TITLE		X	X	X
$FB0B	NBRNCH	X			
$FB11	XLTBL		X	X	
$FB11	INITBL	X			
$FB12	PWRUP2				X
$FB19	RTBL	X	X	X	
$FB1E	PREAD	X supported	X supported	X supported	X supported
$FB25	PREAD2	X	X	X	X
$FB2E	RTS2D	X	X	X	X
$FB2F	INIT	X	X	X	X
$FB39	SETTXT	X	X	X	X
$FB40	SETGR	X	X	X	X
$FB4B	SETWND	X	X	X	X
$FB59	VTAB23				X
$FB5B	TABV	X	X	X	X
$FB60	APPLEII		X	X	X
$FB60	MULPM	X			
$FB63	MUL	X			
$FB65	STITLE		X	X	X
$FB65	MUL2	X			
$FB6D	MUL3	X			
$FB6F	SETPWRC		X	X	X supported
$FB76	MUL4	X			
$FB78	VIDWAIT		X	X	X
$FB78	MUL5	X			
$FB81	DIVPM	X			
$FB84	DIV	X			
$FB86	DIV2	X			
$FB88	KBDWAIT		X	X	X
$FB94	NOWAIT		X	X	X
$FB97	ESCOLD		X	X	
$FB9B	ESCNOW		X	X	
$FBA0	NEWADV				X
$FBA0	DIV3	X			
$FBA4	MD1	X			
$FBA5	ESCNEW		X	X	
$FBAF	MD2	X			
$FBB0	NEWADV1				X
$FBB3	F8VERSION				X
$FBB3	VERSION			X	
$FBB4	GOTOCX			X	
$FBB4	DOCOUT1				X
$FBB4	MD3	X			
$FBBC	DCX				X
$FBC0	MDRTS	X			

"X" = Label appears in source listing
"supported" = Documented as a supported built-in subroutine

Address	Label	Apple II	Apple II Plus	Apple IIe	Apple IIc
$FBC1	BASCALC	X	X	X	X
$FBD0	BSCLC2	X	X	X	X
$FBD9	CHKBELL				
$FBD9	BELL1	X	X	X	
$FBDD	BELL1	supported	supported	supported	X supported
$FBE4	BELL2	X	X	X	X
$FBEF	RTS2B	X	X	X	X
$FBF0	STORADV		X	X	X
$FBF0	STOADV	X			
$FBF4	ADVANCE	X	X	X	X
$FBF8	ADV2				X
$FBFC	RTS3	X	X	X	X
$FBFD	VIDOUT	X	X	X	X
$FC04	VIDOUT1				X
$FC10	BS	X	X	X	X
$FC1A	UP	X	X	X	X
$FC22	VTAB	X	X	X	X
$FC24	VTABZ	X	X	X	X
$FC2B	RTS4	X	X	X	X
$FC2C	ESC1	X	X	X	
$FC35	NEWOPS				X
$FC38	NEWOP1				X
$FC42	CLREOP	X	X	X supported	X supported
$FC44	CLREOP2				X
$FC46	CLEOP1	X	X	X	X
$FC58	HOME	X	X	X supported	X supported
$FC5D	CLREOP1				X
$FC62	CR	X	X	X	X
$FC66	LF	X	X	X	X
$FC70	SCROLL	X	X	X	X
$FC72	XGOTOCX			X	
$FC73	NEWCR				X
$FC76	SCRL1	X	X		
$FC80	GETINDX				X
$FC84	RDCX			X	
$FC85	CRRTS				X
$FC86	NEWVTAB				X
$FC8C	SCRL2	X	X		
$FC8D	NEWCLREOL				X
$FC90	NEWCLEOLZ				X
$FC91	ISSLOTS			X	
$FC95	SCRL3	X	X		
$FC99	NEWC1				X
$FC99	ISPAGE1			X	
$FC9C	CLREOL	X	X	X supported	X supported
$FC9E	CLEOLZ	X	X	X supported	X supported
$FCA0	CLRLIN				X
$FCA0	CLEOL2	X	X		
$FCA4	CTLDO				X
$FCA8	WAIT	X supported	X supported	X supported	X supported

"X" = Label appears in source listing
"supported" = Documented as a supported built-in subroutine

Address	Label	Apple II	Apple II Plus	Apple IIe	Apple IIc
$FCA9	WAIT2	X	X	X	X
$FCAA	WAIT3	X	X	X	X
$FCB4	NXTA4	X	X	X	X
$FCBA	NXTA1	X	X	X	X
$FCC8	RTS4B	X	X	X	X
$FCC9	HEADR	X	X	X	
$FCCA	COLDSTART				X
$FCD0	BLAST				X
$FCD6	WRBIT	X	X	X	
$FCDB	ZERDLY	X	X	X	
$FCE2	ONEDLY	X	X	X	
$FCE5	WRTAPE	X	X	X	
$FCE7	COM1				X
$FCEC	RDBYTE	X	X	X	
$FCEE	RDBYT2	X	X	X	
$FCF6	COM2				X
$FCFA	RD2BIT	X	X	X	
$FCFC	COM3				X
$FCFD	RDBIT	X	X	X	
$FD03	APPLE2C				X
$FD0C	RDKEY	X supported	X supported	X supported	X supported
$FD18	KEYIN0				X
$FD1B	KEYIN	X supported	X supported	X supported	X supported
$FD20	DONXTCUR				X
$FD21	RDESC			X	
$FD21	KEYIN2	X	*		
$FD25	GOTKEY				X
$FD2F	ESC	X	X	X	
$FD35	RDCHAR	X supported	X supported	X supported	X supported
$FD38	LOOKPICK				X
$FD3D	NOTCR	X	X	X	
$FD44	NOESCAPE				X
$FD45	NOESC1				X
$FD4A	NOESC2				X
$FD5F	NOTCR1	X	X	X	X
$FD62	CANCEL	X	X	X	X
$FD67	GETLNZ	X supported	X supported	X supported	X supported
$FD6A	GETLN	X supported	X supported	X supported	X supported
$FD6F	GETLN1	supported	supported	supported	X supported
$FD71	BCKSPC	X	X	X	X
$FD75	NXTCHAR	X	X	X	X
$FD7E	CAPTST	X	X	X	
$FD84	ADDINP	X	X	X	X
$FD8B	CROUT1	supported	supported	supported	X supported
$FD8E	CROUT	X supported	X supported	X supported	X supported
$FD92	PRA1	X	X	X	X
$FD96	PRYX2	X	X	X	X
$FDA3	XAM8	X	X	X	X
$FDAD	MOD8CHK	X	X	X	X
$FDB3	XAM	X	X	X	X

"X" = Label appears in source listing
"supported" = Documented as a supported built-in subroutine

Address	Label	Apple II	Apple II Plus	Apple IIe	Apple IIc
$FDB6	DATAOUT	X	X	X	X
$FDC5	RTS4C	X	X	X	X
$FDC6	XAMPM	X	X	X	X
$FDD1	ADD	X	X	X	X
$FDDA	PRBYTE	X supported	X supported	X supported	X supported
$FDE3	PRHEX	X supported	X supported	X supported	X supported
$FDE5	PRHEXZ	X	X	X	X
$FDED	COUT	X supported	X supported	X supported	X supported
$FDF0	COUT1	X supproted	X supported	X supported	X supported
$FDF6	COUTZ	X	X	X	X
$FE00	BL1	X	X	X	X
$FE04	BLANK	X	X	X	X
$FE0B	STOR	X	X	X	X
$FE17	RTS5	X	X	X	X
$FE18	SETMODE	X	X	X	X
$FE1D	SETMDZ	X	X	X	X
$FE20	LT	X	X	X	X
$FE22	LT2	X	X	X	X
$FE2C	MOVE	X	X	X supported	X supported
$FE36	VERIFY			supported	X supported
$FE36	VFY	X	X	X	
$FE58	VFYOK	X	X	X	X
$FE5E	LIST	X	X	X	X
$FE63	LIST2	X	X	X	X
$FE75	A1PC	X	X	X	X
$FE78	A1PCLP	X	X	X	X
$FE7F	A1PCRTS	X	X	X	X
$FE80	SETINV	X supported	X supported	X supported	X
$FE84	SETNORM	X supported	X supported	X supported	X
$FE86	SETIFLG	X	X	X	X
$FE89	SETKBD	X	X	X	X
$FE8B	INPORT	X	X	X	X
$FE8D	INPRT	X	X	X	X
$FE93	SETVID	X	X	X	X
$FE95	OUTPORT	X	X	X	X
$FE97	OUTPRT	X	X	X	X
$FE9B	IOPRT	X	X	X	X
$FEA7	NOTPRT0				X
$FEA7	IOPRT1	X	X	X	
$FEA9	IOPRT2	X	X	X	
$FEAB	IOPRT2				X
$FEAF	CKSUMFIX			X	
$FEB0	XBASIC	X	X	X	X
$FEB3	BASCONT	X	X	X	X
$FEB6	GO	X	X	X	X
$FEBF	REGZ	X	X	X	X
$FEC2	OPRT2				X
$FEC2	TRACE	X	X	X	
$FEC4	STEPZ	X	X	X	
$FECA	USR	X	X	X	X

"X" = Label appears in source listing
"supported" = Documented as a supported built-in subroutine

Address	Label	Apple II	Apple II Plus	Apple IIe	Apple IIc
$FECD	WRITE	X	X	X supported	X
$FECE	DOPR0				X
$FED4	WR1	X	X	X	
$FEDE	IOPRT1				X
$FEE2	DECCH				X
$FEE9	CLRCH				X
$FEEB	WDTHCH				X
$FEEC	SETCUR				X
$FEED	WRBYTE	X	X	X	
$FEEE	SETCUR1				X
$FEEF	WRBYT2	X	X	X	
$FEF6	CRMON	X	X	X	X
$FEFD	READ	X	X	X supported	X
$FEFE	OPTBL				X
$FF0A	RD2	X	X	X	
$FF15	INDX				X
$FF16	RD3	X	X	X	
$FF2D	PRERR	X supported	X supported	X supported	X supported
$FF3A	BELL	X supported	X supported	X supported	X supported
$FF3F	IOREST	supported	supported	supported	supported
$FF3F	RESTORE	X	X	X	X
$FF44	RESTR1	X	X	X	X
$FF4A	IOSAVE	supported	supported	supported	supported
$FF4A	SAVE	X	X	X	X
$FF4C	SAV1	X	X	X	X
$FF58	IORTS			supported	
$FF59	OLDRTS		X	X	X
$FF59	RESET	X			
$FF65	MON	X	X	X	X
$FF69	MONZ	X	X	X	X supported
$FF73	NXTITM	X	X	X	X
$FF7A	CHRSRCH	X	X	X	X
$FF8A	DIG	X	X	X	X
$FF90	NXTBIT	X	X	X	X
$FF98	NXTBAS	X	X	X	X
$FFA2	NXTBS2	X	X	X	X
$FFA7	GETNUM	X	X	X	X
$FFAD	NXTCHR	X	X	X	X
$FFBE	TOSUB	X	X	X	X
$FFC7	ZMODE	X	X	X	X
$FFCC	CHRTBL	X	X	X	
$FFCD	CHRTBL				X
$FFE0	SUBTBL				X
$FFE3	SUBTBL	X	X	X	
$FFF7	GETHEX				X
$FFFE	IRQVECT				X

"X" = Label appears in source listing
"supported" = Documented as a supported built-in subroutine

Machine Identification

By looking at the identification bytes in the Monitor ROM, it is possible to identify which machine your software is running on so that it can take advantage of the special features of that particular machine.

The original Apple II and Apple II Plus used two different monitor ROMs: the "original" monitor and the "auto-start" monitor. They are interchangeable between the two machines. In almost every case, it makes no difference whether your software is running on an Apple II or an Apple II Plus, since the hardware was identical, only the Monitor and BASIC ROM sets were changed. This section explains how to determine which Monitor ROM is present, and, if you need to test the BASIC ROMs, you may look at $E000 for a $4C (JMP instruction) to identify an Applesoft ROM set, or a $20 (JSR instruction) to identify an Integer BASIC ROM set.

All other revisions of the Apple II have Applesoft BASIC built in. Note, however, that the Apple III has an Apple II emulation mode, which permits it to emulate a 48K Apple II Plus with either Applesoft or Integer BASIC.

To identify the various Monitor ROMs, look for the following:

Machine	**$FBB3 (64435)**	**$FB1E (64286)**	**$FBC0 (64435)**
Apple II (original monitor)	$38 (56)		
Apple II Plus (autostart monitor)	$EA (234)	$AD (173)	
Apple III emulation mode	$EA (234)	$8A (138)	
Apple IIe	$06 (6)		$EA (234)
Apple IIe with ICON support	$06 (6)		$E0 (224)
Apple IIc	$06 (6)		$00 (0)

Apple's Developer Technical Support group has routines that identify the various versions of the Apple II family. To obtain a copy, write to:

Apple Computer, Inc.
Developer Technical Support
20525 Mariani Ave., MS 22-W
Cupertino, CA 95014

or phone:

(408) 554-5213

Apple IIc Applesoft Firmware Differences

The vectors for the following Applesoft key words:

- SHLOAD
- RESTORE
- STORE
- LOAD
- SAVE

have been changed since they are associated with cassette tape, which is no longer supported. The vectors now point to the ampersand vector so that you can write routines to intercept control when any of these key words appear. If you simply leave the ampersand vector as it is at boot-up, the commands are not rejected with a SYNTAX ERROR, but become "do-nothing" commands.

Under DOS 3.3, hook your routine directly into the ampersand hook at $3F5.

Under ProDOS, $3F5 points to the external command vector in the BASIC.SYSTEM global page. You can hook your routine into the ampersand vector, or into the external command vector in the global page.

In either case, the pointing to the ampersand and/or external command vectors is automatic. No ampersand prefix or "PRINT CONTROL-D" prefix is needed.

Since the Apple IIc has a true uppercase/lowercase keyboard, Applesoft on the Apple IIc will accept and upshift lowercase characters when input in immediate mode. No upshifting will occur

inside of quotes, REMs, DATA statements, or while a BASIC program is executing. All keywords and variable names will be uppercase only when the program is listed.

The Apple IIc firmware supports MouseText. The video firmware, when properly enabled, is able to display a set of graphic characters that were designed to be used with the mouse. To use the mouse characters:

- Turn on the video firmware (PR#3)
- Enable mouse characters (PRINT CHR$(27) {Hex $1B})
- Set inverse mode (INVERSE or PRINT CHR$(15) {Hex $0F})
- Print capital letters or PRINT CHR$(64 to 95)
- Disable mouse characters (PRINT CHR$(24) {Hex $18})
- Set normal mode (PRINT CHR$(14) {Hex $0E})

When actually in screen memory, the 32 mouse characters have ASCII codes 64 - 95 ($40 - $5F). Inverse characters that previously occupied that range are remapped to ASCII codes 0 - 31 ($00 = $1F).

Interrupt Handling on the Apple IIc

This document contains excerpts from the *Apple IIc Reference Manual* and describes the handling of IRQ interrupts. It is intended as an overview of the interrupt capabilities of the Apple IIc. It is not intended as a programmer's guide. The full details are in the *Apple IIc Reference Manual*.

What Is an Interrupt?

On a computer, an interrupt is a signal that tells the computer to stop what it is currently doing and devote its attention to a more important task. For example, the Apple IIc mouse sends an interrupt to the computer every time it moves. This is necessary because, unless the mouse is read shortly after it moves, the signal indicating its direction is lost.

Interrupts on the Apple IIc Computer

The Apple IIc built-in interrupt handler, unlike earlier systems in the Apple II family, now saves the accumulator on the stack instead of in location $45. Thus, both DOS and the Monitor work with interrupts on the Apple IIc.

Interrupts are effective only if they are enabled most of the time. Interrupts that occur while interrupts are disabled cannot be detected. Due to the critical timing nature of disk reads and writes, Pascal, DOS, and ProDOS turn off interrupts while performing disk operations. Thus, it is important to remember that while a disk drive is being accessed, all sources of IRQ interrupts are, in effect, turned off.

Interrupt Handling on the 65C02

From the point of view of the 65C02 in the Apple IIc, there are two possible causes of interrupts:

1. If interrupts to the 65C02 are not masked (that is, the CLI instruction has been used), the IRQ line on the microprocessor could be pulled low.
2. The processor executed a break instruction (BRK = opcode $00)

   (NOTE: The NMI line in the Apple IIc is not used, thus an NMI interrupt can never happen.)

These two options cause the 65C02 to save the current program counter and status byte on the stack and then jump to the routine whose address is stored in $FFFE and $FFFF. The sequence performed by the 65C02 is:

- If IRQ, finish executing the current instruction.
- Push high byte of program counter onto stack.
- Push low byte of program counter onto stack.
- Push status byte onto stack.
- Jump to address stored in $FFFE, $FFFF [JMP ($FFFE)].

The Interrupt Vector at $FFFE

In the Apple IIc computer, there are three separate regions of memory that contain address $FFFE: the built-in ROM, the bank-switched memory in main RAM, and the bank-switched memory in auxiliary RAM. The vector at $FFFE in the ROM points to the Apple IIc's built-in interrupt-handling routine. Because the interrupts in the Apple IIc are complex, we recommend that you use it rather than write your own interrupt-handling routine.

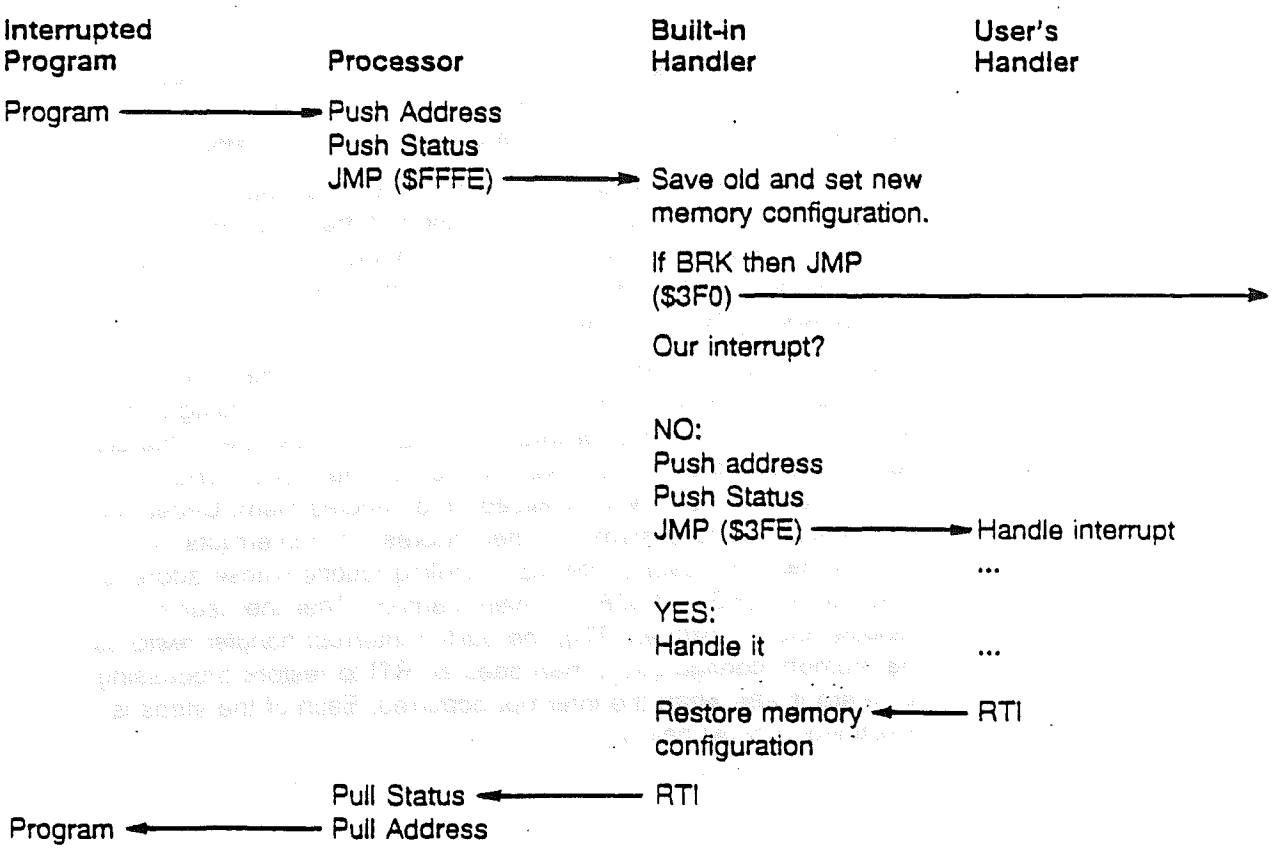
When you initialize the mouse firmware or the communications firmware, copies of the ROM's interrupt vector are placed in the interrupt vector's addresses in both main and auxiliary bank-switched memory. If you plan to use interrupts and the bank-switched memory without the mouse or communications firmware, you must copy the ROM's interrupt vector yourself.

The Built-in Interrupt Handler

The built-in interrupt handler is responsible for determining whether a break or an interrupt occurred. If an interrupt occurred, the built-in handler decides whether the interrupt should be handled internally, handled by the user, or simply ignored.

The built-in interrupt-handling routine records the state of the computer's current memory configuration. It then sets the computer's memory configuration to a standard state. This allows a user's interrupt handler to know the precise memory configuration when it is called.

Next, the built-in interrupt handler checks to see if the interrupt was caused by a break instruction and handles it accordingly. If it was not a break, it looks for interrupts that it knows how to handle (for example, if the interrupt was caused by the mouse, and the mouse has been properly initialized) and handles them. Depending on the state of the system, it either ignores other interrupts or passes them to a user's interrupt-handling routine whose address is stored at $3FE and $3FF of main memory. After the user's handler returns (with an RTI), the built-in interrupt handler restores the memory configuration, then does an RTI to restore processing to where it was when the interrupt occurred. Each of the steps is explained in detail below.



Saving the Memory Configuration

The built-in interrupt handler saves the state of the system and sets it to a known state according to these rules:

- If 80STORE and PAGE2 are on, then text page 1 is switched in so that main screen holes are accessible (PAGE2 off).
- Main memory is switched in for reading (RAMRD off).
- Main memory is switched in for writing (RAMWRT off).
- $D000-$FFF ROM is switched in for reading (RDLCRAM off).

- Main stack and zero page are switched in (ALTZP off).
- Auxiliary stack pointer is preserved, and the main stack is restored.

Since main memory is switched in, all memory addresses used later in this section are in main memory unless otherwise specified.

Managing the Memory Configuration

Because the Apple IIc has two stack pages, we have adopted a convention that allows the system to be run with two separate stack pointers. Two bytes in the auxiliary stack page are to be used as storage for inactive stack pointers: $100 for the main stack pointer when the auxiliary stack is active, and $101 for the auxiliary stack pointer when the main stack is active.

When a program uses interrupt switches in the auxiliary stack for the first time, it should place the value of the main stack pointer at $100 in the auxiliary stack and initialize the auxiliary stack pointer to $FF (the top of the stack). When it subsequently switches from one stack to the other, it should save the current stack pointer before loading the pointer for the other stack.

User's Interrupt Handler at $3FE

The screen hole locations can be set up to indicate that the user's interrupt handler should be called when certain interrupts occur. To use such a routine, place the address of the routine at $3FE and $3FF in main memory (low byte first).

The user's interrupt handler should

- verify that the interrupt came from the expected source;
- handle the interrupt as desired;
- ccear the interrupt, if necessary;
- return with an RTI.

In general, there is no guaranteed response time for interrupts because the system may be doing a disk operation that could last for several seconds.

Once the built-in interrupt handler has been called, it takes about 250 to 300 microseconds for it to call your interrupt handling routine. After you routine returns, it takes 40 to 140 microseconds to restore memory and return to the interrupted program.

Sources of Interrupts

The Apple IIc can receive interrupts from many different sources. Each source is enabled and used slightly differently than the others. There are two basic classes of interrupt sources: those associated with use of the mouse, and those associated with the two 6551 ACIA circuits.

The interrupts associated with the mouse are

- an interrupt generated when the mouse is moved in the horizontal (X) direction
- an interrupt generated when the mouse is moved in the verticle (Y) direction
- an interrupt generated every 1/60 second, synchronized with the video vertical blanking signal
- using the firmware, an interrupt generated when the mouse button is pressed.

The interrupts associated with the ACIA's are

- an interrupt generated when a key is pressed
- an interrupt generated by a device attached to the external disk drive port
- an interrupt generated when either ACIA has received a byte of data from its port
- an interrupt generated when pin 5 of either serial port changed state
- an interrupt generated when either ACIA is ready to accept another character to be transmitted
- an interrupt generated when the keyboard strobe is cleared.

Firmware-Handling of Interrupts

The following sections present an overview of how the built-in firmware handles interrupts.

Firmware for Mouse and Vertical Blanking

When the mouse is initialized, the interrupt vector is copied to main and auxiliary bank-switched RAM. When the mouse is active, possible sources of interrupts are

- mouse movement in the X direction
- mouse movement in the Y direction
- change of state of the button
- leading edge of the vertical blanking signal.

When an interrupt occurs, the built-in interrupt handler determines whether that particular interrupt source was enabled by the SETMOUSE call. If so, the user's interrupt handler, whose address is stored at $3FE, is called.

The interrupt handler should first call SERVEMOUSE to determine the source of the interrupt. If the interrupt was due to mouse movement or button, the interrupt handler should then do a call to READMOUSE. The interrupt should then be serviced and terminated with an RTI.

Remember: An interrupt may be missed during disk accesses.

If you turn on mouse interrupts without initializing the mouse, the built-in interrupt handler will absorb the interrupts. If you wish to handle mouse interrupts yourself, you must write your own interrupt handler and place vectors to it in bank-switched RAM. Interrupts will be ignored whenever the $D000-$FFFF ROM is switched in.

Firmware for Keyboard Interrupts

The Apple IIc is able to generate an interrupt when a key is pressed. Keyboard interrupts are received through the ACIA for port 2. When the user's interrupt handler is called, it can identify the interrupt source as the keyboard rather than the serial port.

The firmware is able to buffer up to 128 keystrokes. After the buffer is full, any additional keystrokes are ignored. Because interrupts are generated only when a key is pressed; auto-repeated characters are not buffered.

Once keyboard buffering has been turned on, the next key should be read by calling RDKEY ($FD0C). Pressing ⌘-CONTROL-X clears the buffer.

Keyboard buffering is automatically turned on when the serial firmware is placed in Terminal mode. Otherwise, you must turn it on yourself. A PR#2 or IN#2 or the equivalent will shut off keyboard buffering.

Using External Interrupts Through Firmware

Pin 9 of the external disk drive connector (EXTINT) can be used to generate interrupts through the ACIA for port 1. It can be used as a source of interrupts (on a high-to-low transition) if enabled.

When the user's interrupt handler is called, it can identify the source of the interrupt.

Firmware for Serial Interrupts

The Apple IIc is able to generate interrupts both when the ACIA received data and when it is ready to send data. The built-in interrupt handler responds to incoming data only. The firmware is able to buffer up to 128 incoming bytes of serial data from either serial port. After the buffer is full, data are ignored. Only one port can be buffered at a time.

Serial buffering is automatically turned on when serial firmware is placed in Terminal mode. Otherwise, you must turn it on yourself. When enabled, normal reads from the serial port firmware fetch data from the buffer rather than directly from the ACIA.

It is also possible to use the firmware to call the user interrupt handler whenever a byte of data is read by the ACIA. In this mode, buffering is not performed by the firmware. When thus enabled, the user's interrupt handler is called each time the port receives a byte of data. The handler can identify the source of the interrupt.

The serial firmware does not implement buffering for serial output. Instead, it waits for two conditions to be true before transmitting a character:

- The ACIA's transmit register must be ready to accept a character.
- The device must signal that it is ready to accept data.

A Loophole in the Firmware

So that programs can make use of interrupts on the ACIAs without affecting mouse interrupt handling, we left a time loophole in the built-in handler. If transmit interrupts are enabled on the ACIA, then control is passed to the user's interrupt handler if the interrupt is not intended for the mouse (movement, button, or VBL).

This means that you can write more sophisticated serial interrupt-handling routines than we could provide (such as printer spooling). The firmware will still set memory to its standard state, handle mouse interrupts, and restore memory after your routine is finished.

When you receive the interrupt, neither ACIA's status register has been read. It is your responsibility to check for interrupts on both ACIAs. You must determine which of the four interrupt sources on each ACIA caused the interrupt and how to handle them. The built-in firmware itself is an excellent example of how interrupts on the ACIA can be handled.

Apple IIc Firmware

This section is a brief user's guide to the firmware of the Apple IIc. It assumes that you are familiar with the use and operation of the Apple IIe, and it places emphasis on the differences between the IIe and IIc.

Video Firmware

40 Columns Versus 80 Columns

The Apple IIe has two distinct video modes: Apple II mode (checkerboard cursor) and Apple IIe mode (solid cursor). The system boots up in Apple II mode; you switch to Apple IIe mode with the PR#3 commmand and return to Apple II mode using ESC CONTROL-Q. On the Apple IIc, the commands ESC 4 and ESC 8 will also switch into Apple IIe mode.

Diagnostics

The Apple IIc does not have a diagnostic program as we know it in the Apple IIe. Instead, it has a memory exerciser that exercises all the RAM and I/O switches. To activate it, press

-CONTROL-RESET

To reboot the system, press

CONTROL-RESET

65C02 Microprocessor

The Apple IIc uses the 65C02 microprocessor, an extended version of the 6502 chip used in the IIe. If you use the Monitor program in the Apple IIc, you will find that the L command (List) disassembles the extended instruction set provided by the 65C02. (The ProDOS version of EDASM supports this extended instruction set if you use the X6502 directive.)

Window Widths

The Apple IIe video firmware allows only even window widths and window left edges when you are using 80-column mode. The Apple IIc video firmware allows you to use both odd and even window widths in all situations.

Mouse Firmware

Mouse Character Set

The Apple IIc is endowed with the world-famous mouse character set. The Apple IIc character ROM, when properly enabled, is able to display a set of graphics characters that were designed to be used with the mouse. To use the mouse characters

- Turn on the video firmware (use the PR#3 command).
- Enable mouse characters (PRINT CHR$(27) (hex $1B)).
- Set inverse mode.
- Print capital letters.
- Disable mouse characters (PRINT CHR$(24) (hex $18)).
- Set normal mode.

The mouse character set itself is included at the end of this document. Here is a BASIC program that prints all the mouse characters:

```
10  D$=CHR$(4)
20  PRINT D$;"PR#3"
30  INVERSE
40  PRINT
CHR$(27);"@ABCDEFGHIJKLMNOPQRSTUVWXYZ[]^";CHR$(24);
50  NORMAL
```

The 32 mouse characters have ASCII codes 64-95 ($40-$5F).

Using the Mouse as Paddles

With the Apple IIc, the mouse can either be used instead of the paddles (not true of the IIe), or as an X-Y pointing device in slot 4. If the mouse is turned on, the monitor ROM paddle routines will take input from the mouse instead of from the paddles. This is acceptable because the mouse and the paddles (and the joystick) are all plugged into the same port in the back of the Apple IIc. For example, a BASIC program that uses the PDL function to read from the paddles works just as well reading from the mouse. Try this:

1. Boot DOS 3.3 (the old one with LITTLE BRICK OUT on it).
2. Type `PR#4` and press RETURN to turn on the mouse.
3. Press CONTROL-A and then press RETURN to initialize the mouse.
4. Type `PR#0` and press RETURN to restore output to the screen.
5. Type `RUN LITTLE BRICK OUT` and press RETURN to run the program.

Play LITTLE BRICK OUT using the mouse instead of the paddles. Ignore the clicking noise when you move the mouse. This is a diagnostic aid that tells us that the mouse is alive and squeaking.

Using the Mouse From BASIC

If you would rather use the mouse in a more conventional manner, you can treat it as a device in slot 4. The general method is like this:

1. Initialize the mouse by printing a 1 to it.
2. Set input to come from slot 4.
3. INPUT X, Y, and button status from the mouse.
4. When done, set input to come from slot 0 (or 3).

Here is a BASIC program that demonstrates the use of the mouse. It reads from the mouse and prints the current values to the screen. When you press and then release the mouse button, the X and Y settings are reinitialized to 0. When a (readable) key is pressed, the program ends.

The X and Y coordinates are initialized to 0 when you print a 1 to the mouse firmware. They have a range from 0 to 1023. The mouse button returns values are as follows:

```
+/- 2  =  just pressed
+/- 1  =  still pressed
+/- 3  =  just released
+/- 4  =  still up
```

The value of the button status is normally positive. It becomes negative if a key is pressed.

The Built-in Printer Firmware

The Apple IIc printer firmware is intact and works from BASIC. However, its ID bytes do not identify it as any existing peripheral card. Thus, anyone (i.e. Pascal) that looks at ID bytes will not be able to use it. To use the serial Dot Matrix Printer (Imagewriter) from BASIC:

1. Set printer DIP switches like this:

   DN DN DN UP DN DN DN DN

   DN DN UP UP

2. Type PR#1 to direct output to the printer.
3. Subsequent output goes to the printer.
4. Type PR#0 (or PR#3) to redirect output to the screen.

By default, the printer firmware has the following settings:

- 9600 baud
- 8 data bits, 1 stop bit
- no parity
- 80-column line width with no video echo
- Line feed generated after RETURN
- Delay after line feed of 250 ms (1/4 second)
- Default command character is set to CONTROL-I.

These settings can be changed as described below.

Printer Firmware Commands

Once the printer firmware has been activated (by a PR#1), it operates very much like the Apple II Super Serial Card when it is in printer mode. Refer to the *Super Serial Card Manual* for more details on using the following commands. ^I means CONTROL-I.

^InnB Set baud rate to nn

Baud Rate	nn
50	1
75	2
110	3
135	4
150	5
300	6
600	7
1200	8
1800	9
2400	10
3600	11
4800	12
7200	13
9600	14
19200	15

^ InnD Set data format bits to nn

Data Format	nn
8 data, 1 stop	0
7 data, 1 stop	1
6 data, 1 stop	2
5 data, 1 stop	3
8 data, 2 stop	4
7 data, 2 stop	5
6 data, 2 stop	6
5 data, 2 stop	7

^ II Enable video echo

^ IK Disable linefeed after CR

^ IL Enable linefeed after CR

^ InnN Disable video echo and set printer width to nn. nn is printer width in decimal.

^ InnP Set parity bits to nn

Parity	nn
none	0,2,4,6
odd	1
even	3
MARK	5
SPACE	7

^ IZ Zap control commands

^ IX Set command char to ^ X (default, ^ I)

^ InnCR Set printer width (CR = carriage return). Video echo must be disabled.

The Built-in Communcations Firmware

The Apple IIc communications firmware is intact and works from BASIC. Its ID bytes identify it as an Apple II communications card to most programs, and as a Super Serial Card to Access II.

Refer to the Apple II Super Serial Card manual for a description of the use of the communications firmware (Chapter 3, Communications Mode).

Communications Firmware Commands

Refer to the Super Serial Card manual for more details on the use of the following commands. ^A means CONTROL-A.

^AnnB Set baud rate to nn

Baud Rate	nn
50	1
75	2
110	3
135	4
150	5
300	6
600	7
1200	8
1800	9
2400	10
3600	11
4800	12
7200	13
9600	14
19200	15

^AnnD Set data formmat bits to nn

Data Format	nn
8 data, 1 stop	0
7 data, 1 stop	1
6 data, 1 stop	2
5 data, 1 stop	3
8 data, 2 stop	4
7 data, 2 stop	5
6 data, 2 stop	6
5 data, 2 stop	7

^AI Enable video echo

^AK Disable linefeed after CR

^AL Enable linefeed after CR

^AnnN Disable video echo and set printer width to nn. nn is printer width in decimal.

^AnnP Set parity bits to nn

Parity	nn
none	0,2,4,6
odd	1
even	3
MARK	5
SPACE	7

^AQ Quit terminal mode

^AR Reset the ACIA, IN#0, PR#0

^AS Send a 233 ms break character

^AT Enter Terminal mode

^AZ Zap control commands

^AX Set command char to ^X (default, ^A)

^AnnCR Set printer width (CR = carriage return). Video echo must be disabled.

Mouse Technical Note #1
15-Mar-84

This technical note explains what you need to be concerned about regarding the computer's interrupt environment with the mouse, regardless of whether you are using interrupts or not.

For further information contact:
PCS Developer Technical Support
M/S 22-W. Phone (408) 996-1010

/bg

Software developers who are writting mouse based programs in machine language need to be concerned about the computer's interrupt environment even if they are using the mouse in passive mode. Listed below are several conditions which a machine language programmer should take into account if their programs are to run on the Apple // family of computers.

* Do not disable interrupts unless you must. Then be sure to re-enable them.

* Disable interrupts when calling any mouse routine (SEI).

* Do not re-enable interrupts (CLI) or (PLP if previously had done a PHP) after READMOUSE until X & Y data have been removed from the screen holes.

* Be sure to disable interrupts (SEI) before placing position information in the screen holes (POSMOUSE or CLAMPMOUSE)./

* Enter all mouse routines (not required for SERVEMOUSE) with the X register set to $Cn and Y register set to $n0 where n = slot number.

* Some programs may need to turn off interrupts for purposes other then reading the mouse. This is sometimes done on the Apple //e to keep from having to handle interrupts are turned off and then back on, the first call to READMOUSE may give incorrect values. Subsequent calls to READMOUSE will return correct values until interrupts are turned off and on again. Turning off interrupts for mouse calls does not create this problem. If you are watching numbers coming from the mouse while moving it in a direction that would increase values you might see the following: 6, 7, 8, 9, 8, 9, 10. In practice, this momentary 'glitch' in the stream of mouse data has little importance and would probably only be noticed by programmer testing his/her program - no one's hand is that steady. If you must keep this 'glitch' from happening then do not keep interrupts off for more then 40 microseconds or be sure that at least one mouse interrupt has taken place since interrupts were turned back on.

Mouse Technical Note #2
15-Mar-84

This technical note explains how to vary the "VBL" interrupts between 50 Hz (European rate) or 60 Hz (North American rate).

For further information contact:
PCS Developer Technical Support
M/S 22-W. Phone (408) 996-1010

/bm

This technical note documents a previously undocumented call to the AppleMouse II firmware which allows the user to set the interrupt rate to 50 Hz, or 60 Hz. (60 Hz is the default, and keeps the mousecard-generated "VBL" interrupts synchronized with the actual VBL rate on a standard North American Apple. 50 Hz is necessary for European machines. "60 Hz" and "50 Hz", as used here, are actually shorthand for the Apple video cycle rates used in North America and Europe, respectively).

Call: TIMEDATA

Offset Location: $Cn1C

Input: Accumulator bit 0 = 0 for 60 Hz

= 1 for 50 Hz

Note: <u>All other accumulator bits are reserved, and MUST be set to 0.</u>

Output: carry bit clear.
screenholes unchanged.

This call must be made before INITMOUSE, and then followed by an INITMOUSE call in order to be effective. If you want to change the interrupt rate in the middle of an application, you must call TIMEDATA, with the appropriate value in the accumulator, and then INITMOUSE (until the INITMOUSE is called, no interrupts will be generated). INITMOUSE will, of course, reset the mouse position, mode, clamps, etc., back to their default values.

If TIMEDATA is never called, then the interrupt rate will default to 60 Hz when INITMOUSE is called.

NOTE: This call exists only on the Mousecard for the //e or][+ and should <u>only</u> be used when you know you are working with a //e or][+.

Mouse Technical Note #3
15-Mar-84

This technical note explains what happens when you turn the mouse on and off through t. mode byte of the SETMOUSE routine.

For further information contact:
PCS Developer Technical Support
M/S 22-W. Phone (408) 996-1010

A) What turning the mouse "off" does:

In the description of SETMOUSE and the mouse mode (see AppleMouse II Users Manual pg. 44), the low-order bit of the mouse mode is said to control "mouse off/mouse on". This is somewhat misleading terminology. When this bit is set to 0, the mouse is off only in the following respects:

(1) the mouse position is not tracked. Any mouse motion is ignored.

(2) READMOUSE calls do not update the status byte or the screen holes (the 6502 firmware makes the READMOUSE command a NOP, and does not even issue the READMOUSE command to the 6805)

(3) button and movement interrupts are not generated, regardless of the other mouse mode bits. "Pure" VBL interrupts can still be generated, however, if bit 3 is set.

B) What turning the mouse "off" doesn't do:

Other mouse functions will continue to work as usual when the mouse is "off". POSMOUSE and CLEARMOUSE will change the mouse position, CLAMPMOUSE will set new clamp values, etc. HOMEMOUSE is an odd case; it will change the mouse position as recorded in the 6805, but this change will not appear in the screen holes until a READMOUSE is done with the mouse "on". In particular:

(1) turning the mouse "off" and "on" with the mode byte does not reset any mouse values, including position, to their defaults. The mouse position retains the last values it had before the mouse was turned off, until it is turned "on" again.

(2) a mode byte of $08 - mouse "off", but VBL interrupt on - will still generate VBL interrupts.

Mouse Technical Note #4
15-Mar-84

This technical note explains a bug in the Mouse Firmware having to do with the way that SERVEMOUSE works.

For further information contact:
PCS Developer Technical Support
M/S 22-W. Phone (408) 996-1010

/bm

There is a bug in the AppleMouse II 6805 firmware which may affect the way SERVEMOUSE works in an application program. If the application program takes more than 1 video cycle (normally, about 16 milliseconds) to respond to a mouse-generated interrupt, then there is a chance that SERVEMOUSE will not claim the interrupt: that is, the 6805 will return an interrupt status byte of $00 (i.e. no Mouse interrupt pending), and the 6502 firmware will set the carry bit (although the interrupt will also be cleared by the SERVEMOUSE call). This can be confusing, and under ProDOS or Pascal it can be lethal. We have identified the following solutions, any one of which should work:

(I) If you are not working under an established system (like ProDOS or Pascal):

(A) don't allow unclaimed interrupts to be fatal to your application. Ignore them.

or

(B) Always service mouse interrupts within 1/60 of a second. If you are forced to turn off interrupts for about that length of time or more, first:

use SETMOUSE to set the mouse mode to 0.

call SERVEMOUSE to clear any existing mouse interrupt.

After interrupts are turned back on, restore the mouse mode.

(II) If you are working under an established operating system, like ProDOS or Pascal, for which unclaimed interrupts are fatal:

(A) If the mouse is the <u>only</u> interrupting device: write your interrupt handler so that it claims all interrupts. That is, regardless of whether the mouse admits to generating the interrupt, clear the carry bit before exiting the interrupt handler, to let ProDOS or Pascal know the interrupt was "serviced".

(B) If the mouse is <u>not</u> the only interrupting device:

(1) Write the mouse interrupt handler to claim all unclaimed interrupts, as described in (II)-(A) above, and make sure the mouse interrupt handler is installed <u>last</u> - otherwise the interrupt will never get through to any interrupt handlers whic

follow the mouse's.

Note: This solution may cause curser flicker by delaying the application's response to VBL interrupts.

or

(2) Write a spurious interrupt handler (also known as a "demon"), not associated with any device, which claims all unclaimed interrupts (that is, clears the carry bit and then exits). For the reason just mentioned, this interrupt handler must be installed <u>last</u>.

Note: Under ProDOS, this cuts down the number of interrupting devices that can be used to 3.

or

(3) Include code in every interrupt handler to check if that interrupt handler is last. If it is, then that interrupt handler should claim any previously unclaimed interrupts, even if its device was not generating it.

Under ProDOS, this would permit 4 interrupting devices; but it may be tricky to implement, and it requires identical code in each interrupt handler which must be executed every time the handler is called.

Note: This bug will be fixed in future versions of the mouse card (but that's not much help, is it?).

Mouse Technical Note #5
4-May-84

This technical note explains how you can go about checking to see if the mouse firmware card you have identified through software is able to support interrupts.

For further information contact:
PCS Developer Technical Support
M/S 22-W. Phone (408) 996-1010

/gs

Checking To See Whether A Mouse Type Device Supports Interrupts.

After identifying a card as a mouse type device it is important to check if the card supports interrupts. There is a convention defined in the mouse firmware protocol that does just that. It defines location $CN11 as a flag to indicate whether or not interrupts can be supported. The value at location $CN11 will be a 0 if interrupts are supported. It is important to check this byte if your program uses interrupts. The reason that you must check this is that the device may not be a mouse at all, rather some other type of device that is emulating a mouse, without interrupt generating capability. (This could be a track ball, graphics tablet, etc.) If the byte at $CN11 is a non zero value then that device does not support interrupts and must be used passively.

If you are a hardware developer and would like your device to emulate a mouse you must follow the mouse protocol as described in the AppleMouse II users guide. (pg 43-49) Your device must also have the same signature bytes as the mouse card. These are $CN0C=$20 and CNFB=$D6. (The N in the above addresses represents the slot number that the card happens to be in at the time. So for slot 4, $CNFB you would have $C4FB.)

Note: The use of location $CN11 is not described in the AppleMouse II users guide. Having your program check this byte is highly recommended since it is quite likely that devices which emulate the mouse will be developed, and some of them may not support interrupts. Through this byte you have a simple way to check if the device supports interrupts.

MOUSE TECHNOTE #6

Revision of general handout on the Apple//e Dec 83*
5-July 84

This technote explains changes that will be made to the Apple//e ROM so that it will support text icons. These icons will be used by the new 'mouse' interface tool kit.

For further information contact:
PCS Developer Technical Support
M/S 22w. Phone (408). 996-1010

Dear Developers:

THIS IS TO NOTIFY YOU THAT APPLE COMPUTER WILL BE MAKING A CHANGE TO THE CHARACTER SET IT USES IN ITS APPLE//e. THIS MAY AFFECT YOUR SOFTWARE.

This new character set will be available to the public in 1984.

WE ARE NOTIFYING YOU OF THIS CHANGE AT THIS TIME BECAUSE SOME CURRENT SOFTWARE MAY NO LONGER FUNCTION CORRECTLY WITH THE NEW CHARACTER SET. UNDER THE RIGHT CONDITIONS AN INVERSE UPPERCASE LETTER WILL NOW BE A GRAPHIC ICON.

The following will help you identify if the changes we are making will affect you or not.

1. If your program is written entirely in BASIC or Pascal or your Assembly Language program calls the COUT routine to put characters on the screen you will not be affected. The only exception would be if you are using BASIC pokes to Poke inverse upper case characters directly to the text screen.

2. If your program is using the standard character set (checkerboard cursor) you will not be affected.

3. If your program is using the alternate character set (solid cursor) and is directly POKING (storing) values to the text display area you will have problems if your character values are from 64 ($40) to 95 ($5F). These values now display inverse uppercase characters plus some special characters. In the future these values will display graphic icons. To recreate the origional displays (which include inverse uppercase plus some special characters) use values in the range from 0 ($0) to 31 ($1F) rather then the values from 64 ($40) to 95 ($5F). Note that using these lower values will work properly on the current character set.

We at Apple are excited about this new extension to the Apple//e's alternate character set. The new icons are similar to those used in LISA and will enhance the use of pointing devices such as a mouse on the Apple//e. If used effectively, the icons, in connection with pointing devices, can significantly simplify the human interface of your programs.

What we foresee as the ways to access these ICONS from various programming languages are described below. We have also included a sample of the current ICON set. More detailed and accurate information will be provided prior to making the ICONS available.

The following method will probably be used for showing ICONS from BASIC:

1) Set up the alternate character set by POKING 49162 ($C00A) with any value then doing a PR#3. If an 80 column card is present you may remain in 80 columns. If there is no card or you want to be in 40 columns PRINT CHR$(11).
2) PRINT CHR$(27) to enable the mouse characters.
3) Use the INVERSE command to set inverse mode.
4) PRINT the appropriate capital letter for the desired ICON. See attached

ICON chart.
Disable the ICONS by PRINTing CHR$(24).

Machine Language programs are expected to follow the same procedure as BASIC. Use calls to COUT to perform the print operations. The following is a sample Machine Language program which will 'print' two ICONS followed by the two inverse uppercase letters that have the same ASCII values.

```
START    STA    $C00A          ;FLIP IN 80 COLUMN FIRMWARE
         LDA    #$A0           ;USE A BLANK TO
         JSR    $C300          ;TURN ON VIDEO FIRMWARE
         LDY    #0             ;INIT COUNTER
LOOP     LDA    STR,Y          ;GET VALUE
         JSR    $FDED          ;SEND IT THROUGH COUT ROUTINE
         INY
         CPY    STRLEN
         BNE    LOOP           ;=>NOT DONE YET
         RTS
STR      DFB    $1B,$46,$47,$18,$46,$47 ;ICONS ON, SHOW, ICONS OFF, SHOW
STRLEN   EQU    *-STR          ;LENGTH OF STR
```

NOTE: 'printing' ICONS on the text screen by directly poking or storing ICON values into the text buffer is not supported.

The probabable method for using the ICONS from Pascal 1.1 will be as follows:

1) Output a chr(27), an escape character, to enable ICONS.
2) Output a chr (15) to turn on inverse video.
3) Output the appropriate capital letter for the desired ICON. See attached ICON chart.
Disable the ICONS by outputting chr(24).

Pascal sample program:

```
program Output_mouse_icon;
var cmd : packed array [ 0..1] of 0..255;
begin
  cmd [ 0 ] := 27;  cmd [ 1 ] := 15;
  Unitwrite (1, cmd, 2 );  {turn on ICON mode }
  {code to display icons...
            .
            .
            .
 }
 cmd [ 0 ] := 24;
 Unitwrite ( 1, cmd, 1 ); {turn off ICON mode }
end.
```